Intelligent Reliability Management in Large-scale Cloud Systems

LIU, Jinyang

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
July 2024

Thesis Assessment Committee

Professor MENG Wei (Chair)

Professor LYU Rung Tsong Michael (Thesis Supervisor)

Professor XU Hong (Committee Member)

Professor KAO Odej (External Examiner)

Abstract of thesis entitled:

Intelligent Reliability Management in Large-scale Cloud Systems

Submitted by LIU, Jinyang

for the degree of Doctor of Philosophy

at The Chinese University of Hong Kong in July 2024

The rapid emergence of cloud services and systems has made them indispensable components of the modern digital landscape. Leading cloud providers like AWS, Microsoft Azure, and Google Cloud Platform operate on an immense scale, serving billions of users and businesses globally. Ensuring the reliability of these cloud systems is paramount, as service interruptions can lead to significant financial impacts. Despite the high priority placed on system reliability, the complexity and scale of cloud environments pose substantial challenges to incident management.

To address these challenges, this thesis explores intelligent reliability management for cloud systems through data-driven approaches. Our research is driven by the heterogeneous data generated by cloud systems, including logs, metrics, alerts, and tickets. We present three significant contributions:

First, we propose Prism, a method to enhance system observability in virtualized cloud environments by inferring functional clusters of instances with similar communication and resource usage patterns. This approach addresses the challenge of degraded observability due to virtualization, which complicates the correlation of issues across different layers. Prism employs a coarse-to-fine strategy: initially, it uses communication patterns to coarsely

divide instances into smaller chunks (trace-based partitioning), followed by fine-grained clustering within each chunk based on resource usage patterns (metric-based clustering). By identifying these functional clusters, Prism provides deeper insights into the relationships between instances and their functionalities, aiding in timely detection and mitigation of issues.

Second, we introduce SeaLog, a scalable and adaptive log-based anomaly detection method. SeaLog leverages the combined strengths of large language models (LLMs) and traditional machine learning techniques to effectively address the challenges of resource constraints and adaptability in dynamic cloud environments. It consists of a lightweight detection agent that efficiently filters normal log data and forwards only suspicious logs to a backbone analyzer. The backbone analyzer, powered by LLMs, comprehends log semantics to identify anomalies, leveraging extensive training on natural language corpora to handle unseen logs and ensure adaptability. Additionally, both components incorporate human feedback to enhance their adaptability over time. SeaLog addresses the impracticality of existing methods in production environments by optimizing for computational and space efficiency, ensuring accurate, lightweight, and adaptive log-based anomaly detection in real-world cloud environments.

Third, we develop a solution, iPACK, for incident-aware duplicate ticket aggregation by leveraging cloud-side runtime information (*i.e.*, alerts). This approach addresses the challenge of efficiently managing and aggregating duplicate support tickets caused by incidents, which is essential to reduce the burden on support engineers and resolve issues more efficiently. Our method involves preprocessing alerts into more coarse-grained events to reduce redundancy, using graph-based incident profiling to filter noisy events and link those caused by the same incident, and implementing an attentive interaction network to correlate tickets with responsible events. By leveraging relationships between

alerts and tickets, this method allows us to aggregate semantically different tickets through alert-alert and ticket-alert links, addressing the diverse symptom descriptions and varied usage scenarios that traditional semantic similarity-based methods fail to capture.

In summary, this thesis leverages data analytics and machine learning techniques to improve the reliability management of cloud systems in large scale. Our proposed methods—Prism, SeaLog, and iPACK—demonstrate promising results in enhancing system observability, anomaly detection, and incident management, ultimately contributing to more reliable and efficient cloud services.

論文題目：大規模雲係統下的智能化可靠性管理

作者　　：劉金楊
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　　：

雲服務和系統的迅速興起使其成為現代數位環境中不可或缺的組成部分。領先的雲提供商如亞馬遜，微軟和谷歌在全球範圍內運營，服務數十億用戶和企業。確保這些雲系統的可靠性至關重要，因為服務中斷可能會導致重大財務損失。儘管雲廠商始終高度重視雲系統的可靠性，但雲環境的複雜性和巨大的規模仍然給事故的管理帶來了巨大挑戰。本論文通過數據驅動的方法探討了雲系統的智能可靠性管理。我們的研究基於雲系統生成的異構數據，包括日誌、指標、鏈路、告警和工單。我們提出了三個重要的貢獻：

首先，我們提出了Prism，一種通過推斷具有相似通信和資源使用模式的實例功能集群來增强虛擬化雲環境中系統可觀測性的方法。這種方法緩解了由於虛擬化導致的雲係統可觀測性下降的問題，這導致跨不同層的問題關聯變得複雜，難以追溯根因。Prism採用一種粗到細的策略：首先使用通信模式將實例粗略地劃分為較小的塊，然後在每個塊內根據資源使用模式進行細粒度聚類（基於指標的聚類）。通過識別這些功能集群，Prism能夠深入了解實例之間的關係及其功能，有助於及時檢測和緩解雲係統中潛在的問題。

其次，我們介紹了SeaLog，一種可擴展且自適應的基於日誌的異常檢測方法。SeaLog結合了大型語言模型（LLMs）和傳統機器學習技術的優勢，有效解決了動態雲環境中資源限制和適應性的挑戰。它包含一個輕量級檢測代理，可以高效過濾正常日誌數據，並僅將可疑日誌轉發到骨幹分析器。骨幹分析器由大模型驅動，可以理解日誌語義以識別異常，利用大量自然語言語料庫的訓練來處理未見過的日誌並確保適應性。此外，這兩個組件都整合了人類反饋以增强其隨時間變化的適應

性。SeaLog通過優化計算和空間效率，緩解了現有方法在生產環境中的不實用性，確保了在實際雲環境中準確、輕量且自適應的基於日誌的異常檢測。

第三，我們提出了一種名為iPACK的解決方案，用於通過利用雲端運行時信息（即告警）來實現重複工單聚合。這種可以有效管理和聚合由事件引起的重複工單，這可以高效地減少支持工程師的負擔，提升問題解決效率。我們的方法包括將告警預處理為更粗粒度的事件以減少冗餘，使用基於圖的事件分析來過濾噪聲事件並鏈接由同一事件引起的告警，並實施一個專注的交互網絡來將工單與負責的事件相關聯。通過利用告警和工單之間的關係，這種方法使我們能夠通過“告警-告警”和“工單-告警”鏈接來聚合語義不同的工單，解決傳統語義相似性方法未能捕捉到的重複工單。

總之，本論文通過數據分析和機器學習技術，提升了雲系統的可靠性監控和工程。我們提出的方法Prism, SeaLog和iPACK，在增强系統可觀測性、異常檢測和事故管理方面展現了良好的效果，最終有助於提供更可靠和高效的雲服務。

Acknowledgement

First and foremost, I would like to express my deepest gratitude to my supervisor, Prof. Michael R. Lyu, for his generous support throughout my four-year study. Prof. Lyu has not only provided comprehensive academic guidance but has also taught me the invaluable lesson of remaining calm in the face of challenges. His wisdom, patience, and encouragement have been instrumental in my academic and personal growth, and I am profoundly grateful for his mentorship. I will always remember the advice he gave me when I felt things were meaningless: "Just enjoy the journey".

I would like to acknowledge the invaluable contributions of my thesis committee members, Prof. Wei Meng and Prof. Henry Xu, for their constructive comments and valuable suggestions on this thesis and all my term presentations. I extend my sincere gratitude to Prof. Odej Kao from the Technical University of Berlin for kindly serving as the external examiner for my thesis.

I was very fortunate to collaborate with exceptional researchers from both industry and academia during my Ph.D. journey. I received invaluable guidance from Dr. Jieming Zhu from Huawei Noah's Ark Lab, Dr. Shilin He and Dr. Yu Kang from Microsoft's STCA DKI group, and Prof. Zhuangbin Chen, Prof. Yuxin Su, and Prof. Zibin Zheng from Sun Yat-sen University.

I want to thank every member in Prof. Michael R. Lyu's ARISE group. In particular, I am deeply grateful to my good friends from our time working together at Huawei Cloud: Zhuangbin Chen, Jiazhen Gu, Jiacheng Shen, Yichen Li, Zhihan Jiang,

Junjie Huang, and Xuchuan Luo. We shared both happiness and sadness, and supported each other through challenging times. Their companionship and support were invaluable during my Ph.D. study.

I want to convey my heartfelt appreciation to my beloved wife, Ms. Qiwen He. She has been my confidant, someone with whom I can share everything. She has always been there, supporting me through any challenges I faced. Happiness has always felt more profound when shared with her. We have always planned our future together, keeping each other in mind. Having her by my side during my Ph.D. journey has been an invaluable and cherished part of my life.

Finally, I would like to thank my family, including my parents, Mr. Jun Liu and Ms. Tiancui Yang, my grandma, Ms. Yu'e Peng, and my parents-in-law, Mr. Huafen He and Ms. Yanli Deng, for their unwavering support and encouragement. Additionally, a special thank you goes to my beloved dog, Goofy, and my cat, Johnny, for their constant companionship and unconditional love.

To my family.

Contents

Abstract **i**

Acknowledgement **vi**

1 Introduction **1**
1.1 Overview 1
1.2 Thesis Contributions 5
1.3 Thesis Organization 8

2 Reliability Management in Cloud Systems **12**
2.1 Architecture of Typical Cloud Systems 13
2.2 Real-time Monitoring, Automatic Alerting and Quick Response 16
2.3 Intelligent Site Reliability Engineering 20
2.3.1 Metric-based Monitoring and Analysis . . . 21
2.3.2 Log-based Monitoring and Analysis 22
2.3.3 Incident and Support Ticket Management . 25

3 Literature Review on Intelligent Cloud Reliability Management **29**
3.1 Metric-based Analysis 29
3.2 Log-based Analysis 31
3.3 Incident Management 33
3.4 Support Ticket Management 34

4 Functional Cluster Identification **35**
4.1 Problem and Contributions 36
4.2 Pilot Study 40
4.3 Methodology 43
4.3.1 Overview 43
4.3.2 Trace-based partitioning 45
4.3.3 Metric-based Clustering 48
4.4 Evaluation 51
4.4.1 Experimental Setting 51
4.4.2 Experimental Results 54
4.5 Industrial Experience 60
4.5.1 Vulnerable Deployment Identification . . . 61
4.5.2 Latent Issue Discovery 63
4.6 Threats to Validity 65
4.7 Summary 66

5 LLM-enhanced Log Anomaly Detection **68**
5.1 Problem and Contributions 69
5.2 Background and Motivation 73
5.2.1 Log-based Anomaly Detection 73
5.2.2 Characteristics of Log Data in Cloud Systems 74
5.3 Methodology 77
5.3.1 Overview 77
5.3.2 Detection Agent 78
5.3.3 Backbone Analyzer 84
5.3.4 Deployment 87
5.4 Evaluation 88
5.4.1 Experimental Setting 88
5.4.2 Experimental Results 91
5.5 Industrial Experience 97
5.6 Threats to Validity 99
5.7 Summary 100

6 Incident-aware Ticket Aggregation 102
6.1 Problem and Contributions . . . 103
6.2 Motivation . . . 107
6.2.1 Background . . . 107
6.2.2 Alert-Alert Relation . . . 108
6.2.3 Ticket-Alert Relation . . . 109
6.2.4 A Motivating Example . . . 110
6.3 Methodology . . . 112
6.3.1 Overview of iPACK . . . 112
6.3.2 Alert Parsing . . . 114
6.3.3 Incident Profiling . . . 115
6.3.4 Ticket-Event Correlation . . . 118
6.3.5 Deployment . . . 121
6.4 Evaluation . . . 123
6.4.1 Experimental Setting . . . 123
6.4.2 Experimental Results . . . 126
6.5 Industrial Experience . . . 131
6.6 Threats to Validity . . . 133
6.7 Summary . . . 134

7 Conclusion and Future Work 135
7.1 Conclusion . . . 135
7.2 Future Work . . . 137
7.2.1 LLM-driven Distributed Diagnosis . . . 138
7.2.2 Reliability of LLM Training Systems . . . 140

8 List of Publications 142

Bibliography 146

List of Figures

2.1 Reliability Management of Cloud Systems 15
2.2 An Example of Alerts 18
2.3 An Example of Log Parsing 23
2.4 An Example of Support Tickets. 27

4.1 The Abstraction of Cloud Systems 36
4.2 Results of the Study on Communication and Resource Usage Patterns. 42
4.3 The Overall Workflow of Prism 44
4.4 Parameter Sensitivity of Prism 57
4.5 Case I: Vulnerable Deployment Identification . . . 61
4.6 Case II: Latent Issue Discovery 63

5.1 Statistics of Log Messages in Huawei Cloud 75
5.2 The Overall Workflow of Sealog 78
5.3 The Structure of N-gram Probabilistic Tree (NPT) 81
5.4 Processing Suspicious Windows with Backbone Analyzer 86
5.5 Experimental Results of Online Anomaly Detection 93
5.6 Anomaly Detection Performance w.r.t. the Number of Queries 94
5.7 Experimental Results of Time and Space Efficiency Comparison 96
5.8 Practical Application of Sealog in Huawei Cloud . 98

6.1 An Example of An Alert and Its Resultant Ticket . 107
6.2 Statistics of Alert and Incident Data in Azure . . . 109

6.3 Time Interval between Alerts And Resultant Tickets 109
6.4 Ticket Number Trend during An Incident 109
6.5 The Overall Framework of iPACK. 113
6.6 The Overall Framework of AIN. 120
6.7 The Effectiveness of Graph-based Profiling (GIP) . 130
6.8 A Success Case of iPACK in Azure 132

List of Tables

4.1 Dataset Statistics 52
4.2 Effectiveness of Functional Cluster Discovery . . . 54
4.3 Contribution of Different Components in Prism . . 55
4.4 Efficiency Comparison with Increasing Scales of Instances 59

5.1 Content of Word-level LRU Sketch 82
5.2 Statistics of Evaluation Datasets 90
5.3 Experimental Results of End-to-end Offline Anomaly Detection 91

6.1 Alerts Caused by The Same Incident and The Resultant Tickets 111
6.2 Effectiveness of Aggregating Duplicate Tickets Caused by the Same Cloud Incident 124
6.3 Effectiveness of Correlating a Ticket to An Event . 129

Chapter 1

Introduction

1.1 Overview

Cloud services and systems are rapidly emerging as essential components in the modern digital landscape, driven by a vast and expanding market projected to reach $832.1 billion by 2025 [72]. Typical cloud providers such as Microsoft Azure, Amazon Web Services (AWS), and Google Cloud Platform (GCP) operate on an immense scale, serving billions of users and businesses globally. AWS, the market leader, boasts over a million active customers, including major corporations, startups, and government agencies. Microsoft Azure supports over 95% of Fortune 500 companies [113]. These platforms collectively power a significant portion of the internet, providing critical infrastructure and services to a vast and diverse user base. Ensuring the reliability of these cloud systems is of paramount importance, as any service interruption can have significant financial impacts. According to a Parametrix report[1], an AWS outage happening on the "us-east-1" region could cost more than one-third of Fortune 500 companies $3.4 billion in 24 hours and $7.8 billion in 48 hours. A 24-hour halt across multiple AWS regions could result in a $9.5 billion loss, escalating to $20.2 billion in 48 hours. Cloud providers have

[1]https://www.parametrixinsurance.com/in-the-news/fortune-500-vulnerable-to-cloud-risks-could-suffer-losses-of-20-billion

placed a high priority on improving the reliability of their systems, recognizing the critical importance of maintaining uninterrupted service for their vast and diverse user base.

Monitoring is foundational for ensuring the reliability and performance of cloud systems. Leading cloud providers like AWS, Microsoft Azure, and Google Cloud Platform integrate a wide array of monitoring tools within their services to continuously track system health and performance. These tools generate a wealth of data, including logs [35, 108], metrics [144, 171], traces [169, 180], alerts [111, 174], and tickets [47, 57], which provide deep insights into system operations. By analyzing this data, cloud providers can quickly identify and address potential issues, optimize resource utilization, and maintain high levels of service availability.

When an incident (*i.e.*, unexpected interruption or downgrade of services) occurs, on-call engineers (OCEs) typically use basic tools to analyze the generated data, identify the root cause, and diagnose the issue. For example, if a major e-commerce platform hosted on AWS experiences a significant slowdown, engineers would begin by examining the logs for any error messages or anomalies. They might notice a spike in CPU usage metrics and trace logs indicating a specific service or microservice is consuming excessive resources. By correlating these findings with recent changes or deployments, such as a new software update or configuration change, they could identify that a recent update introduced a memory leak causing the service to degrade. With this diagnosis, they can roll back the update or apply a patch to resolve the issue, restoring normal service performance. This analysis of logs, metrics, and traces is critical for pinpointing the exact cause of the problem. Such a manual practice has two major limitations. (1) It can be time-consuming and prone to human error, as engineers must sift through vast amounts of data, including logs, metrics, and traces, to identify the root cause of a problem. This process can lead to delays in resolving issues, resulting

in prolonged service disruptions. (2) It requires in-depth domain knowledge of cloud infrastructure and applications, which can be challenging to maintain given the rapid evolution and complexity of cloud technologies.

However, the vast amounts of data generated by cloud systems also present opportunities to address these limitations in a data-driven paradigm. By leveraging machine learning, deep learning, and even large language models, we can automate the analysis and correlation of data from various sources. Nevertheless, it is challenging to develop these solutions in real-world cloud systems. We summarize the challenges as follows:

- Large Scale of Cloud Systems: Cloud environments often encompass thousands of servers, virtual machines, and numerous applications running simultaneously. Managing and processing the sheer volume of data generated in such large-scale systems requires significant computational resources and sophisticated algorithms that can handle high-dimensional data efficiently.

- Complicated Dependency Between Different Cloud Components: Cloud systems are composed of numerous interconnected components, such as databases, storage systems, networking elements, and various services. These components often have intricate dependencies, making it difficult to isolate issues and understand their ripple effects across the system. Developing models that can accurately capture and analyze these dependencies is a complex task.

- Fast-Evolving Environment: Cloud technologies and infrastructures are constantly evolving, with frequent updates, new features, and changes in configurations. This rapid evolution poses a challenge for maintaining and updating AI models, as they must continuously adapt to new patterns and behaviors in the system. Ensuring that machine learning and deep

learning models remain effective in such a dynamic environment requires ongoing retraining and validation.

- Model Interpretability: Ensuring that AI models are interpretable and their predictions understandable by human operators is crucial for gaining trust and facilitating their adoption. Complex models, such as deep learning networks, often lack transparency, making it difficult to explain their decisions.

- Data Availability and Privacy Concerns: Cloud vendors are often highly concerned about data privacy and security, which leads to restrictions on the availability of detailed operational data. As a result, only limited low-level data is accessible to external AI systems. This restricted access makes it challenging to gain comprehensive insights into the cloud environment.

We conduct research on intelligent reliability management of cloud systems to address these challenges, covering the comprehensive pipeline of cloud systems, as illustrated in Figure 2.1. Our studies are driven by the heterogeneous data generated by cloud systems. First, we propose a method called Prism to gain more insights from limited fundamental data and improve system observability. Prism infers functional clusters among large-scale virtual instances, facilitating the identification of reliability issues in cloud systems. Second, we present a synergistic approach named Sealog, which fuses large language models (LLMs) with traditional machine learning methods to detect log-based anomalies in an evolving environment. Finally, we introduce iPACK, a solution designed to optimize the ticket and alert processing procedure by linking them together. This integration helps support engineers significantly reduce duplicate efforts during an incident. Through these innovative approaches, we aim to enhance the reliability

and efficiency of cloud systems, addressing the challenges posed by their scale, complexity, and dynamic nature.

1.2 Thesis Contributions

We summarize the contribution of this thesis as follows:

1. **System Observability Enhancement via Inferring Functional Clusters**

   Cloud systems typically use virtualization techniques to abstract hardware resources, such as computation, storage, and networks, into instances (e.g., virtual machines). This architecture provides flexibility and elasticity, allowing tenants to subscribe to various instances to run services with different functionalities, enabling the creation of complex and customizable applications. However, virtualization introduces significant challenges in ensuring cloud system reliability by degrading system observability. The additional abstraction layer between hardware and applications makes it difficult to correlate issues across different layers. Despite using monitors to collect runtime data, cloud vendors still view instances as isolated black boxes, lacking insight into application deployment across the infrastructure. This complicates assessing the impact of platform-level issues on applications.

   To tackle the problem of degraded system observability in virtualized cloud environments, we propose identifying functional clusters of instances, where each cluster contains instances with similar functionalities. We first present a pilot study on the internal services of a cloud provider (e.g., Huawei Cloud) using only external monitoring data to identify instances with similar communication and resource usage patterns. We then develop a clustering solution called Prism,

which employs a coarse-to-fine strategy: first, we use communication patterns to coarsely divide instances into smaller chunks (trace-based partitioning), and then we perform fine-grained clustering within each chunk based on resource usage patterns (metric-based clustering). This method enhances system reliability by providing deeper insights into the relationships between instances and their functionalities, aiding in timely detection and mitigation of issues, and efficiently handling the large scale of cloud systems.

2. **Scalable and adaptive log-based anomaly detection.**

   Logs in cloud systems, like those in traditional software systems, are invaluable for understanding system functionality and identifying issues, making log-based anomaly detection essential for cloud management. However, existing approaches face significant challenges. Traditional machine learning methods, such as isolation forest (IF), support vector machine (SVM), decision tree (DT), and logistic regression (LR), along with recent deep learning methods that extract semantic information from logs, although effective on benchmark datasets, are impractical for production environments. This impracticality stems from two main issues: resource constraints and adaptability. These solutions often prioritize detection accuracy without optimizing for computational and space efficiency, making them unsuitable for resource-constrained cloud instances. Additionally, deep learning methods require GPUs for real-time inference, which are not always available, and transmitting large-scale log data to centralized nodes incurs significant network and I/O overhead. Furthermore, the adaptability of these methods is limited, as they struggle to handle evolving log data due to frequent software updates, leading to performance degradation in real-world scenarios.

To tackle the limitations of existing studies, we propose a scalable and adaptive log-based anomaly detection method named SeaLog. SeaLog integrates the strengths of large language models (LLMs) and traditional machine learning methods to address resource constraints and adaptability issues. It consists of a lightweight detection agent that efficiently filters normal log data and forwards only suspicious logs to a backbone analyzer. The backbone analyzer, powered by LLMs, comprehends log semantics to identify anomalies, leveraging extensive training on natural language corpora to handle unseen logs and ensure adaptability. Additionally, both components incorporate human feedback to enhance their adaptability over time. By combining the lightweight efficiency of traditional machine learning with the semantic understanding and few-shot learning capabilities of LLMs, SeaLog provides an accurate, lightweight, and adaptive solution for log-based anomaly detection in real-world cloud environments.

3. **Incident-aware duplicate ticket aggregation**

   Cloud platforms serve millions of users who submit support tickets when encountering technical problems. These tickets, submitted by customers, include a textual description of the issue and some basic attributes. Timely assistance is crucial for cloud providers to avoid user dissatisfaction and financial loss. Incidents, or unexpected service interruptions, are inevitable in large-scale cloud platforms and can trigger numerous support tickets, many of which may be duplicates reported in a distributed and uncoordinated manner. Efficiently aggregating these duplicate tickets is essential to reduce the burden on support engineers and resolve issues more efficiently. Existing solutions for aggregating duplicate support tickets in cloud systems have significant limitations. They primarily rely on measuring the semantic similarity be-

tween the textual descriptions of tickets using natural language processing techniques. However, these approaches are inadequate because customers using the same service may encounter different issues due to various usage scenarios, and multiple services can be impacted by the same incident, leading to diverse symptom descriptions in tickets. Consequently, relying solely on textual descriptions is insufficient for accurately clustering duplicate tickets caused by the same incident.

To tackle these limitations, we propose introducing cloud-side runtime information (alerts) to facilitate ticket aggregation. By leveraging the relationships between alerts and tickets, we can accurately cluster duplicate tickets caused by the same incident. This approach involves preprocessing alerts into more coarse-grained events to reduce redundancy, using graph-based incident profiling to filter noisy events and link those caused by the same incident, and implementing an attentive interaction network to correlate tickets with responsible events. This method allows us to aggregate semantically different tickets through alert-alert and ticket-alert links, addressing the diverse symptom descriptions and varied usage scenarios that traditional semantic similarity-based methods fail to capture.

1.3 Thesis Organization

The remainder of this thesis is organized as follows:

1. **Chapter 2: Reliability Management in Cloud Systems**

   Chapter 2 introduces the background and challenges of reliability management in cloud systems. Section 2.1 describes the typical architecture of cloud systems. Section 2.2 then

discusses common practices for managing reliability in these systems. Finally, Section 2.3 presents intelligent solutions that can be integrated into reliability management practices.

2. **Chapter 3: Review on Intelligent Cloud Reliability Management**

   Chapter 3 reviews fruitful studies on cloud reliability management in the literature. We cover a range of approaches including metric-based analysis, log-based analysis, incident management, and support ticket management. Additionally, we discuss the limitations of these studies in this section which we try to address in the following sections.

3. **Chapter 4: Functional Cluster Identification**

   Observability is fundamental to ensuring cloud reliability. However, the virtualization of cloud environments can degrade observability. Chapter 4 proposes a method called Prism to infer functional clusters based on limited available data while protecting users' privacy. Specifically, Section 4.1 first introduces the problem and summarizes the contributions made in improving cloud observability. Then, Section 4.2 presents a pilot study based on the analysis of data collected from Huawei Cloud to motivate our method design. The following Section 4.3 describes the details of the proposed method, Prism. Section 4.4 provides a comprehensive evaluation of the proposed method. Section 4.5 validates the usefulness of Prism using two real-world cases. Section 4.6 discusses the threats to the validity of this study. Finally, Section 4.7 summarizes this chapter.

4. **Chapter 5: LLM-enhanced Log Anomaly Detection**

   Log data is a crucial source of information that captures software runtime details, assisting developers in the diagnosis of various problems. Automatic anomaly detection on

log data has also played a key role in monitoring cloud systems. However, existing solutions are often either inefficient or not adaptive. In Chapter 5, we propose a scalable and adaptive solution called Sealog to address these limitations. Specifically, Section 5.1 provides a general introduction to the problem of log-based anomaly detection and summarizes our contributions. Then, Section 5.2 elaborates on the detailed background of this problem and our motivation. Next, Section 5.3 introduces the design details of Sealog, aiming to achieve both scalability and adaptiveness. Section 5.4 extensively evaluates our method. Section 5.5 shares our experience in validating Sealog in Huawei Cloud. Section 5.6 discusses threats to the validity of this study. Finally, we summarize this chapter in Section 5.7.

5. **Chapter 6: Incident-aware Ticket Aggregation**

   Support tickets are submitted by customers seeking assistance from cloud providers. However, when unexpected failures occur in cloud systems, a large number of users can be affected, leading to numerous tickets being received by cloud providers. It is crucial to aggregate duplicate tickets to reduce the processing effort. In Chapter 6, we propose an incident-aware ticket aggregation solution called iPACK. Specifically, Section 6.1 provides an overview of the problem of ticket aggregation and summarizes our contributions. Following this, Section 6.2 presents a motivating example to illustrate the need for our method design. Section 6.3 then introduces the two-stage linking strategy of iPACK. In Section 6.4, we provide a comprehensive evaluation using real-world data collected from Azure. Additionally, Section 6.5 discusses the industrial experience of evaluating iPACK within Azure. Section 6.6 addresses the threats to the validity of this study. Finally, Section 6.7 summarizes this chapter.

6. **Chapter 7: Conclusion and Future Work** In this chapter, we summarize this thesis and discuss our future work. We plan to focus on LLM-driven diagnosis in large-scale distributed systems. Additionally, we aim to explore methods to enhance the reliability of LLM training systems.

□ **End of chapter.**

Chapter 2

Reliability Management in Cloud Systems

Reliability management is a critical aspect of maintaining robust and dependable cloud systems. It encompasses a range of practices and methodologies aimed at ensuring that services remain available, performant, and resilient under various conditions. One of the key frameworks that support reliability management in modern cloud environments is Site Reliability Engineering (SRE). SRE is a practical discipline that merges software engineering with IT operations to ensure scalable and highly reliable software systems. Originating at Google, SRE focuses on defining Service Level Objectives (SLOs), managing error budgets, and emphasizing automation, monitoring, and proactive incident management. SRE roles include site reliability engineers and managers, who work to improve system reliability, efficiency, and scalability through engineering practices and collaboration between development and operations teams. While SRE offers significant benefits like improved reliability and efficiency, it requires the engineers to have a diverse skill set and careful balancing of priorities for a cloud provider.

As cloud systems, such as those provided by Azure and AWS, grow increasingly larger, SRE faces significant challenges, particularly in maintaining reliability. The sheer scale of modern cloud in-

frastructures demands advanced automation, sophisticated monitoring, and proactive incident management to ensure system stability. Balancing the need for rapid development with stringent reliability goals becomes more complex, requiring SRE teams to continuously innovate and adapt their practices to manage the heightened demands of expansive cloud environments effectively. To address the large scale and complexity of cloud systems and ensure reliability, tools for real-time monitoring and automatic analysis have been developed and extensively studied. In addition, mature cloud systems, such as Azure, have established standard reliability data management protocols. In the following, we first introduce the hierarchical architecture of typical cloud systems in Section 2.1. Then we introduce the common reliability management framework of modern cloud systems in Section 2.2. Finally, we introduce how intelligence can be integrated within this framework 2.3.

2.1 Architecture of Typical Cloud Systems

Cloud computing services are generally categorized into three primary models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Each model offers varying levels of control, flexibility, and management to meet different business needs. IaaS provides the fundamental building blocks of computing infrastructure, PaaS offers a platform for developing and deploying applications, and SaaS delivers fully functional software applications over the internet.

Infrastructure as a Service (IaaS) Layer. IaaS is the most basic cloud service model, providing virtualized computing resources over the internet. It offers essential infrastructure components such as virtual machines, storage, and networking. Users can rent these resources on a pay-as-you-go basis, which allows for significant cost savings compared to maintaining physical hardware.

IaaS provides high flexibility and scalability, enabling businesses to quickly scale up or down based on demand. However, users are responsible for managing the operating systems, applications, and middleware, which requires technical expertise. Examples of IaaS providers include Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP).

Platform as a Service (PaaS) Layer. PaaS sits atop IaaS and offers a higher level of abstraction, providing a platform that allows developers to build, deploy, and manage applications without worrying about the underlying infrastructure. PaaS includes development tools, middleware, database management systems, and runtime environments, which streamline the application development process. This model enhances productivity by simplifying the complexities associated with hardware and software management, allowing developers to focus solely on coding and application logic. PaaS is particularly useful for collaborative projects and continuous integration/continuous deployment (CI/CD) workflows. Examples of PaaS providers include Google App Engine, Microsoft Azure App Service, and Heroku.

Software as a Service (SaaS) Layer. SaaS is the most comprehensive cloud service model, delivering fully functional software applications over the internet. Users can access these applications via web browsers, eliminating the need for installation, maintenance, and management of the software. SaaS applications are typically subscription-based, offering features like automatic updates, scalability, and accessibility from any location with internet connectivity. This model is ideal for businesses looking to reduce the burden of IT management and focus on core activities. SaaS covers a wide range of applications, including customer relationship management (CRM), enterprise resource planning (ERP), email, collaboration tools, and specialized industry-specific software. Examples of SaaS providers include Salesforce, Microsoft Office 365, and Google Workspace.

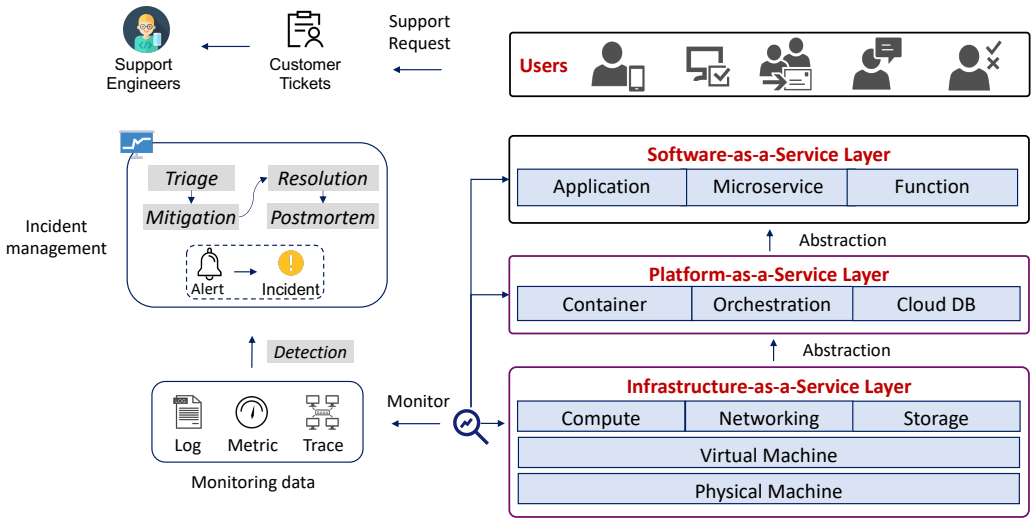


Figure 2.1: Reliability Management of Cloud Systems

Cross-layer Interactions. Within a cloud provider, the interaction between IaaS, PaaS and SaaS is a seamless integration where each layer builds upon the capabilities of the one beneath it. IaaS provides the essential virtualized resources such as computing power, storage, and networking, forming the foundation. PaaS leverages these IaaS resources to offer a managed environment with development tools and middleware, simplifying the process of creating and deploying applications. SaaS, in turn, utilizes the underlying PaaS and IaaS layers to deliver fully functional software applications over the internet, accessible to end-users without the need to manage any infrastructure. This hierarchical interaction ensures flexibility, scalability, and efficiency, enabling cloud providers to offer comprehensive and integrated cloud solutions.

2.2 Real-time Monitoring, Automatic Alerting and Quick Response

Real-time Monitoring

Cloud system monitoring is a critical practice in managing and maintaining the health, performance, and security of cloud-based applications and infrastructure. It involves continuously observing and analyzing the various components of a cloud environment to ensure they are functioning optimally and to quickly identify and address any issues that arise. Specifically, typical monitoring practice continuously collects *metrics, logs*, and *traces* generated by various components within the cloud environment. Metrics provide quantitative measurements such as CPU utilization, memory usage, disk I/O, and network traffic, offering insights into resource performance and health. Logs capture detailed records of events, including application operations, system events, and security incidents, providing a chronological account of activities. Traces track the flow of requests through distributed systems, helping to identify bottlenecks and failures by capturing the lifecycle and performance of requests across different services. Tools like Amazon CloudWatch [4], Azure Monitor [111] and Google Cloud Operations Suite [44] facilitate effective cloud system monitoring. By leveraging these tools, organizations can proactively detect and resolve issues, optimize performance, ensure security and compliance, and manage costs, ultimately maintaining an optimal cloud environment.

Automatic Alerting

Automatic alerting is a pivotal component of cloud system monitoring, enabling organizations to promptly react to anomalies and potential issues before they escalate into significant problems. This process involves setting up predefined thresholds and condi-

tions on various *metrics, logs*, and *traces* to trigger alerts when deviations from expected behavior are detected.

Metrics-Based Alerting: Monitoring the health of cloud systems using *metrics* is a critical practice to ensure the reliability, performance, and availability of cloud services. Specifically, metrics denote quantitative measures that provide insights into the performance and health of cloud systems. Examples include CPU usage, memory utilization, disk I/O, network latency, and error rates. In particular, another time-series data, *i.e.*, *Key Performance Indicators (KPIs)* are more often tied to business objectives and service level agreements (SLAs). Examples include uptime percentage, response time, transaction throughput, and user satisfaction scores. Alerts can be configured based on specific metrics or KPIs. For instance, if CPU usage exceeds a certain percentage, an alert can be triggered to notify the operations team of potential resource saturation.

Log-Based Alerting: Logs are detailed records of events that occur within a cloud environment, capturing a wide range of information about the operations, errors, and activities of applications and systems. These logs are crucial for understanding the behavior and performance of cloud services, as they provide a chronological account of events that can be analyzed to diagnose issues, track user activities, and ensure security compliance. Logs can include application logs, system logs, and security logs, each providing different perspectives on the state and health of the cloud environment. Log-based alerting involves monitoring these logs for specific patterns, keywords, or anomalies that indicate potential problems or security threats. Alerts are produced when predefined conditions are met, such as the occurrence of error messages, multiple failed login attempts, or unusual activity patterns. For example, an alert can be configured to trigger if there are repeated instances of a particular error code within a short time frame, indicating a recurring issue that needs immedi-

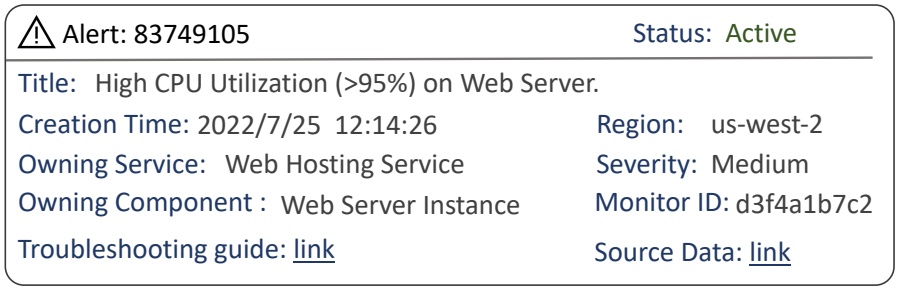


Figure 2.2: An Example of Alerts

ate attention.

Trace-Based Alerting: Traces provide a detailed view of the flow of requests through various services and components in a distributed cloud environment, capturing the lifecycle and performance of each request. This information is crucial for understanding the end-to-end journey of requests, identifying bottlenecks, and diagnosing performance issues or failures in complex, microservices-based architectures. Traces typically include *spans*, which are individual units of work within a trace, detailing start and end times, duration, and any errors encountered. Trace-based alerting denotes monitoring these traces for specific performance metrics, anomalies, or error patterns that could indicate potential issues. Alerts are produced when predefined conditions are met, such as unusually high latency, increased error rates, or unexpected changes in the request flow. For example, an alert can be configured to trigger if the response time for a critical service exceeds a certain threshold, indicating a performance degradation that requires immediate investigation.

Alerts produced from different data sources share a similar format. An alert in a cloud environment typically contains several key pieces of information to help the recipient quickly understand the nature of the issue and take appropriate action as shown in Figure 2.2. For example, an alert might be named "High CPU Utilization on Web Server" with a severity level of "Critical" and a timestamp indicating when it was triggered. It would detail

the specific metric that caused the alert, such as CPU utilization exceeding 90% for five consecutive minutes, and provide the current value, like 95%. The alert would identify the affected resource, including its type (e.g., virtual machine), ID, name, and region. Contextual information would include historical data showing CPU utilization over the past hour, related metrics like memory utilization and network traffic, and recent logs and traces that highlight relevant events and request flows. The alert would also offer recommended actions, such as investigating recent deployments, scaling up resources, optimizing application performance, and notifying relevant team members. Notification channels would be specified, indicating that the alert can be sent via email, SMS, and integrated into an incident management system and involve quick response of on-call engineers (OCEs). Additional information might include links to troubleshooting guides and support contacts, and a visualization such as a CPU utilization graph to aid in quick assessment. This structured format ensures the alert is comprehensive, actionable, and easy to understand, enabling quick and effective responses to potential issues in the cloud environment.

Quick Response

Quick response to alerts is crucial for minimizing downtime and mitigating the impact of issues on cloud-based applications and services. Once an alert is triggered, it is essential to have a well-defined process for responding to and resolving the issue. When a serious failure occurs, it often generates multiple related alerts, which can be grouped together into a single **incident** to provide a comprehensive view of the issue. This incident is then automatically assigned to the appropriate on-call engineer or team based on predefined escalation policies, ensuring that the right personnel are notified and can take immediate action. This process helps identify and address issues within the cloud systems efficiently, en-

suring that even complex failures are managed effectively.

In addition to internal alerts and incidents, issues can also be identified through customer-reported **support tickets**. Customers may report problems by submitting support tickets through a customer support portal. Support engineers review these tickets, prioritize them based on severity and impact, and respond with appropriate solutions. In cases where the issue aligns with an existing alert or incident, support engineers collaborate with the OCEs to ensure a coordinated response. This correlation between support tickets and internal alerts helps in accurately identifying and resolving issues that may impact customers.

After an issue is resolved, a postmortem review can be conducted to analyze the root cause, assess the effectiveness of the response, and identify areas for improvement. This review process helps refine alerting rules, response procedures, and overall system resilience. By establishing a streamlined process for quick response, organizations can effectively manage and resolve issues, ensuring minimal disruption to their cloud services and maintaining high levels of customer satisfaction.

2.3 Intelligent Site Reliability Engineering

Manual processing of reliability data in cloud systems presents significant challenges, primarily due to the scale and complexity of modern cloud environments. With numerous components generating vast amounts of data in the form of metrics, logs, and traces, it becomes increasingly difficult for human operators to effectively monitor, analyze, and respond to issues in real-time. The sheer volume of data can overwhelm teams, leading to missed alerts, delayed responses, and potential system failures. Additionally, the complexity of distributed systems, with their intricate dependencies and interactions, makes it challenging to accurately diagnose problems and identify root causes manually. This can result in

inefficient troubleshooting, prolonged downtimes, and increased operational costs.

To address these challenges and enhance operational efficiency, it is essential to leverage intelligent and automatic analysis of reliability data. By incorporating advanced analytics, machine learning, and automation tools, organizations can proactively detect anomalies, predict potential issues, and streamline incident response processes. Intelligent systems can continuously analyze large datasets, identify patterns, and provide actionable insights faster and more accurately than manual methods. Embracing these technologies not only improves the reliability and performance of cloud systems but also allows teams to focus on strategic initiatives and innovation, ultimately driving better business outcomes.

2.3.1 Metric-based Monitoring and Analysis

In modern cloud systems, the sheer volume and diversity of metrics generated can be overwhelming, making it essential to implement automatic *anomaly detection* and *clustering* techniques. Automatic anomaly detection helps identify unusual patterns or behaviors in real time, ensuring prompt responses to potential issues. Simultaneously, metric-based clustering groups similar data points, enabling the identification of patterns, trends, and anomalies at scale. Together, these techniques aim to provide deeper insights into system performance and usage patterns, facilitating more informed decision-making and resource optimization.

Metric-based Anomaly Detection

Metric-based anomaly detection is a technique used to identify unusual patterns or behaviors in system metrics, indicating potential issues such as hardware failures, security breaches, or performance bottlenecks. It involves monitoring various quantitative measures

like CPU usage, memory utilization, and network latency to detect deviations from the norm. Techniques for anomaly detection include statistical methods (e.g., Z-score, moving average), machine learning (e.g., supervised, unsupervised learning), time series analysis (e.g., ARIMA), and deep learning (e.g., RNNs, autoencoders). The process involves data collection, preprocessing, baseline establishment, anomaly detection, alerting, and continuous evaluation.

Metric-based Clustering

Metric-based clustering is a technique that groups similar data points based on specific metrics or features, aiding in the identification of patterns, trends, and anomalies within cloud systems and IT infrastructure. By employing clustering algorithms such as K-Means, hierarchical clustering, DBSCAN, and Gaussian Mixture Models, organizations can optimize resource allocation, detect anomalies, analyze performance, and plan capacity. The process involves collecting and preprocessing data, selecting relevant features, choosing an appropriate algorithm, training the model, and interpreting the results. Metric-based clustering enhances system understanding, enables proactive management, and improves efficiency by identifying and grouping similar workloads and usage patterns.

2.3.2 Log-based Monitoring and Analysis

Log data in a cloud environment is generated at a large scale and often consists of unstructured free text written by engineers, making it challenging to be handled by machine learning-based solutions directly. To manage this complexity, log parsing is essential as it transforms unstructured logs into structured formats, enabling more effective feature extraction and learning. Once parsed, the structured log data can be utilized for log-based anomaly

Raw Logs

```
1.Sep 18 08:45:36 LabSZ sshd[24200] INFO Running task 1.0 in stage 0.0 (TID 0).
2.Sep 18 08:45:37 LabSZ sshd[24200] INFO Started reading broadcast variable 0.
3.Sep 18 08:47:01 LabSZ sshd[24241] INFO Running task 2.0 in stage 0.0 (TID 1).
4.Sep 18 08:47:22 LabSZ sshd[24265] DEBUG Partition rdd_2_1 not found, computing it.
5.Sep 18 08:49:35 LabSZ sshd[24301] DEBUG Partition rdd_2_2 not found, computing it.
                                        ……
```

Log Parsing

Log Parsing Results

Timestamp	Component	Level	Log Templates	Parameters
Sep 18 08:45:36	LabSZ sshd[24200]	INFO	Running task <*> in stage <*> (TID <*>)	1.0 | 0.0 | 0
Sep 18 08:45:37	LabSZ sshd[24200]	INFO	Started reading broadcast variable <*>	0
Sep 18 08:47:01	LabSZ sshd[24241]	INFO	Running task <*> in stage <*> (TID <*>)	2.0 | 0.0 | 1
Sep 18 08:47:22	LabSZ sshd[24265]	DEBUG	Partition <*> not found, computing it	rdd_2_1
Sep 18 08:49:35	LabSZ sshd[24301]	DEBUG	Partition <*> not found, computing it	rdd_2_2

Figure 2.3: An Example of Log Parsing

detection, identifying unusual patterns that may indicate system issues, and log-based root cause analysis, pinpointing the underlying causes (root cause-indicating components and logs) of a failure.

Log Parsing

Log parsing aims to transform unstructured free text into a structured format that can be easily analyzed. This process automatically extracts *log templates* (or log events) and *log parameters* from raw log entries as shown in Figure 2.3. Log templates represent the fixed part of the log message, while log parameters are the variable parts that change with each log entry. For example, consider the raw log message: "User user4bxs failed to login due to incorrect password.", its log template is "User * failed to login due to incorrect password."

Log-based Anomaly Detection

Log-based anomaly detection aims to identify unusual patterns or behaviors within log data, which can indicate potential issues such as system failures, security breaches, or performance bottlenecks. This approach leverages the structured data obtained through log parsing to detect anomalies in both the sequence and parameters of log entries.

Sequence Anomaly Sequence anomaly detection focuses on identifying deviations in the order or structure of log events. In a typical system, certain sequences of log entries are expected based on normal operational workflows. For example, a successful login sequence might include events such as "User authentication initiated", "User credentials verified", and "User login successful". If an unexpected sequence occurs, such as "User authentication initiated" followed by "User login failed," it could indicate an anomaly. Detecting such sequence anomalies helps in identifying abnormal behaviors or potential issues in the system's processes.

Parameter Anomaly Parameter anomaly detection involves identifying unusual values or patterns in the parameters extracted from log entries. Each log entry contains specific parameters, such as user IDs, timestamps, error codes, and resource usage metrics (*e.g.*, request latency). By analyzing these parameters, it is possible to detect anomalies that deviate from the expected range or pattern. In general, it is common to extract the parameters and utilize a metric-based anomaly detection solution to detect such parameters. For instance, if the parameter "response time" in a web server log consistently exceeds the normal threshold, it could indicate a performance issue or a potential bottleneck. Detecting parameter anomalies enables the identification of issues at a granular level, facilitating more precise and effective troubleshooting.

Log-based Root Cause Analysis

Log-based root cause analysis (RCA) aims to identify the underlying causes of issues or anomalies within a system by analyzing log data. This technique leverages the structured information obtained through log parsing to trace the sequence of events and pinpoint the exact origin of a problem. The process usually consists of collecting and preprocessing log data from various sources, correlating events to build a comprehensive timeline, and utilizing log-based anomaly detection to identify unusual patterns. By analyzing these correlated events and anomalies, the root cause of the issue can be traced back to the initial triggering event or condition. In practice, visualization tools and detailed reporting help represent the sequence of events and the identified root cause clearly, providing actionable insights. For example, in a scenario where a web application experiences intermittent downtime, log-based RCA might reveal that a misconfigured database connection pool is unable to handle peak traffic, leading to server crashes. By automatically identifying and addressing such root causes, OCEs can quickly execute actionable mitigation steps, reduce downtime, and improve the overall reliability of cloud systems.

2.3.3 Incident and Support Ticket Management

Incident Management

Incident management in cloud systems is a critical process that involves identifying, analyzing, and resolving incidents to restore normal service operations as quickly as possible and minimize the impact on business operations. In cloud environments, where infrastructure and applications are hosted on remote servers, incident management becomes even more crucial due to the complexity and scale of these systems. The incident management process typically includes the following steps:

- Detection: Incidents are detected and reported through monitoring tools, automated alerts, or user reports. Cloud systems often use sophisticated monitoring solutions to detect anomalies and trigger alerts.
- Triage: Once an incident is detected, it undergoes a triage process to assess its severity and impact on the business. During triage, incidents are classified and prioritized, with high-priority incidents that affect critical services or a large number of users being addressed first.
- Mitigation: Before fully resolving the incident, immediate actions are taken to mitigate its impact and prevent further damage. This could involve temporary fixes, isolating affected components, or redirecting traffic to maintain service availability.
- Resolution and Recovery: The team implements a solution to not only recover from the failure but also fix the underlying bugs causing the incident. This may involve rolling back changes, applying patches, fixing code defects, restarting services, or reallocating resources to ensure that the issue is fully resolved and does not recur.
- Postmortem Analysis: A postmortem analysis is conducted to analyze the incident and the response process. The goal is to identify lessons learned and implement changes to prevent similar incidents in the future.

The incident management lifecycle in cloud systems can accumulate a vast amount of historical data, *e.g.*, triage history and incident reports. This wealth of information provides a valuable foundation for developing intelligent and data-driven methods to enhance the detection, analysis, and resolution of incidents. By leveraging this data, we are allowed to design advanced algorithms

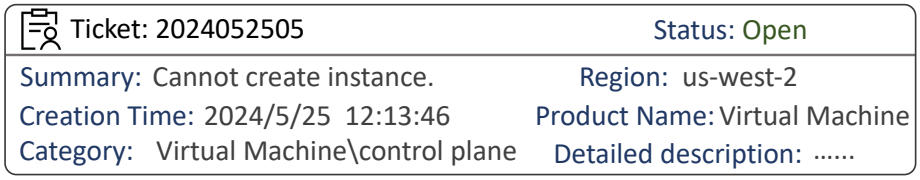


Figure 2.4: An Example of Support Tickets.

and machine learning models to automate and optimize various aspects of incident management, leading to more efficient operations and improved system reliability.

Support Ticket Management

Support ticket management is another perspective from the user side within IT and customer service operations, aimed at tracking, managing, and resolving customer or user issues efficiently. A support ticket is a documented record of a problem, request, or inquiry submitted by a user, typically through a helpdesk or support portal (*e.g.*, Azure Support [110]). Each ticket contains detailed information about the issue, including the user's contact information, a description of the problem, and any relevant attachments or logs. Similar to incident management, the support ticket management process generally includes the following steps:

- Ticket Creation: Users submit their issues via various channels such as email, web forms, phone calls, or chat. The support system automatically creates a ticket for each submission, assigning it a unique identifier.
- Categorization and Prioritization: Tickets are categorized based on the type of issue (e.g., technical support, billing, general inquiry) and prioritized according to their urgency and impact on the user or business operations.

- Assignment: Tickets are assigned to appropriate support agents or teams based on their expertise, availability, and the ticket's priority. This ensures that issues are handled by the most qualified personnel.
- Investigation and Resolution: Support agents investigate the issue by gathering additional information, diagnosing the problem, and implementing a solution. This may involve troubleshooting, consulting documentation, or escalating the ticket to higher-level support if necessary.
- Closure: Once the issue is resolved, the ticket is closed, and the user is informed. The resolution is documented within the ticket for future reference and knowledge base updates.
- Post-Resolution Review: Periodically, closed tickets are reviewed to identify trends, recurring issues, and areas for improvement in the support process. This helps in refining support strategies and enhancing overall service quality.

This process can also generate a wealth of data, including detailed descriptions of issues, resolutions, and user interactions. By leveraging Natural Language Processing (NLP) methods, we are allowed to analyze this data to identify common problems, automate ticket categorization, predict issue resolution times, and even provide automated responses for frequently asked questions. Utilizing NLP can significantly enhance the efficiency and effectiveness of the support ticket management pipeline, leading to quicker resolutions and improved user satisfaction.

□ **End of chapter.**

Chapter 3

Literature Review on Intelligent Cloud Reliability Management

In this chapter, we review existing studies on intelligent site reliability engineering in the literature. Fruitful studies have been conducted in this area. Our goal is to provide a comprehensive overview of these studies, highlighting their contributions and also discussing their limitations.

3.1 Metric-based Analysis

Metric-based Anomaly Detection

Anomaly detection on metrics has been a hot topic and is widely studied. Hundman et al. [60] leveraged LSTM without expert-labeled telemetry anomaly data to detect anomalies in multivariate metrics of spacecraft based on prediction errors. Malhotra et al. [107] proposed an LSTM-based encoder-decoder network to reconstruct the "normal" time series with high probabilities. Another way to model normal patterns is to learn the distribution of input data like deep generative models [43] and deep Bayesian network [143]. Donut [157] employed Variational AutoEncoder (VAE) to generate the normal hidden state of seasonal metrics without expert-labeled data. Donut successfully detects anoma-

lies in seasonal metrics with various patterns and data quality, but incurs high time complexity in the training phase. To reduce the training complexity, DAGMM [182] simplifies the hidden state as a combination of several Gaussian distributions. USAD [9] improves the autoencoder framework by incorporating adversarial samples to speed up the training phase. OmniAnomaly [136] employs a stochastic recurrent neural network to capture the normal patterns of multivariate time-series by simulating normal data distribution through stochastic latent variables. Similar to Hundman et al. [60]'s approach, OmniAnomaly provides interpretations based on the reconstruction probabilities of its constituent univariate metric. However, the generalization capability of these generative approaches with implicit modeling is degraded when they encounter severe noise in temporal metrics, which is very common in industrial production systems.

Metric-based Clustering

Clustering is crucial to handle the large volume of metrics generated from cloud systems. For example, Kane et al. [73] employ Principal Component Analysis (PCA) to transform multivariate metric data into univariate time series before clustering. Li et al. [91] proposed ROCKA, a robust and rapid metric clustering algorithm based on shape-based distance [125], designed to handle the challenges of large-scale anomaly detection, such as noise and phase shifts. ROCKA clusters metrics based on their underlying shapes, achieving high accuracy and reducing model training time with minimal performance loss. More recently, Zhang et al. [170] developed OmniCluster, a system instance clustering method that combines a one-dimensional convolutional autoencoder with a novel three-step feature selection strategy. This approach efficiently clusters system instances, significantly reducing the training overhead of anomaly detection models. However, these studies primarily focus on improving the accuracy of cluster-

ing rather than considering efficiency. They often require pairwise distance computations before performing clustering, which makes it challenging to scale to the cloud environment with tens of millions of metrics.

3.2 Log-based Analysis

Log Parsing

Log parsing has emerged as an active research topic in recent years [52, 67, 74, 181]. Existing log parsers can be divided into two main groups: syntax-based and semantic-based. Syntax-based log parsers are further categorized into three types: *Frequency-based parsers* [31, 117, 141, 142] utilize frequent patterns of token positions or n-gram information to distinguish templates and parameters in log messages. *Similarity-based parsers*: These parsers [50, 134, 139] compute similarities between log messages to cluster them and then extract the constant parts of log messages. *Heuristics-based parsers*: These parsers [34, 53, 68, 105, 109, 114, 146] use various heuristic algorithms or data structures to identify log templates based on designed characteristics. *Semantic-based parsers* achieve higher parsing accuracy by mining semantics from log messages, which is crucial for some downstream tasks [62, 90]. These methods typically require labeled log data for model training or tuning. For example, some semantic-based parsers [62, 90, 100] formulate log parsing as a token classification problem using bidirectional long short-term memory networks, while LogPPT [82] fine-tunes a pre-trained language model (*e.g.*, RoBERTa) for log parsing. However, recent benchmark studies [67, 74] have highlighted the inadequacy of these log parsers in handling large-scale, complex log data. To tackle this problem, recent studies have begun exploring LLMs for log analysis. Le et al. pioneered the investigation of LLM performance in log parsing, demonstrating their potential [81] . Xu et al. proposed LogDiv, a method lever-

aging the in-context learning (ICL) capability of LLMs for more accurate log parsing [158].

Log-based Anomaly Detection

Prior research related to log-based anomaly detection can be broadly categorized into two classes, *machine learning (ML)-based* methods and *deep learning (DL)-based* methods.

Machine learning-based methods include the use of principal component analysis (PCA) by Xu et al. [161] for mining system problems from console logs and the work of Lou et al. [101], who detects system anomalies by mining invariants among log messages. Lin et al. [93] proposed LogCluster, which recommends representative log sequences for problem identification by clustering similar log sequences. He et al. [54] proposed Log3C, which incorporates system monitoring metrics into the identification of high-impact issues in service systems. Loglizer [55] provides a comprehensive evaluation of using ML-based methods for log-based anomaly detection. **Deep learning-based methods** for log-based anomaly detection have been approached through various methods. One such method, proposed by Du et al. [35], is Deeplog, which utilizes a Long Short-Term Memory (LSTM) network to model log sequences. LogAnomaly [108] further utilizes log count vectors and log semantic vectors to model log sequences more comprehensively. However, both Deeplog and LogAnomaly are trained in an unsupervised manner, which has been shown to be less effective than supervised models [80]. Typical supervised solutions include a CNN-based approach [102] and GRU-based LogRobust [172]. Since labeled data is usually insufficient due to the labor-intensive nature of manual labeling, the semi-supervised method, PLElog [163], addresses this problem via label probabilistic estimation. DL-based methods utilize different neural network structures (*i.e.*, LSTM and Transformers) to capture patterns from historical log messages. However, these structures

are too complex in terms of time and space complexity to be deployed locally in an instance. Additionally, these methods still encounter accuracy degradation when logs change.

3.3 Incident Management

Researchers have devoted sustained efforts on empirical studies [25, 30, 48, 58, 96] of cloud incidents in the last few years. Gunawi et al. [48] discussed why outages still take place in cloud environments by analyzing headline news and public postmortem reports of 32 popular Internet services. Huang et al. [58] discussed their experiences with gray failure in production cloud-scale systems and demonstrated its broad scope and consequences. Chen et al. [25] presented a comprehensive study on how alerts and incidents are managed in large-scale public cloud systems. Cloud alerts are notoriously blamed for their great volume. In general, there are two threads of studies proposed towards resolving the challenge. The major thread aims to correlate alerts that are caused by the same incident [147] [47] [174]. Given a large number of alerts happening, Chen et al. [20] empirically found that only a small portion of alerts matters and proposed to prioritize alerts based on historical data. Chen et al. [23] [24] proposed to predict the link between two alerts by combining alert textual information and the topology information among alerts (i.e., the topology of components that generate these alerts). These studies either require experts' manual annotations [19] [23] or precise system topology [174] [28]. With large language models (LLMs) becoming increasingly popular, they have significantly enhanced the processing of incidents. The pioneering study by Ahmed et al. [2] explored fine-tuning the GPT-3 model to recommend mitigation steps for incidents. Additionally, using GPT-3.x for automatic understanding and summarization of incidents has shown promising results in cloud systems [69]. Microsoft has also introduced RCA-

Copilot [22], an LLM-based tool deployed for root cause analysis in real-world cloud systems, further demonstrating the potential of LLMs in improving incident management.

3.4 Support Ticket Management

Issue reports, including app reviews, user feedback, bug reports, test reports, GitHub issues, support tickets, etc., are crucial for service providers to gain a better understanding of their customers' experiences. A large body of research has been devoted to the analysis of issue reports, covering topics such as duplicate bug reports detection [145] [119] [178] [15], emerging issue detection [38] [177] [39] bug reproduction [176] [16], bug report summarization [129] [87] and empirical studies [76] [183] [104]. Most existing studies focus on natural language text information such as titles and descriptions. In addition, some latest attempts [51] [94] [29] proposed to jointly consider multi-modality features, e.g., text and images (e.g., app screenshots), which has become a recent hot trend in the research direction. However, these studies primarily focus on customer-side issue reports and do not fully address the complexities of ongoing alerts and incidents within cloud systems. In this thesis, we aim to bridge the gap between cloud alerts and user support tickets by integrating both sources of information. By correlating real-time alerts from cloud infrastructure with user-reported issues, our approach seeks to enhance the efficiency and accuracy of ticket processing. This holistic view enables a more comprehensive understanding of incidents, leading to quicker resolutions and improved service reliability.

□ **End of chapter.**

Chapter 4

Functional Cluster Identification

Ensuring the reliability of cloud systems is critical for both cloud vendors and customers. Cloud systems often rely on virtualization techniques to create instances of hardware resources, such as virtual machines. However, virtualization hinders the observability of cloud systems, making it challenging to diagnose platform-level issues. How to improve system observability and provide more insights for system maintenance is critical. In this chapter, we introduce providing more insights for observability by inferring *functional clusters* of instances with a non-intrusive solution called *Prism*. The remainder of this chapter is organized as follows. Section 4.1 provides the problem background and contributions we made. In Section 4.2, we conduct a pilot study to motivate our method design. Then Section 4.3 introduces the detailed design of the proposed solution Prism. Section 4.4 elaborates the evaluation results of Prism based on the real-world data collected from Huawei Cloud. Then Section 4.5 provides two industrial cases demonstrating the usage scenarios of Prism. Section 4.7 summarizes this chapter.

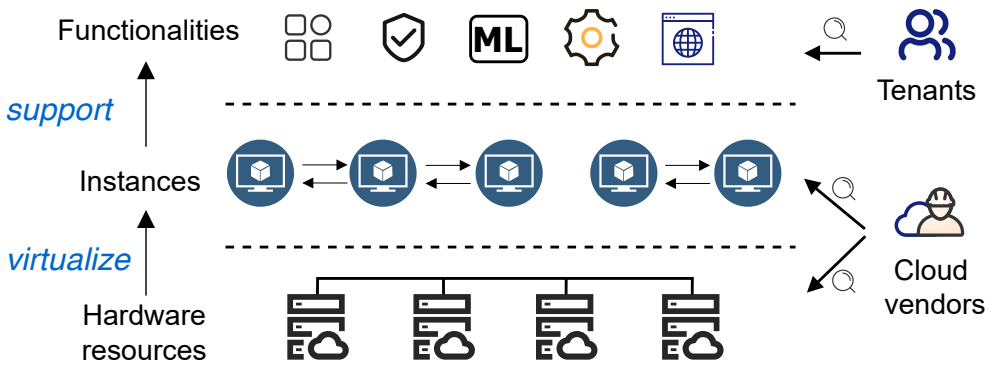


Figure 4.1: The Abstraction of Cloud Systems

4.1 Problem and Contributions

Cloud systems typically leverage virtualization techniques to abstract hardware resources (as shown in Figure 4.1), such as computation, storage, and networks, into instances (*e.g.*, virtual machines), serving as basic components of cloud services [64,106,156]. Such architecture provides flexibility and elasticity for tenants to subscribe to various instances to run services with different functionalities *e.g.*, machine learning and database services. This, in turn, enables them to create complex and customizable applications.

However, just as each coin has two sides, such practice makes it more challenging to ensure the reliability of cloud systems. In particular, virtualization degrades the observability of the system, *i.e.*, the ability to understand the system's internal execution state. Virtualization introduces an additional layer of abstraction between the underlying hardware and the running applications, making it difficult to correlate the problems across different layers [147]. For example, an issue at the application layer may be caused by problems either within the instance itself or with the underlying hardware.

To enhance system observability, a common practice for cloud vendors is deploying a variety of monitors to collect runtime in-

formation of each instance [20, 25, 42, 98], which record only data related to reliability issues without touching users' privacy. The monitoring data are then utilized for downstream maintenance tasks. For example, *communication traces*, which record network packet transmissions between instances (*e.g.*, the source and destination IP addresses and port numbers), are often used to identify abnormal network behaviors, such as network attacks and excessive traffics [8, 63, 70, 118, 159]. On the other hand, *performance metrics*, such as CPU utilization and memory usage, are commonly utilized for detecting anomalies and localizing faults [27, 164, 173].

The monitoring data have provided valuable insights to ensure the reliability of individual instances. However, cloud vendors still view instances as distributed black boxes without knowing how an application is deployed across the infrastructure [124]. Consequently, it can be challenging to assess the impact of issues at the platform level (such as instance or hardware problems) on applications that are deployed on top of them. For example, packet losses in individual instances are commonplace in cloud systems and are generally ignored, as they seldom impact customer applications. However, when multiple instances, all supporting the same application, concurrently experience packet losses, it likely indicates a more significant issue that users may encounter, such as interruptions due to network disconnections. The limited awareness of relationships between instances complicates the detection of such problems, thereby impeding timely mitigation efforts.

To bridge this gap and improve the system observability, we propose to infer *functional clusters* of instances, where each cluster contains the instances having similar functionalities. With this additional knowledge, cloud vendors can enhance the reliability of the cloud by improving various downstream management and maintenance tasks (to be detailed in section 4.5). However, there are two major challenges that we need to overcome to achieve this

goal. The first challenge is that only limited information is available. As mentioned before, cloud vendors cannot access tenants' private data, including logs and source codes. A non-intrusive solution that relies solely on external data (*e.g.*, traces and metrics) is required. The second challenge is the large scale of instances in cloud systems. A typical cloud system can consist of millions of instances in total [124], resulting in an enormous amount of data for analysis. Valuable insights are concealed within the vast and noisy data of cloud systems, making it difficult to reveal the hidden function clusters.

To tackle the first challenge and explore a feasible non-intrusive solution, we first conduct a pilot study on the services deployed in Huawei Cloud. For privacy reasons, we only use internal services of Huawei Cloud without touching tenants' instances. Specifically, we utilize a total of 3,062 internal instances covering services with 397 types of functionalities and study whether different functionalities can be identified simply based on external monitoring data. Our study uncovers that instances having similar functionalities share similar *communication* and *resource usage* patterns. Communication patterns mean that instances with similar purposes may frequently communicate with the same set of destinations, reflected in their communication traces. We find that 75% of instances within the same functional clusters have a high overlap (≥ 0.7) in their communicated destinations. Conversely, for 92% of instances with different functionalities, the overlap is only less than 0.2. Additionally, despite the large scale of instances, 99.1% of instances communicated with a limited number of destinations (fewer than 50), indicating a strong locality in communication patterns. Resource usage patterns, on the other hand, denote that instances with similar functionalities would demonstrate comparable resource consumption, which is reflected in their metrics. For example, a machine learning service is expected to exhibit greater CPU usage, while instances running an in-memory database like

Redis would primarily require more memory. We find that most ($\sim$75%) of instance pairs with the same functionalities have high metric-based similarities ($\geq$ 0.8), while the similarities decrease for those instances having different functionalities.

Motivated by the two kinds of inherent patterns of the instances, we formulate the identification of functional clusters as a clustering problem. Intuitively, we aim to cluster the instances by harmoniously integrating the communication patterns and resource usage patterns. To achieve this goal and alleviate noises within the tremendous data, we propose *Prism*, which adopts a coarse-to-fine clustering strategy. Prism consists of two components, *i.e.*, *trace-based partitioning* and *metric-based clustering*. In the trace-based partitioning step, we leverage the communication patterns to coarsely divide the entire large set of instances into smaller chunks. This step helps limit the comparison space within each chunk, thus reducing the complexity of the subsequent clustering process and eliminating noises introduced by instances from other clusters. In the metric-based clustering step, we perform fine-grained clustering by comparing the resource usage patterns of instances in a pairwise manner. This step allows us to carefully group instances within the same functional cluster.

To evaluate Prism, we conduct extensive experiments on two datasets collected from the production environment of Huawei Cloud, a top-tier cloud provider serving global customers. To evaluate the generality of Prism, these datasets were procured from two regions of Huawei Cloud, each covering a diverse set of functionalities. The experimental results show that Prism achieves a v-measure of $\sim$0.95, surpassing existing state-of-the-art solutions, and is robust to parameter changes. Moreover, Prism is both scalable and efficient, with a linear time complexity, enabling it to handle a substantial number of instances. Furthermore, we have deployed Prism in Huawei Cloud, and we share two real-world use cases to demonstrate the usefulness of functional clusters in

maintaining Huawei Cloud. In the first case, functional clusters showcase the ability to detect vulnerable application deployments that may be at risk of disruption due to hardware failures. The second case shows how functional clusters can aggregate minor packet loss errors across instances, thus enabling the identification of latent issues that are not observable at either the instance or region level. We summarize our contributions as follows:

- We conduct a pilot study to understand the characteristics of functional clusters across over 3,000 instances based on a real-world cloud system, Huawei Cloud (Section 4.2). Our findings reveal two clues for identifying functional clusters (*i.e.*, communication patterns and resource usage patterns).

- We design a non-intrusive solution called Prism to identify functional clusters in large-scale cloud systems, which is able to effectively capture and integrate the inherent communication and resource usage patterns among instances (Section 4.3).

- Extensive experiments are conducted on two real-world industrial datasets (Section 4.4). Our results demonstrate that Prism is effective, efficient and practically useful in identifying functional clusters in industrial cloud systems. Our dataset and code are made public to benefit the community[1].

4.2 Pilot Study

In this section, we present a pilot study to understand the characteristics of instances that can facilitate the identification of functional clusters.

Specifically, we conduct the pilot study across over *three thousand* internal instances in Huawei Cloud, aiming to find clues to uncover the valuable functional clusters. We conduct manual inspections in collaboration with the corresponding teams within

[1]https://github.com/OpsPAI/Prism

Huawei Cloud to understand their functionalities. We obtain services covering 397 types of functionalities in total, and more details about this dataset are in §4.4.1.

Communication Pattern

The communication pattern serves as an indicator that instances within the same functional cluster tend to exhibit comparable network behaviors, as evidenced by the communication traces they generate. As inspired by [124], instances within the same functional clusters might communicate with similar destinations. To investigate this, we combine every two instances and compute the overlap of their destinations through Jaccard similarity [152]. Then, we compare the similarities within the same clusters and across different clusters.

Fig. 4.2-(a) presents the comparison results of communication pattern similarities within or across clusters, where we can observe a significant difference between them. When examining instances within the same cluster, we find that 50% of the instance pairs demonstrate more than 0.8 similarity and over 75% of them exhibit more than 0.6 similarity. In contrast, when comparing instance pairs from different clusters, over 75% of the pairs exhibit a similarity score of 0, indicating no overlap between their destinations. Additionally, 96% of the pairs have a similarity score of $<$0.4, indicating that instances from different clusters rarely communicate with the same destinations. However, for some cross-cluster instances, there is still a little overlap in their destinations. These destinations are usually common services such as network gateway and authentication that are shared by multiple applications.

To further understand the communication patterns, we study how many different destinations one instance can frequently communicate with. Fig. 4.2-(b) shows the results. We can find that even though there are thousands of instances in total, the ma-

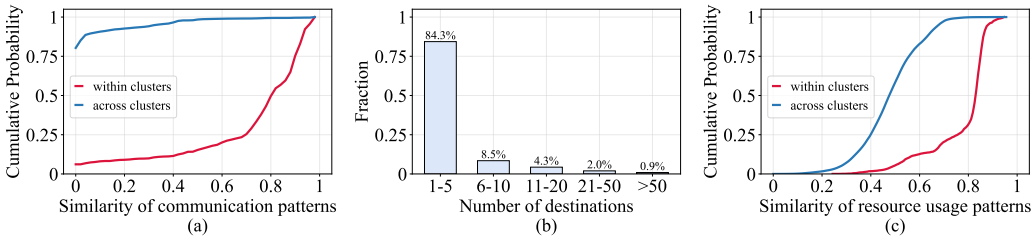


Figure 4.2: Results of the Study on Communication and Resource Usage Patterns.

jority of the instances only communicate with a small number of destinations. For example, 84.3% of instances communicate with 1 to 5 instances, and 99.1% of instances communicate with less than 50 instances. This suggests a strong *locality* of instances, *i.e.*, most instances tend to communicate with a small set of other instances frequently.

Resource Usage Pattern

Intuitively, instances within the same functional cluster should observe similar patterns in their resource consumption (*i.e.*, resource usage patterns). To investigate whether resource usage patterns can be utilized to uncover functional clusters, we analyze the similarities in the metric data among instances, either within the same functional cluster or across different clusters. Thus, we compare the multivariate metric similarity on two instances using the multivariate dynamic time warping (DTW) distances [116], a distance metric to compare a pair of time series that may vary in timing.

Fig. 4.2-(c) shows the distribution of resource usage pattern similarities among instances, either within or across functional clusters. We can observe that the similarities of instance pairs within clusters are generally large, with over 75% of such pairs exhibiting 0.7 similarity or higher. In contrast, instance pairs across clusters display smaller similarities, with 92% of pairs across dif-

ferent clusters possessing less than 0.2 similarity. However, it is worth noting that there is a small portion ($\leq$10%) of cross-cluster instance pairs that have high metric-based similarities, with a value of $\geq$0.8. This is reasonable since instances having different functionalities could behave similarly, *e.g.*, have a high CPU utilization. Nevertheless, it still suggests that leveraging the similarities between instance metrics is promising in distinguishing their functional clusters.

Summary. We summarize our findings as follows.

- Instances that belong to the same functional cluster exhibit comparable communication patterns, as evidenced by the considerable overlap in their communication destinations. Furthermore, the analysis reveals that the majority of instances interacted with a limited number of other instances, indicating a strong locality of instances.
- Instances within clusters generally exhibit high similarities in their resource usage patterns, while instance pairs across clusters show smaller similarities.
- While communication and resource usage patterns provide valuable insights, they are not entirely reliable indicators for distinguishing between different functional clusters, as some *noises* in the form of cross-cluster instances with high similarities in both patterns are observed.

4.3 Methodology

4.3.1 Overview

The goal of this chapter is to design a non-intrusive solution to discover functional clusters among massive instances in a large-scale cloud system. The input is an entire set of instances and their associated monitoring data, *i.e.*, communication traces and

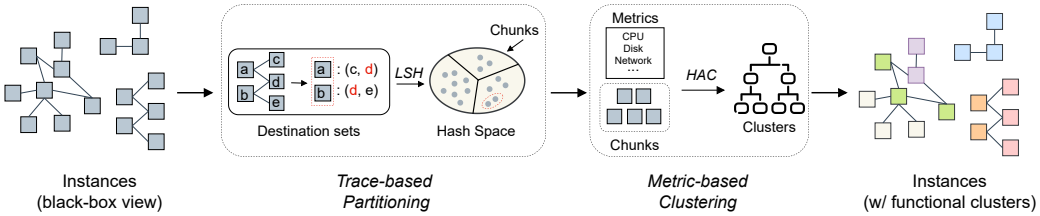


Figure 4.3: The Overall Workflow of Prism

performance metrics. The output of our approach is multiple clusters, where each cluster represents a functional cluster consisting of instances that have similar functionalities.

To achieve this goal, we propose Prism, an automated approach that can effectively discover functional clusters based on both the communication patterns and resource usage of instances. Fig. 4.3 illustrates the overall workflow of Prism, which comprises two main components: *trace-based partitioning* and *metric-based clustering*. Given a set of instances, Prism adopts a two-stage clustering process, which progressively divides the entire set of instances to coarse-grained *chunks*, then fine-grained *functional clusters*. Specifically, the *trace-based partitioning* step is inspired by the strong locality of communication patterns, as shown in section 4.2. Based on communication patterns, Prism first separates all instances into different chunks. Instances in the same chunk share similar communication destinations. By dividing the complete instances set into multiple small chunks, we can reduce the noises introduced from other instances during the subsequent fine-grained clustering step. For each chunk, *metric-based clustering* is then applied to generate fine-grained clusters by measuring the similarities of monitoring metrics of instances. Finally, instances belonging to the same resultant cluster are considered to have similar functionalities. Such a coarse-to-fine design avoids pairwise comparisons between a large number of instances and reduces noises between instances, making Prism salable and practical for large-scale cloud systems.

It is important to note that Prism relies solely on external monitoring data and does not access any of the tenants' private data, which ensures that there are no privacy concerns. While we can infer which instances have similar functionalities, we cannot identify the specific type of the functionalities in use. This approach maintains our tenants' confidentiality.

4.3.2 Trace-based partitioning

As studied in section 4.2, instances sharing the same functional clusters are more likely to communicate with a similar set of destination hosts. Thus, the trace-based partitioning of Prism measures the communication pattern similarity and divides instances into coarse-grained *chunks*.

Data Preprocessing. Let x_i represent an instance in the cloud system. Communication traces can be represented as tuples of the form (x_{src}, x_{dst}), where x_{src} and x_{dst} represent the instances that communicate with each other. By analyzing the communication traces, we can obtain the *destination set* of each instance, denoted by $S_i = (x_1, x_2, x_3, ...)$, which contains all the instances that have communicated with x_i. However, as demonstrated in section 4.2, instances with dissimilar functionalities may share common destinations, such as network gateways, which can introduce noise when comparing the communication patterns between instances. To mitigate this issue, we remove instances that interact with more than 100 different instances, which is rare as shown in Fig. 4.2-(b).

Jaccard Similarity-based Partitioning. Next, we divide all instances into chunks by measuring how much their destination sets overlap. To achieve this, a straightforward solution is to calculate the Jaccard similarity [140] of destination sets of every pair of instances, which is denoted as $J(x_i, x_j) = \frac{|S_i \cap S_j|}{|S_i \cup S_j|}$, *i.e.*, the ratio of the size of their intersection to the size of their union. However, it requires conducting pairwise comparisons between millions

of instances in a large-scale cloud system. This process can be extremely time-consuming and may render the approach unfeasible in practice.

To address this issue, we propose to leverage locality-sensitive hashing (LSH) [85] to enable efficient partitioning. LSH is a technique developed for identifying similar items in large datasets. Its idea involves hashing the items into signatures such that similar items are more likely to be assigned to the same bucket. Given a query, LSH can efficiently return similar items with a sub-linear time cost without pairwise comparison with the entire instance set. In our context, we combine LSH with the MinHash function, which allows items with high Jaccard similarities put into the same buckets [12].

Algorithm 1 describes the trace-based partitioning process. First, we extract the destination sets S of each instance from historical communication traces (lines 1-5). Second, for each instance x_i, we apply MinHash function to its destination set S_i to obtain the hash signature. The hash signature is then inserted into the LSH model (lines 7-10), which assigns the item to a bucket. Third, for each instance x_i, we search its nearest neighbors $\mathcal{N}_i$ within the buckets produced by the LSH model (lines 12-14). Here, a manual-defined threshold $\theta_{LSH} \in [0, 1]$ is included, where a smaller θ_{LSH} value allows more dissimilar neighbors to be included. After that, we group the instance x_i with its neighbors $\mathcal{N}_i$ based on the Disjoint-set data structure U (lines 15-19). This data structure U provides two efficient operations, *i.e.*, $U.findSet$ that find the set that contains a specific item and $U.unionSet$ that merge two disjoint sets. If we find the sets containing x_i and containing x_j are disjoint (line 16), we merge these two sets (line 17) since x_i and x_j are similar. In this way, we progressively divide the entire set of instances into multiple disjoint sets (*i.e.*, chunks) managed by U. Finally, we can obtain all the instance chunks $\mathcal{C}$ by enumerating the records in U (line 21).

Algorithm 1: Trace-based Partitioning

Input: List of instances: $\mathcal{X} = \{x_1, x_2, ..., x_l\}$; Communication trace records: $\mathcal{R} = \{r_1, r_2, ..., r_t\}$; Similarity threshold: θ_{LSH}
Output: Multiple instance chunks: $\mathcal{C} = \{C_1, C_2, ...\}$
Init: $S \leftarrow$ Empty list of feature sets; $M_{LSH} \leftarrow$ empty LSH model; $U \leftarrow$ Disjoint-set data structure

```
1  // (1) Construct feature sets
2  for i ← 1 to t do
3  |   x_src, x_dst ← r_i
4  |   S[x_src].insert(x_dst)
5  end
6  // (2) Build the LSH model
7  for each instance x_i ∈ X do
8  |   S_i ← S[x_i]
9  |   M_LSH.insert(MinHash(S_i))
10 end
11 // (3) Search neighbors and build chunks
12 for each instance x_i ∈ X do
13 |   S_i ← S[x_i]
14 |   N_i = M_LSH.search(S_i, θ_LSH) // find neighbors
15 |   for each instance x_j ∈ N_i do
16 |   |   if U.findSet(x_i) != U.findSet(x_j) then
17 |   |   |   U.unionSet(x_i, x_j) // merge x_i and neighbors
18 |   |   end
19 |   end
20 end
21 C ← U.getAllSets()
```

The trace-based partitioning algorithm is highly efficient for two reasons. First, we bypass the expensive pairwise similarity computation for all the instances by using LSH with MinHash. Secondly, we leverage the disjoint-set data structure to merge similar instances into chunks efficiently. The *findSet* and *unionSet* operations of the disjoint-set data structure can be completed within nearly constant time complexity, which further ensures the efficiency of the merging process. Moreover, the number of neighbors $\mathcal{N}_i$ (line 14) is generally fewer than 50, which is much smaller than the total instance number $\mathcal{X}$ (line 12) due to the locality of communication patterns (Fig. 4.2-(b)), which improves Prism's

scalability, making it feasible for large-scale cloud systems like Huawei Cloud.

4.3.3 Metric-based Clustering

Trace-based partitioning tends to group as many instances as possible together, which can inevitably include instances with different functionalities to the same chunk. The reason is that instances from different clusters can still communicate to the same destinations (as studied in section 4.2), and this leads to overlap of the destination sets of these instances, which may be wrongly grouped together.

To address this problem, we further group these instances by utilizing more fine-grained monitoring metrics that record detailed runtime information of instances (*i.e.*, resource usage patterns as studied in section 4.2). Each instance is monitored via multiple dimensions to ensure its reliability, producing multivariate metrics, including CPU utilization rate, network incoming/outgoing bytes rate, disk read/write request rate, and disk read/write bytes rate. In the following, we aim to calculate a metric-based distance for each pair of instances. Then, we can cluster those instances that are close to each other.

Data Preprocessing. We apply the following preprocessing techniques to the raw metric data collected to remove noises and normalize the data within a comparable scale. First, we regard apparent extreme values as anomalous noises within the metric data because these values can bias the subsequent distance computation step. For each metric, we replace the data points that are out of the three-sigma range with the average value of the nearest ten points. Next, since the amplitude scales of different metrics are different, *e.g.*, network-related metrics are highly variable and may range from tens of bytes to millions of bytes. This can make the produced distances incomparable between instances

with different network traffic volumes. To address this issue, we apply natural logarithm to these metrics following [124] to make it more robust to its variance. The logarithm only solves the issue of highly variable amplitudes but does not ensure that the data points fall within the same range. Therefore, finally, we apply min-max normalization to scale each of the metrics to the range of 0-1, allowing comparison across different metrics. Formally, using y to denote a metric time series, the normalized values can be calculated as $y' = \frac{y-min(y)}{max(y)-min(y)}$.

Metric-based Distance Calculation. For an instance x, its preprocessed monitoring metrics form a group of multivariate time series represented as a matrix $\mathbf{M}_i \in \mathbb{R}^{n \times k}$, where n is the number of timestamps and k is the number of metrics used. We measure the metric-based similarity of two instances using a distance that simultaneously considers all the multivariate metrics of them. To achieve this, we first compare each metric, then aggregate the distances to produce an overall distance.

Specifically, we adopt dynamic time warping (DTW) distances for distance measurement [116]. The reason we use DTW is to overcome the problem that the monitoring metrics of different instances can have time shifts, namely, these time series may not be aligned in terms of the collection timestamps, making traditional distance measures such as Euclidean distance ineffective. In contrast, DTW allows for flexible matching of similar patterns in the time series, even when they occur at different timestamps. Based on the DTW calculation, the overall distance $d(x_i, x_j)$ between

two instances x_i and x_j can be formulated as follows:

$$d(x_i, x_j) = \sum_{u=1}^{k} \omega(i,j)_u \times DTW\big(\mathbf{M}_i(:,u), \mathbf{M}_j(:,u)\big), \tag{4.1}$$

$$\omega(i,j)_u = \frac{\omega(i,j)'_u}{\sum_{v=1}^{k} \omega(i,j)'_v}, \tag{4.2}$$

$$\omega(i,j)'_u = \frac{1}{2}\big(\sigma(\mathbf{M}_i(:,u)) + \sigma(\mathbf{M}_j(:,u))\big), \tag{4.3}$$

where u denotes the metric in concern, $\mathbf{M}_{i/j}(: .u)$ is the u_{th} column of the corresponding metric matrix. In particular, we use $\omega(i,j)_u$ as a weight associated with the u_{th} metric to measure the importance of each metric. Each weight is calculated as the average of the standard deviation (*i.e.*, $\sigma(\cdot)$) of the two metrics of corresponding instances as shown in Equation 4.3, which is normalized to the range of 0 to 1 across different metrics using Equation 4.2. In doing this, we reduce the weight of the metrics that barely fluctuate (*e.g.*, two instances keep the CPU utilization rate around 80%), since these metrics are less informative in representing the characteristics of instances. In contrast, if two metrics are simultaneously changing following the same trend, they are more likely to indicate instances performing the same functionalities.

Clustering Algorithm. We then apply a clustering algorithm in each *chunk* based on the metric-based distances to produce more fine-grained clusters (*i.e.*, functional clusters). Specifically, we choose the hierarchical agglomerative clustering (HAC) [120] algorithm because it allows us to adjust the number of produced clusters via setting a distance threshold, *i.e.*, θ_{HAC}. The clustering algorithm starts by considering each instance as a single cluster and then iteratively merges the closest pairs of clusters until a user-defined threshold θ_{HAC} is reached. In this process, we use complete linkage [32] to find the closest pair of clusters, *i.e.*, the distance between two clusters is defined as the maximum DTW distance between any pair of instances in the two clusters.

While HAC requires the computation of distances between instances in a pairwise manner, it is still efficient since HAC is applied separately in each chunk. Recall that chunks are produced by the trace-based partitioning step, and each chunk only contains tens of instances because of the locality of communication patterns (as shown in section 4.2). Therefore, the computation within each small chunk can significantly reduce the computation cost, making our framework scalable to a large number of instances in cloud systems.

4.4 Evaluation

We evaluate Prism by answering the following research questions (RQs):

- **RQ1:** How effective is Prism in clustering instances having similar functionalities?
- **RQ2:** How does each component contribute to the overall performance of Prism?
- **RQ3:** What is the parameter sensitivity of Prism?
- **RQ4:** What is the efficiency of Prism?

4.4.1 Experimental Setting

Dataset. We evaluate Prism using two datasets collected from the production environment of Huawei Cloud. To evaluate the generalizability of Prism, the two datasets ($\mathcal{A}$ and $\mathcal{B}$) are collected from two different geographically isolated regions with different numbers of users. The detailed statistics of the two datasets are listed in Table 4.1. These datasets only include instances that are subscribed by internal customers, where we are able to manually inspect their functionalities by collaborating with corresponding

Table 4.1: Dataset Statistics

Datasets	# Functionalities	# Instances	# Traces	# Metrics
Dataset $\mathcal{A}$	292	2,035	100.2 M	7.25 M
Dataset $\mathcal{B}$	105	1,027	121.6 M	3.71 M
Total	397	3,062	212.6 M	10.96 M

teams. We select the instances running on our production environment that are most frequently invoked according to their communication traces. Then, we reach the owners of these instances to figure out the concrete functionalities these instances support, and we finally obtain 3,062 labeled instances. Although we are unable to fully cover all instances within the Huawei Cloud due to the manual effort required, our datasets encompass a diverse range of functionalities (397 types in total), such as databases, disaggregated memory, authentication servers, search engines, and machine learning algorithms. Such diversity would help evaluate whether a clustering algorithm can generalize to different functionalities. Additionally, these functionalities can belong to different applications. For example, while various applications may each have their own databases, these database functionalities are distinguished from one another in our datasets since they are utilized by distinct applications that serve diverse workloads (e.g., databases of an online shopping application and a face recognition application). For the monitoring data, traces are extracted from the network packet transmission records, while metrics are collected at five-minute intervals. Given the extensive usage and frequent communication of instances, we ultimately collect hundreds of millions of traces. In terms of metrics, the total number of points is 10.96 million for all instances. We have made our datasets publicly available in our GitHub repository. However, due to confidentiality concerns, the actual functionality names have been anonymized and are represented as "cluster_ID".

Evaluation Metrics. We use the metrics *homogeneity*, *completeness* and *V-measure* to evaluate the effectiveness of Prism in grouping the instances within the same functional cluster. These metrics have been widely adopted in evaluating the quality of clustering results in previous studies. Homogeneity measures the proportion of instances in the same cluster that share the same ground truth labels. Completeness, on the other hand, measures the proportion of instances with the same ground truth labels that are grouped into a single predicted cluster. V-measure is a harmonic mean of homogeneity and completeness, providing an overall indicator for clustering performance considering the trade-off between these two metrics.

Competitors. We select the competitors from recent studies:

- *OSImage* is a basic baseline that uses the name of the operating system (OS) image to differentiate between instances. Cloud providers offer various pre-installed OS images to cater to diverse customer needs. For example, an OS image named *deeplearning-pytorch-2.0* implies that the instance is designed for executing deep learning applications.

- *CloudCluster* [124] clusters instances based on their pairwise traffic matrix in cloud projects to determine the functional structure of the cloud service. It normalizes each row of the traffic matrix by feature scaling, then reduces its dimensionality through low-rank approximation. Finally, HCA is employed to group all instances.

- *ROCKA* [91] aims to cluster instances by using their monitoring metrics. ROCKA first normalizes the metrics to eliminate amplitude differences. It then uses shape-based distance (SBD) as a distance measure, which is robust to phase shift and efficient for high-dimensional time series data. Then, clusters are created based on DBSCAN algorithm.

- *OmniCluster* [170] clusters instances based on multivariate met-

Table 4.2: Effectiveness of Functional Cluster Discovery

Methods	Dataset $\mathcal{A}$			Dataset $\mathcal{B}$		
	Homo.	Comp.	V Meas.	Homo.	Comp.	V Meas.
OSImage	0.238	0.894	0.376	0.258	0.889	0.400
CloudCluster	0.346	0.748	0.473	0.369	0.753	0.495
ROCKA	0.831	0.882	0.856	0.875	0.900	0.887
OmniCluster	0.932	0.862	<u>0.896</u>	0.944	0.877	<u>0.909</u>
Prism	0.976	0.916	**0.945**	0.979	0.922	**0.950**

rics of each instance. It employs a one-dimensional convolutional autoencoder (1D-CAE) to extract the low-dimensional features of all metrics. These features are selected based on their periodicity and redundancy. Finally, it uses HAC to divide all instances into different clusters.

4.4.2 Experimental Results

Effectiveness in functional cluster Discovery (RQ1)

In this RQ, we evaluate the accuracy of the functional clusters discovered by Prism in comparison with state-of-the-art baseline methods. To achieve this, we apply Prism and baseline methods to cluster instances in the dataset of $\mathcal{A}$ and $\mathcal{B}$. We present the results of our experiments in terms of homogeneity (Homo.), completeness (Comp.), and v-measure (V Meas.) in Table 4.2, where we highlight the best V Meas. with boldface and the second-best ones with underline.

It can be observed that Prism outperforms three state-of-the-art baseline methods, namely CloudCluster, ROCKA, and OmniCluster, by a significant margin, achieving V-measures of 0.945 and 0.950 on datasets $\mathcal{A}$ and $\mathcal{B}$, respectively. These results indicate that Prism can achieve the best balance between homogeneity and completeness. This can be attributed to the fact that Prism effectively integrates communication and resource usage patterns to discover functional clusters. Unlike Prism, baseline methods

Table 4.3: Contribution of Different Components in Prism

Methods	Dataset $\mathcal{A}$			Dataset $\mathcal{B}$		
	Homo.	Comp.	V Meas.	Homo.	Comp.	V Meas.
Prism	0.976	0.916	**0.945**	0.979	0.922	**0.950**
Prism w/o Metrics	0.462	**0.920**	0.615	0.463	**0.949**	0.622
Prism w/o Traces	0.949	0.869	<u>0.907</u>	0.915	0.893	<u>0.904</u>

typically focus on either trace or metric data, leading to worser performance. Specifically, OSImage exhibits low homogeneity but high completeness, as using only image names to separate instances can overly group instances with different functionalities that share the same images. While CloudCluster outperforms OSImage in v-measure, it falls short of other metric-using baseline methods, suggesting that metric similarities are more effective in distinguishing functionalities than communication trace similarities.

Answer to RQ1: Prism outperforms all state-of-the-art comparative methods in revealing the functional clusters across two different datasets, achieving a v-measure of 0.945 and 0.950 in dataset $\mathcal{A}$ and $\mathcal{B}$.

Contribution of Each Component (RQ2)

In this RQ, we evaluate each component's contribution to Prism's overall performance. We created two Prism variants and compared them with the original approach across datasets $\mathcal{A}$ and $\mathcal{B}$. The first, *Prism w/o metrics*, eliminates metric-based clustering, relying solely on communication destination similarity. The second, *Prism w/o traces*, omits trace-based partitioning, directly applying the HAC algorithm to cluster instances based on resource usage patterns.

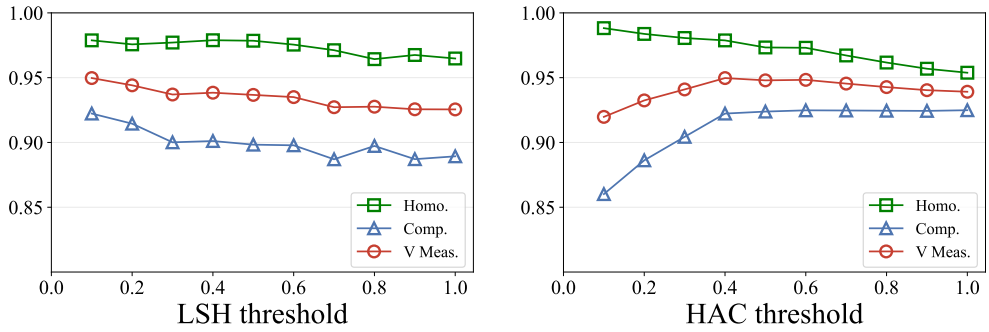
We present the comparison results in Table 4.3, from which we make the following observations. (1) Removing either of the

two components can adversely affect the performance of Prism, underscoring the necessity of integrating both communication and resource usage patterns. (2) The V-measure of *Prism w/o metrics* is significantly lower than that of *Prism* and *Prism w/o traces*, primarily due to its low homogeneity. This suggests that the trace-based partitioning step over-clusters many instances that should be separated. The communication pattern alone is not distinctive enough because instances having different functionalities should still communicate with some common instances, such as network gateway and proxy services (as illustrated in Fig. 4.2-(a)). Nonetheless, the use of solely communication patterns achieves the best completeness score, implying that it barely separates clusters that should be grouped. (3) *Prism w/o traces* has the lowest completeness score, indicating that it can overly split clusters apart, but it has a considerably high homogeneity. This observation implies that Prism harnesses the benefits of both performance metrics and communication traces, achieving the optimal balance between homogeneity and completeness.

Answer to RQ2: The variants, *Prism w/o metrics* and *Prism w/o traces*, each sacrifice either homogeneity or completeness. Yet, Prism effectively combines communication traces and metric data, yielding the highest v-measure, *i.e.*, a balanced performance in completeness and homogeneity.

Parameter Sensitivity (RQ3)

In the design of Prism, we identify the following two parameters that are manually selected and potentially affect the performance of Prism. For clarity, we present the evaluation results in Dataset $\mathcal{B}$; similar results are obtained in dataset $\mathcal{A}$.



(a) Impact of θ_{LSH}

(b) Impact of θ_{HAC}

Figure 4.4: Parameter Sensitivity of Prism

LSH threshold (θ_{LSH})

In Section 4.3.2, we utilize LSH algorithm to perform a search for similar neighbors during the trace-based partitioning step. The LSH algorithm groups similar items together into the same bucket with high probability, but it cannot guarantee that all items in the same bucket are actually similar; therefore, θ_{LSH} is utilized to filter dissimilar items within each bucket.

We varied the value of θ_{LSH} from 0 to 1 with a step size of 0.1 and evaluated the performance of Prism. The results, shown in Fig. 4.4(a), indicate that the V-measure remains stable with only a slight decrease as θ_{LSH} increases, which is primarily due to the decrease in completeness. This is because the LSH algorithm has already grouped similar items together into different buckets. Furthermore, since the communication patterns of most instances are distinct from one another, as depicted in Fig. 4.2-(a), there are only a small number of dissimilar items in the same bucket. As a result, adjusting θ_{LSH} does not significantly affect the clustering results.

HAC threshold (θ_{HAC})

In Section 4.3.3, HAC is used for clustering instances within each chunk, where the parameter θ_{HAC} controls the granularity of clustering: a smaller value of results in more fine-grained clusters, while a larger value results in fewer, coarser clusters.

We enumerated the value of θ_{HAC} from 0 to 1 with a step size of 0.1 and evaluated the performance of Prism. The results are shown in Fig. 4.4(b). We observed that increasing θ_{HAC} can increase completeness and decrease the homogeneity. This is because larger clusters are generated when θ_{HAC} is larger. The best v-measure is achieved when θ_{HAC} is around 0.4. Subsequently, there is a slight decrease in homogeneity, while the v-measure remained stable. This decline in homogeneity is due to the inclusion of more dissimilar instances in a cluster, thereby reducing its homogeneity. Nevertheless, the preceding trace-based partitioning step groups similar instances together, resulting in a limited number of dissimilar instances. Hence, the overall performance is not significantly affected.

Answer to RQ3: Prism is not significantly sensitive to the parameters θ_{LSH} and θ_{HAC}. This is because the trace-based partitioning step already groups similar instances together and separates dissimilar instances based on their communication patterns. Thus, adjusting these two parameters only has a minor effect on the clustering results.

Efficiency of Prism (RQ4)

In this section, we assess the efficiency of Prism in the context of large-scale cloud systems with millions of instances that are frequently created, deleted or updated. To this end, we apply them to 1,000 / 5,000 / 10,000 / 50,000 / 100,000 instances and record the time needed (in seconds) to complete the clustering process.

Table 4.4: Efficiency Comparison with Increasing Scales of Instances

Methods	# Instances				
	1,000	5,000	10,000	50,000	100,000
CloudCluster	0.9	23.87	78.65	1768.7	5585.7
ROCKA	80.7	1981.8	7850.3	-	-
OmniCluster	31.7	264.6	1048.6	26531.8	-
Prism w/o Metrics	3.9	19.1	40.2	195.1	392.4
Prism w/o Traces	80.3	2066.1	8232.3	-	-
Prism	18.2	89.4	183.9	929.2	1912.7

Table 4.4 presents the results, from which we can make the following observations: (1) ROCKA, OmniCluster, and Prism w/o Traces require increasingly more time as the number of instances increases, and they cannot complete the clustering process within a reasonable time when clustering 100,000 instances. This is mainly because these methods require pair-wise similarity computation based on instance metrics, resulting in a quadratic growth in time complexity as the number of instances increases. OmniCluster mitigates this issue by reducing the dimensionality of metrics, requiring less time than the other two methods. (2) CloudCluster and Prism w/o Metrics are more efficient than other baseline methods. Prism w/o Metrics is more efficient because we optimize efficiency using pair-wise comparison with LSH and MinHash, as described in Section 4.3.2. (3) Prism is less efficient than Prism w/o Metrics since it requires an additional metric-based clustering step. In addition, when the number of instances is fewer than 10,000, CloudCluster outperforms Prism because the time required by Prism to build the LSH index is dominant. However, as the number of instances increases to 100,000, Prism's efficiency becomes superior to other baselines, being four times faster than CloudCluster. This is attributed to the coarse-to-fine clustering process, which limits pairwise distance computation within small chunks. Therefore, the time cost of Prism only increases linearly

with the instance numbers.

Answer to RQ4: Compared with state-of-the-art solutions, Prism is the most efficient solution when processing a large number of instances (*e.g.*, 100,000). Moreover, thanks to the coarse-to-fine strategy of Prism, its time cost increases linearly with an increasing number of instances, making it scalable for handling massive instances in cloud systems.

4.5 Industrial Experience

In this section, we share our experience in applying Prism to a real-world cloud system (*i.e.*, Huawei Cloud), which aims to demonstrate the practical usefulness of Prism. Generally, customers usually subscribe instances from Huawei Cloud in a batch manner, *e.g.*, thousands of instances. These customers can then concentrate on the development and deployment of a variety of services across these instances, while the cloud providers handle the often tedious tasks of maintenance and operation to ensure system reliability. Due to privacy concerns, on-site engineers from Huawei Cloud can only rely on limited runtime information of these instances (*e.g.*, network packet drop rate) to monitor their health states [20, 86, 98]. However, without knowing how customers' applications are organized in these instances, we observe that some potential threats in the deployment or underlining errors may be missed, which may later cause service interruptions, consequently impacting the overall availability of the deployed applications [124]. To address this problem, in Huawei Cloud, we adopt Prism to reveal functional clusters in the massive instances hosted by Huawei Cloud. These functional clusters provide additional information regarding the structure of service deployment across the instances, thus enabling us to conduct more comprehensive and fine-grained monitoring of the cloud system. We present

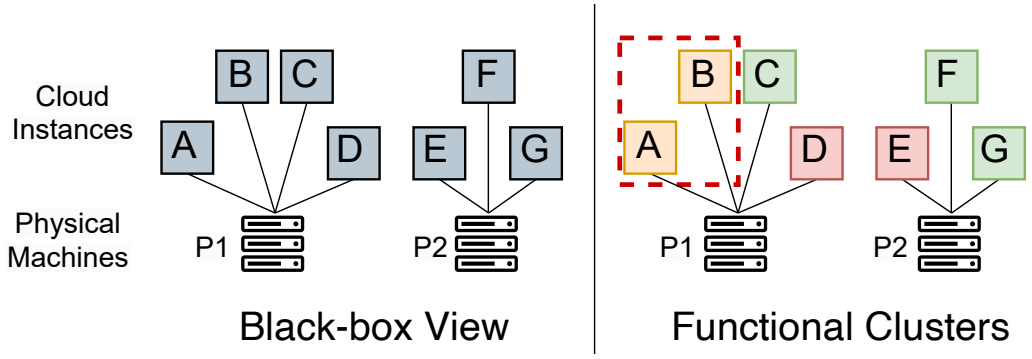


Figure 4.5: Case I: Vulnerable Deployment Identification

two primary usage scenarios of functional clusters within Huawei Cloud: *vulnerable deployment identification* and *latent issue discovery*.

4.5.1 Vulnerable Deployment Identification

Functional clusters can help cloud providers identify instances with vulnerable deployments. Specifically, a vulnerable deployment refers to a scenario where all instances, having the same functionalities, are deployed on the same physical machines. In such case, once a failure happens on this physical machine (*e.g.*, disk failure [96]), the entire functionality can be interrupted. In contrast, if these instances were distributed across different physical machines, only a subset of the instances would be affected in the event of a failure, thereby preventing a complete shutdown of the functionality. However, due to the abstraction of physical resources into instances, customers often deploy their applications within these instances without understanding how these instances are distributed across actual physical machines. On the other hand, cloud vendors possess knowledge of the mapping between instances and physical machines; yet, they are often unaware of the organization of functionalities across these instances due to privacy concerns. Given the vast number of instances in a cloud system, manually identifying vulnerable deployments poses a sig-

nificant challenge for on-site engineers.

To fill in this gap, we apply Prism to identify functional clusters to help detect potentially vulnerable deployments. Fig. 4.5 provides a concrete example. The left-hand side presents a black-box view of instance deployment from a cloud vendor's perspective, where only the information about which instances are deployed on which physical machines is known. In contrast, the right-hand side displays instances with functional clusters. With this knowledge, we can identify three functionalities: a functionality including A and B (marked as yellow), a functionality including D and E (marked as red), and a functionality including C, F, and G (marked as green). The deployment of the yellow functionality is potentially vulnerable because both A and B are deployed on physical machine P1. In contrast, the other two functionalities are more reliable since their instances are deployed across two different physical machines, making them resilient to the failure of either machine. It is worth noting that although Prism can hardly pinpoint what specific functionality of an instance serves, it can identify the instance group having the same functionalities, which facilitates automatic vulnerable deployment identification without violating privacy policies.

We have applied Prism in Huawei Cloud to discover functional clusters for around 3,000 internal instances and identified *eight* cloud services with vulnerable deployments. We then contacted the corresponding teams, confirmed the existence of the vulnerable deployments, and assisted in migrating the instances across different physical machines for improved resilience. In the future, our goal is to broaden the adoption of Prism to benefit a wider group of users and help enhance the reliability of their application deployment.

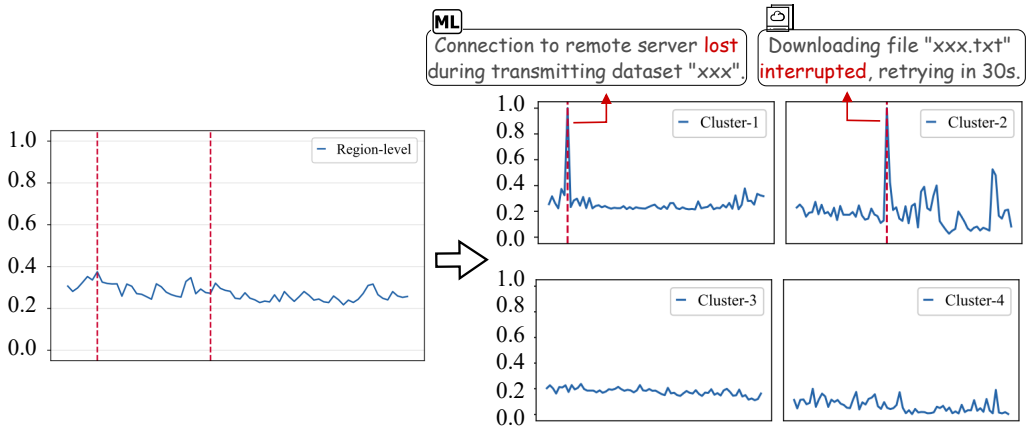


Figure 4.6: Case II: Latent Issue Discovery

4.5.2 Latent Issue Discovery

The second typical use case of Prism in Huawei Cloud is to identify latent network issues that may not be discovered by traditional monitoring methods. Modern cloud providers have been equipped with various monitoring tools to ensure the quality of their network services (*e.g.*, flow logging of AWS [7]). It is essential for such monitoring tools to comprehensively discover underlining problems in the cloud systems that can affect user experience, but without firing too many false alarms to distract the on-site engineers.

One crucial type of network monitoring is to monitor the packet loss of each instance. Packet loss, which denotes network packets that are accidentally dropped, can usually occur in any instance of a cloud system. However, they may not necessarily indicate a problem, as these errors could be caused by transient network congestion and may not affect users' experience. Considering the vast number of instances in a large-scale cloud system, a significant number of packets could be lost every minute. This presents a challenge for cloud vendors in converting this fragmented packet loss data into actionable alarms for on-site engineers.

To address this problem, we resort to the aggregation of packet

loss data from a selected group of instances, using an appropriate granular approach to identify potential problems. The underlying assumption here is that *simultaneous* packet losses occurring within a group of instances are more likely to impact user applications. For instance, if all instances within a region experience packet loss within a short time frame, it strongly suggests a regional network issue. However, one large region can contain millions of instances, and consequently, grouping by a region might fail to reveal local issues for a particular application. Another possible solution is to utilize the metadata (*e.g.*, the TenantID of the customer) to group instances. Nonetheless, there could still be tens of thousands of instances associated with the same identifiers [124]. For example, all instances subscribed to by the same enterprise customer would share the same identifier.

Prism enables a more effective approach, which is to aggregate lost packets in the granularity of the (approximated) functional clusters, which can reveal latent issues that may not be visible at neither a coarser level (*e.g.*, regional level) nor a finer level (*e.g.*, instance level). Fig. 4.6 shows the changes in the number of lost packets (normalized) calculated at either the region grain (left-hand side) or functionality grain (right-hand side). We can observe that while the numbers of packet loss barely change for the whole region, some functionalities (*i.e.*, Cluster-1 and Cluster-2) experience sudden increases in packet loss. This indicates that there may be latent issues affecting the performance of those specific functionalities, which are unnoticed if monitored at the region level. We then contact the corresponding teams and confirm that Cluster-1 and Cluster-2 correspond to machine learning and storage functionalities, respectively. We then validate these latent issues, and both functionalities experience interruption due to unstable network states, as evidenced in their log messages shown in Fig. 4.6. This highlights the potential of Prism in facilitating identifying issues that customers are experiencing without access-

ing their private data, which allows cloud vendors to provide more comprehensive monitoring to enhance the reliability of the cloud systems.

Enhanced Cloud Monitoring Based on Prism. To summarize, these two use cases demonstrate that functional clusters can be utilized with existing monitoring tools and enable identifying vulnerable deployment and discovering latent issues automatically. Prism plays a crucial role to provide comprehensive and precise functional clusters for large-scale instances. With the significant growth of modern cloud systems, instances experience frequent dynamic changes, including creation, deletion, and migration. In this context, Prism can be utilized to efficiently capture relations between instances. Unlike using pre-defined and rule-based monitoring [47, 86], Prism is adaptive to the frequent evolution of cloud applications. By continuously monitoring metrics like packet loss and the distribution of instance deployments, the monitoring system can effectively detect anomalies, such as sudden spikes in packet loss or scenarios indicating vulnerable deployments. This enables prompt alerts to the on-site engineers of relevant teams, resulting in shorter response time and more efficient issue resolution. Overall, the effectiveness and efficiency of Prism significantly contribute to improving the overall monitoring and management of instances in modern cloud systems.

4.6 Threats to Validity

External Validity. The primary external threat of this study is the investigated object. The datasets are collected from Huawei Cloud, as there are no publicly available datasets that include both instance data and corresponding functionality labels. However, Huawei Cloud is a world-leading cloud provider with a vast scale. The data collected from the production environment records real behaviors of instances and covers a broad range of functionalities

from two large regions as detailed in §4.4.1. Therefore, the Huawei Cloud evaluation is representative and convincing. The data used by Prism, which includes traces and metrics, is typically collected by modern cloud vendors like AWS [4] and GCP [44]. This suggests that our solution could be applied to similar cloud systems, potentially benefiting cloud customers globally.

Internal Validity. The primary internal factors that could potentially compromise validity are implementation and parameter setting. To address the implementation threat, we closely followed the original papers for baseline approaches that lacked open-sourced code and re-implemented them accordingly. To minimize this threat further, we utilized several mature libraries (*e.g.*, scikit-learn) for implementing the core algorithms. Moreover, both our proposed methods and the baseline methods were subject to rigorous peer code review. To mitigate the parameter setting threat, we fine-tuned the baseline methods utilizing a grid-search approach, subsequently selecting the most optimal results.

4.7 Summary

This chapter presents an approach to enhance the observability of cloud systems by inferring functional clusters of instances. To achieve this, we conduct a pilot study based on the real-world datasets collected in Huawei Cloud, indicating that communication patterns and resource usage patterns are two essential indicators for revealing functional clusters. Motivated by our findings, we propose a non-intrusive, coarse-to-fine clustering method, Prism, which effectively integrates both communication and resource usage patterns. Experiments on two industrial datasets are conducted to evaluate Prism. Our results show that Prism outperforms state-of-the-art solutions with a v-measure of 0.95; and Prism can efficiently process massive instances. Furthermore, we share our experiences in applying Prism in Huawei Cloud. Two

cases, *i.e.*, vulnerable deployment identification and latent issue discovery, demonstrate the usefulness of Prism in improving the reliability of Huawei Cloud.

□ **End of chapter.**

Chapter 5

LLM-enhanced Log Anomaly Detection

System logs play a critical role in maintaining the reliability of software systems. Fruitful studies have explored automatic log-based anomaly detection and achieved notable accuracy on benchmark datasets. However, when applied to large-scale cloud systems, these solutions face limitations due to high resource consumption and lack of adaptability to evolving logs. In this Chapter, we present an accurate, lightweight, and adaptive log-based anomaly detection framework, namely `Sealog`. The remainder of this chapter is organized as follows. Section 5.1 provides the problem background and contributions we made. In section 5.2, we conduct an empirical analysis to motivate our method design. Then section 5.3 introduces the method design of the proposed solution Sealog. Section 5.4 elaborates evaluation results of Sealog based on the real-world data collected from Huawei Cloud. Deployment experience of Sealog in Huawei Cloud is also shared in Section 5.5. Section 5.6 discusses threats to validation. Section 5.7 summarizes this chapter.

5.1 Problem and Contributions

Ensuring the reliability of cloud systems is a critical task [20, 25, 48], since a small period of downtime could result in significant financial loss for both cloud vendors and their customers [18]. A preliminary step to safeguard reliability is timely and accurate detection of suspicious system behaviors, *i.e.*, anomaly detection. Similar to traditional software systems, logs in cloud systems provide valuable insights into the system's functioning and potential issues. Log-based anomaly detection, *i.e.*, timely identifying anomalous log messages for prompt resolution of issues, has been widely recognized as an essential task of cloud system management [26, 54–56, 181].

Existing approaches typically adopt machine learning or deep learning-based techniques to identify anomalous logs. Traditional machine learning-based approaches primarily consider statistical information (*e.g.*, log occurrence counts) and apply models such as isolation forest (IF) [95], support vector machine (SVM) [92], decision tree (DT) [21], and logistic regression (LR) [10], to identify anomalies. Besides, recent studies have explored the use of deep learning methods to process logs. Such approaches typically extract semantic information from log messages through word embedding [162, 172] or Bidirectional Encoder Representations from Transformers (BERT) [79], and perform anomaly detection accordingly.

Although previous studies have demonstrated impressive performance on benchmark datasets, they are not practical for production cloud systems due to the following two reasons. First, previous solutions tend to pursue a high detection accuracy while overlooking the optimization of computation and space complexity. Cloud systems often use instances (*i.e.*, virtual machines) to host customers' applications. Conducting log anomaly detection for each instance enables close monitoring of its health sta-

tus. However, as customers' applications already take up most resources of the instances, there are limited resources left to run an anomaly detector, which renders existing solutions impractical. For instance, deep learning-based methods [79, 172] require heterogeneous accelerators (*e.g.*, GPUs) [115, 148] to allow real-time inference, which may not be available to every instance. A straightforward approach is to transmit the log data to centralized compute nodes with abundant computational resources, which subsequently return the detection results. However, instances in a cloud can produce extensive amounts of log data (*e.g.*, Azure reported that 5 billion log messages are generated per day [146]), that are distributed across different clusters and datacenters. Transmitting such large-scale distributed logs to compute nodes could cause additional network and I/O overhead, which is prohibitively expensive and time-consuming.

The second reason is that previous approaches struggle to be sufficiently adaptive to deal with diverse and evolving log data. In cloud systems, frequent launches of new software versions result in changes to logging statements over time. For instance, Google has reported that there are thousands of newly added logging statements due to software updates every month [160]. Existing methods typically train models to learn anomaly patterns from historical log data, which can hardly be adjusted to unseen logs, causing performance degradation. To address this problem, a recent study [172] exploits the semantic similarity between historical logs and new logs, enabling the algorithm to transfer knowledge about anomalies from historical data to new data for anomaly identification. However, logs in real systems are complicated, and whether the assumption (*i.e.*, new logs share similar semantics with the historical ones) holds has not been well investigated.

In this chapter, we conduct a study on the logs in Huawei Cloud to better understand the characteristics of logs in production cloud systems. We observe that an instance could generate several gi-

gabytes of logs daily, while only very limited resources (*e.g.*, one CPU core and 200MB memory) are left for a plug-in anomaly detection process. Besides, logs are evolving and have low semantic similarities. Specifically, during a one-month-long development cycle of 20 microservices, approximately 14.5% of new logs are introduced on average. When comparing the semantic similarities between two versions of the same microservice, we find that more than 90% of log message pairs share little semantic similarity. This implies that learning semantics from previous logs is not adaptive enough to handle new logs. Based on these findings, we can summarize that a practical log-based anomaly detection method for cloud systems should be *accurate*, *lightweight*, and *adaptive*. It is non-trivial to achieve all the above three requirements simultaneously. Lightweight anomaly detection methods (*e.g.*, logistic regression [10]) cannot effectively handle newly occurring anomalous logs (*i.e.*, not adaptive). On the other hand, existing "adaptive" attempts (e.g. RobustLog [172]) rely on compute-intensive neural networks to apply learned semantic information to new logs (*i.e.*, not lightweight). Additionally, neither type of method has the few-shot ability for swift adaptation, meaning they cannot handle log data dissimilar to historical data which is common in real-world scenarios.

Our work. Inspired by the recent development of large language models (LLMs), we propose to *integrate the superior few-shot ability of LLMs and traditional machine-learning methods* in a synergistic manner to meet these demanding requirements simultaneously. Specifically, we propose a novel log-based anomaly detection framework, named Sealog, which consists of two components *i.e.*, (1) a lightweight *detection agent*, and (2) an accurate and adaptive *backbone analyzer*. The backbone analyzer, powered by LLMs, is designed to identify anomalies by directly comprehending the semantics of logs. Since LLMs have been trained on extensive natural language corpus, it becomes feasible to ef-

fectively identify anomalies even for unseen logs, hence achieving adaptiveness. However, solely querying LLMs is too expensive to analyse the entire set of logs, especially considering that each query to LLMs would incur charges based on the length of input data, and it is time-consuming to process a large dataset. To overcome this issue, our framework involves a lightweight detection agent to interact with the backbone analyzer. The agent can filter the majority of normal log data efficiently and forward a limited number of suspicious logs to the backbone analyzer for semantic-based analysis. In doing this, the detection agent can be deployed within resource-constrained instances without incurring voluminous network transmitting costs. Additionally, we allow both detection agent and backbone analyzer to take human feedback, which further enhances the adaptability of Sealog to evolving log data in practice.

To evaluate the performance of Sealog, we conduct extensive experiments on two widely used public datasets and an industrial dataset collected from the production environment of Huawei Cloud. The experimental results demonstrate that compared with state-of-the-art solutions, Sealog achieves the best performance (0.949 to 0.993 F1 scores) in the setting where log data do not evolve. In addition, Sealog retains a high and stable accuracy even if log messages evolve. Moreover, compared with existing methods, the Sealog is 2× to 5× faster and its detection agent consumes only 5% to 41% memories. This chapter makes the following contributions:

- We propose Sealog, the first synergistic framework fusing the advantages of machine-leaning approaches and LLMs for log-based anomaly detection. It comprises a lightweight detection agent and a backbone analyzer to meet practical requirements, *i.e.*, accurate, lightweight, and adaptive.

- We conduct extensive evaluations of Sealog on both public and industrial datasets. The results demonstrate that Sealog out-

performs state-of-the-art methods in processing fixed or evolving log data, runs 2× to 5× faster and only consumes 5% to 41% of the memory resource.

- We have deployed Sealog in a real-world cloud system (*i.e.*, Huawei Cloud) for twelve months, which demonstrates stable performance (*i.e.*, 0.9 F1 on average) running in our production environment. We share our experience and open-source Sealog to benefit the community.

5.2 Background and Motivation

In this section, we first discuss the background of log-based anomaly detection. Then, we conduct a study to understand the attributes of logs in cloud systems, from which we derive the industrial requirements that guide our method design.

5.2.1 Log-based Anomaly Detection

The goal of log-based anomaly detection is to identify the log messages that may indicate a system problem in the runtime. In the literature, most studies adopt a two-step paradigm *i.e.*, *log parsing* and *anomaly detection* [26, 55, 80]. In the log parsing step, these methods obtain log templates by identifying the constant and variable parts. For instance, consider the raw log message "Finished task 0.0 in stage 6.0 (TID 247)." Its log template would be "Finished task $*$ in stage $*$ (TID $*$)," where the parameters – "0.0, 6.0, 247" are replaced with '$*$'. The sequence of log templates is then processed through various downstream methods to conduct anomaly detection. For example, the library Loglizer [55] applies various machine learning-based methods (*e.g.*, logistic regression and decision tree) to detect anomalies based on the count distribution characteristics of templates.

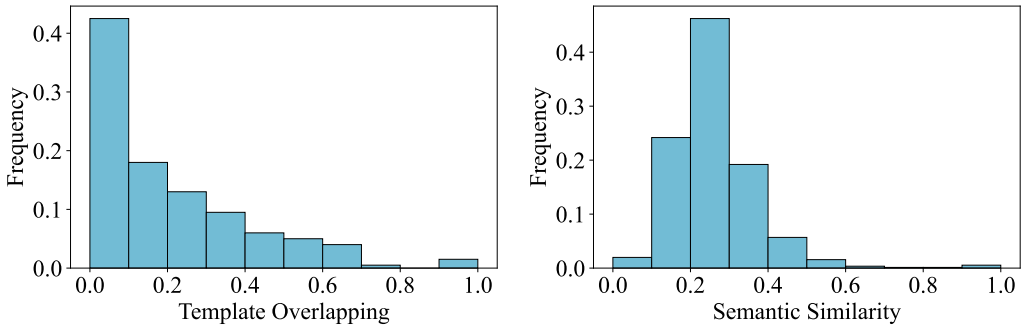
Although log-based anomaly detection methods have long been recognized as an important problem for software maintenance, their practical application in cloud scenarios is still understudied. Unlike traditional software, cloud systems have a larger scale, generating massive volumes of diverse logs from millions of physical and virtual instances. We investigate the log data characteristics in Huawei Cloud through the following study.

5.2.2 Characteristics of Log Data in Cloud Systems

In the following study, we aim to understand the characteristics of log data in a typical cloud system, *i.e.*, Huawei Cloud; then summarize practical requirements for log-based anomaly detection methods in cloud systems.

Massive and Distributed Log Data.

The huge volume of logs has been widely recognized as an essential challenge in recent studies [99, 146]. Addressing this issue is increasingly challenging within the context of the widely adopted microservice architecture of cloud systems [65]. For instance, an application may comprise hundreds of microservices [103, 179], with each one generating tens of gigabytes of log data per day as it supports a large number of users. Transmitting the complete log data of each microservice to a centralized machine for automated analysis is typically impractical, as it can consume significant network bandwidth and can also lead to time delays in promptly identifying anomalies in the runtime. Therefore, it is crucial to directly conduct log-based anomaly detection in a distributed manner within each instance. However, an instance's main functionality (*e.g.*, video stream) usually can consume most of the resources, leaving very limited resources for plug-in processes such as log-based anomaly detection. For example, in Huawei Cloud, we find that most instances allocated for control



(a) Templates Overlapping across 20 Different Microservices

(b) Semantic Similarities of Log Templates between Two Time Frames

Figure 5.1: Statistics of Log Messages in Huawei Cloud

plane services (used for management and coordination of various resources) have only ∼200MB memory and a single CPU core that can be allocated to an anomaly detector without adversely affecting the main functionality running on the instance. *Therefore, it is crucial for the anomaly detector to be lightweight enough, enabling it to be deployed locally within the instances for continuous monitoring.*

Diverse and Evolving Log Data.

Log data can evolve frequently due to software changes. Though previous studies have attempted to address this problem [61], they rely on semantic similarity between unseen and seen logs. However, this may not hold in cloud systems which have significantly diverse log data. To better understand the characteristics of log evolution in Huawei Cloud, we collected log data generated by 20 microservices of the same application from 01/02/2023 to 01/03/2023. Then, we study the following two aspects:

(1) How many log templates are shared across different microservices?

To answer this question, we obtain the templates of the collected log data by running a log parser Drain [53] following pre-

vious studies [146, 172]. Subsequently, we form pairs of every two microservices and compute their template overlapping using Jaccard similarity, which is the ratio between the number of common templates and the total number of unique templates in the two microservices. As depicted in Figure 5.1(a), approximately 40% of microservice pairs exhibit no overlapping log templates. Furthermore, the proportion of pairs with a similarity measure of less than 0.5 constitutes roughly 80% of the total pairs. This shows that microservices with different functionalities tend to log differently, though they share a small number of log templates as they may involve some common components while developing.

(2) How similar are the logs in terms of semantics while the software is evolving?

Moreover, we also investigate the evolution of log semantics over time. To achieve this, we partition all log data based on the date of 15/02/2023 and then proceed to calculate the semantic similarity between each pair of log messages before and after this particular time point within each microservice. Specifically, we utilize the OpenAI embedding service [122] to obtain a semantic vector for each log message. Then, we measure the semantic similarity of two log messages in terms of the cosine similarity of their corresponding semantic vectors. We plotted the cumulative distribution function (CDF) of the normalized similarities, which range from 0 to 1, for these log pairs in Figure 5.1(b). The figure illustrates that the majority of log pairs (over 95%) exhibit similarities of less than 0.5. These findings indicate that software updates have the potential to introduce new log messages that differ significantly from historical logs.

In summary, log data in cloud systems exhibits diversity, with different microservices having distinct logs, and undergoes evolution, as new logs display minimal semantic similarities with old ones. Consequently, previous studies focusing on transferring old semantics to new ones may be ineffective. *This underscores the*

need for a method capable of adaptively accommodating the evolving log semantics in cloud systems.

Insights: In addition to **accuracy**, a practical log-based anomaly detector needs to be **lightweight** enough for deploying as a plug-in of resource-constrained instances. Furthermore, it needs to be **adaptive** to evolving log semantics, enabling effective identification of anomalies within the dynamic landscape of cloud environments.

5.3 Methodology

5.3.1 Overview

Figure 5.2 provides an overview of the workflow of Sealog. First, the complete set of raw input logs is processed by a detection agent to identify suspicious log windows. We design a lightweight n-gram probabilistic tree (NPT) to serve as the core of the detection agent. Its primary objective is to efficiently filter out the majority of normal logs, thereby leaving a limited number of suspicious log windows for further analysis. Second, a backbone analyzer determines anomaly and provides explanation by leveraging the zero/few-shot capabilities [13] of large language models (LLM), such as ChatGPT [123], which can comprehend the semantics of log messages and facilitate accurate and adaptive anomaly detection. In addition, we employ in-context learning (ICL) [13] [158] to learn from historical fault for more precise decisions (anomaly/normal) and explanations. Finally, the anomalies are sent to on-site engineers for confirmation, and the confirmed cases (the raw log window and its groundtruth label) are archived into a fault library. These cases serve as continuous feedback to enhance the accuracy of both the detection agent and backbone analyzer, further enhancing the adaptiveness of our framework Sealog. The synergy between the detection agent and backbone analyzer is

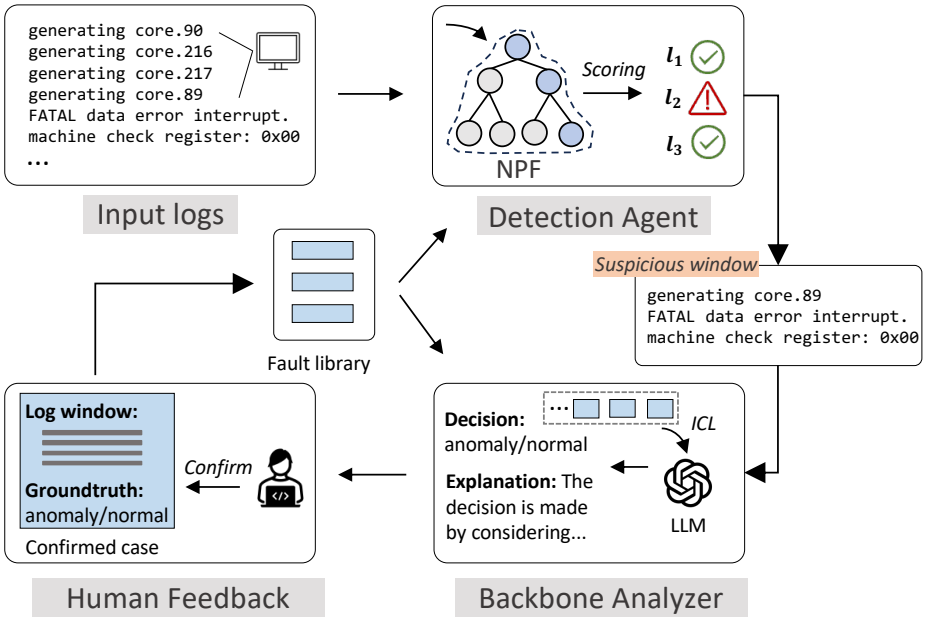


Figure 5.2: The Overall Workflow of Sealog

pivotal in Sealog. The detection agent *efficiently* filters out the majority of normal logs, allowing for more *accurate* and *adaptive* analysis through the use of more resource-intensive computation methods (e.g., LLM). This is because only a small number of queries require analysis, leading to decreased bandwidth, token costs, and analysis time.

5.3.2 Detection Agent

The aim of detection agent is to filter the majority of normal cases and retaining a small number of suspicious cases. Furthermore, the detection agent prioritizes recall over precision, aiming to capture as many suspicious cases as possible. This emphasis on recall is crucial, as any undetected suspicious cases would not be forwarded to the backbone analyzer, leading to the irreversible omission of anomalies. However, this will not result in a substantial increase in the number of cases forwarded to the backbone analyzer, as anomalies are significantly less frequent than normal

cases by definition. To achieve this while ensuring efficiency, we propose an n-gram probabilistic tree (NPT) as the core of the detection agent. In the following, we introduce the construction of NPT and anomaly detection based on it.

NPT Construction

The goal of NPT is to efficiently find unusual words or template patterns within system logs, which barely appear when the software system is running in a normal state [168], *i.e.*, suspicious anomalies. Existing anomaly detectors either adopt deep learning-based methods that are not lightweight enough [26, 35, 79], or utilize traditional machine-learning methods that cannot effectively accommodate unseen logs [55]. Consequently, they are unable to achieve the goal of the detection agent due to their limitations.

To address these limitations, we design a simple yet effective structure, namely NPT, which serves the dual purpose of parsing logs and identifying suspicious anomalies. Additionally, it has the capability to incorporate newly labeled data for adaptation (*e.g.*, avoid duplicate false-positive samples to query the backbone analyzer). Particularly, we detect anomalies in the granularity of log windows (e.g., 10 minutes). Given one log window, the NPF is constructed in three phases: preprocessing, signature-based grouping and sequential clustering.

Preprocessing. For an input log message l_i, we follow a common practice [146, 181] to use pre-defined regular expressions to extract parameter fields such as IP address and URL. After that, we tokenize the log message with non-alphanumeric splitters (*i.e.*, any characters that are not letters or numbers), generating log tokens (or words) denoted as $\hat{l}_i = [w_1, w_2, ...]$.

Signature-based Grouping. We then compute a grouping signature for each log message to roughly group similar logs together. The widely-adopted log parser Drain [53] adopt common prefix words to achieve this, assuming that these logs are more likely to

belong to the same template. However, this assumption may not hold true for complex log data in practical scenarios. Instead, we utilize grouping signatures with a less stringent assumption. Each group signature comprises three parts: *i.e.*, inherent attributes, top-frequent tokens, and its splitter set. (1) Inherent attributes, such as logging levels (e.g., INFO and DEBUG) and components, are considered because log messages with different levels or originating from different components are more likely to belong to separate templates. (2) Top-frequent tokens, typically the most frequently occurring K tokens within a log message (usually with K=3), are included in the log signature. This is because they often represent constant parts of a log template [99], with English stopwords excluded. (3) The splitter set consists of unique non-alphabetical tokens within a log message, representing its format. For example, in the log message "`user=root,delay=10,stat=queued`" the unique splitters are "=". We use the splitter set because logs with the same templates often share the same format [66], such as the presence of "=," indicating a key-value format. Those logs sharing the same grouping signature are sent to the same leaf node in the NPT.

Sequential Clustering. In practice, the proposed grouping signatures can already lead to leaf nodes containing cohesive log messages. Then, we are allowed to directly adopt lightweight sequential clustering [99] in each leaf node to generate more fine-grained *log clusters* effectively. Each log cluster comprises a template and its corresponding mark. The template is derived by analyzing the logs sequentially assigned to the cluster. The mark is an annotation indicating whether this template represents an anomaly or a normal event, as determined by human feedback. The mark is set to null by default if no feedback is given.

Specifically, as shown in Figure 5.3 (right-hand side), each log message tries matching with existing log clusters. If it cannot match with any log cluster, it forms a new cluster, with the tem-

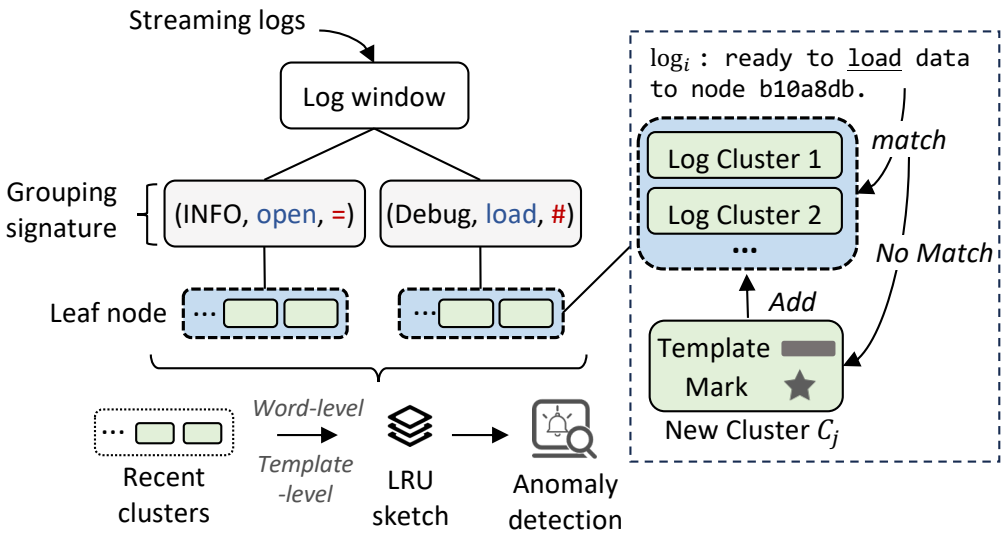


Figure 5.3: The Structure of N-gram Probabilistic Tree (NPT)

plate being the log message itself. Otherwise, the log message is assigned to an existing log cluster, and the log template associated with the cluster is updated. This is achieved by comparing the words of the existing log template with the arriving log message. The differing words are treated as parameters and marked as $*$ in the updated log template.

NPT-based Anomaly Detection

After processing a log window with the NPT, we detect anomalies by determining whether the window contains unusual words (*i.e.*, at word level) or sequence patterns (at template level). To achieve this efficiently, we propose to extract word-level and template-level LRU (least recently used) sketches to inform decisions. The core idea behind the LRU sketches is to maintain a compact data structure that continuously tracks whether a specific word or template combination has recently become unusual.

LRU-Sketch Computation. We use LRU sketches to capture anomalies at both the word level and template level. For sim-

Table 5.1: Content of Word-level LRU Sketch

Symbol	Content
$C(ng_i^w)$	Total count of n-gram words ng_i^w.
$C_a(ng_i^w)$	Total count of n-gram words ng_i^w that are shown in abnormal windows.
$C_n(ng_i^w)$	Total count of n-gram words ng_i^w that are shown in normal windows.

plicity, we will only explain the word-level LRU sketch and its corresponding anomaly detection procedure, omitting the detailed discussion of the template level, which follows a similar approach. Specifically, the word-level LRU sketch is computed utilizing data from the most recently observed log windows within a specific time period. Each sketch comprises multiple variables detailed in Table 5.1. Specifically, ng_i^w represents the *n-gram word*, which is defined as a contiguous sequence of n words derived from a log template, denoted as $ng_i^w = \{w_i, ..., w_{i+n-1}\}$, and $C(ng_i^w)$ denotes its total count in the most recent seen windows. In addition, we use $C_a(ng_i^w)$ and $C_n(ng_i^w)$ to denote the total occurrences of ng_i^w shown in abnormal windows or normal windows, respectively. We continuously update these statistics in the least recently used manner, *e.g.*, by only considering data observed within the past ten days to mitigate the impact of outdated information.

Suspicious Anomaly Identification. We then identify suspicious anomalies based on the LRU sketch, which is formulated as a binary classification problem (*i.e.*, anomaly or normal). Specifically, we propose a streaming classifier based on Bayes' theorem [11], designed to run efficiently and with minimal parameter requirements, making it memory-friendly.

After a log message L is assigned to a log cluster in NPT, we extract its associated statistic data stored in the LRU Sketch based on its n-grams. We then perform the following computation, where we omit the superscript w for simplicity:

$$p(y=1|L) = p(y=1) \times \prod_i p(ng_i|y=1)) \tag{5.1}$$

$$p(y=0|L) = p(y=0) \times \prod_i p(ng_i|y=0)), \tag{5.2}$$

$$p(ng_i|y=1) = \frac{C_a(ng_i)+\alpha}{\sum_j ng_j+\alpha\times u}, p(ng_i|y=0) = \frac{C_n(ng_i)+(1-\alpha)}{\sum_j ng_j+(1-\alpha)\times u} \tag{5.3}$$

where y denotes the label for anomaly detection. $p(y=1)$ and $p(y=0)$ are the prior probabilities for anomalous or normal log windows, respectively. These probabilities are calculated as the ratio of the number of anomalous (or normal) log windows to the total number of log windows encountered so far. The item $p(ng_i|y=1))$ is the likelihood of the i_{th} n-gram indicating an anomaly as described in Equation 5.3. The LRU sketches are typically initialized using labeled training data following existing studies [79,172]. This data can be gathered from historical failures or through fault injection [83, 84]. Furthermore, our LRU-Sketch design can continue to evolve by updating Table 5.1 after new labeled data (feedback) becomes available.

In practice, unseen n-gram words can appear and their statistics are not recorded for anomaly computation. To mitigate this issue, inspired by the Lidstone smoothing [153], we propose to introduce a parameter $\alpha \in [0, 1]$ to compute the likelihood as shown in Equation 5.3. Here, u is the number of unique n-gram words (*i.e.*, vocabulary size). In particular, we intentionally set a larger parameter, *e.g.*, $\alpha = 0.8$. In doing this, when dealing with unseen n-gram words, we have both $C_a(ng_i) = 0$ and $C_n(ng_i) = 0$. A large α tends to increase $p(ng_i|y=1)$ more than $p(ng_i|y=0)$. Hence, we regard new n-gram words as having a much higher likelihood of indicating anomalies. This design is crucial because those missed anomalies cannot be revisited by the subsequent backbone analyzer. Therefore, the detection agent must apply a lenient standard to identify suspicious anomalies.

Finally, we average both word-level and template-level anomaly

scores to obtain the final anomaly scores, *i.e.*, $\bar{p}(y = 1|L)$ and $\bar{p}(y = 0|L)$. We then normalize each of the anomaly scores within the range of $[0, 1]$ by dividing them with their summation. This normalization process allows for better comparison and more intuitive interpretation of the anomaly scores. Particularly, if a log template has been confirmed as an anomaly or not, i.e., it has a non-null mark value (e.g., 0 or 1), we directly use the mark as the anomaly score. Then, we introduce a querying threshold, denoted as θ_{query} for the detection agent. The detection agent will only raise a query when the normalized anomaly score $\bar{p}(y = 1|L)$ exceeds θ_{query}. This allows us to control the number of queries raised by the detection agent.

5.3.3 Backbone Analyzer

We have deliberately designed the detection agent to be lightweight to efficiently process large amounts of log data. However, this approach can inevitably result in false positive alarms, particularly when encountering unseen tokens or sequence patterns in the log data. To tackle this issue, we propose utilizing large language models (LLM) as a backbone analyzer to analyze the detection agent's suspicious results and make more accurate predictions. The reason for adopting LLM is that it has been pre-trained with an extensive corpus (*e.g.*, source code, software documents, and bug reports [175]). This enables LLM to comprehend the log data reported by the detection agent by reasoning with the existing knowledge it has acquired. While directly processing the entire set of log data with LLM is inefficient and costly, the Sealog framework allows us to leverage the power of LLMs once the detection agent has filtered out the majority of normal log data, assuming that anomalies are rare in most software systems [80].

However, LLMs were originally designed for open-world questioning and answering, rather than specifically for the task of log-

based anomaly detection. In order to make LLM suitable for our specific task and to provide it with domain-specific knowledge, we adopt LLM with in-context learning (ICL) as the backbone analyzer of Sealog. ICL provides LLM with *examples* to make LLM learn from the contextual information in the examples and make answers according to the format in the examples, which has been proved to enhance the reasoning ability of LLM significantly [13, 40]. Specifically, our LLM-based backbone analyzer has three steps: *example retrieval*, *prompt formulation* and *LLM querying*.

Example Retrieval

Previous studies have shown that examples that are semantically similar to the query are more effective than random ones [40, 158]. We follow this idea in our backbone design. For every query window q, we select K nearest neighbors (KNN), which are the most similar log windows, from a candidate library of annotated training logs. To achieve this, we first embed all log windows in the training data as a library of semantic vectors using the OpenAI Embedding service [122], denoted as $V = \{v_1, v_2, ...\}$. Similarly, each query q is also embedded as a semantic vector v_q. We then retrieve K examples whose semantic vectors in V are closest to v_q measured by Euclidean distance $d(v_i, v_j) = \sqrt{\sum_u (v_{iu} - v_{ju})^2}$. By doing this, we are able to select K examples that have similar semantics to q, which facilitates LLM's prediction of q.

In practice, computing Euclidean distances directly by comparing v_q with every vector in V can be time-consuming, particularly when V is large and continues to grow as more cases are accumulated during the operations of on-site engineers. To accelerate the retrieval process, we accelerate the similarity search based on FAISS (Facebook AI Similarity Search) [71]. FAISS can build indices for vectors in V and partition the vector space into smaller subspaces, allowing for faster search within these subspaces.

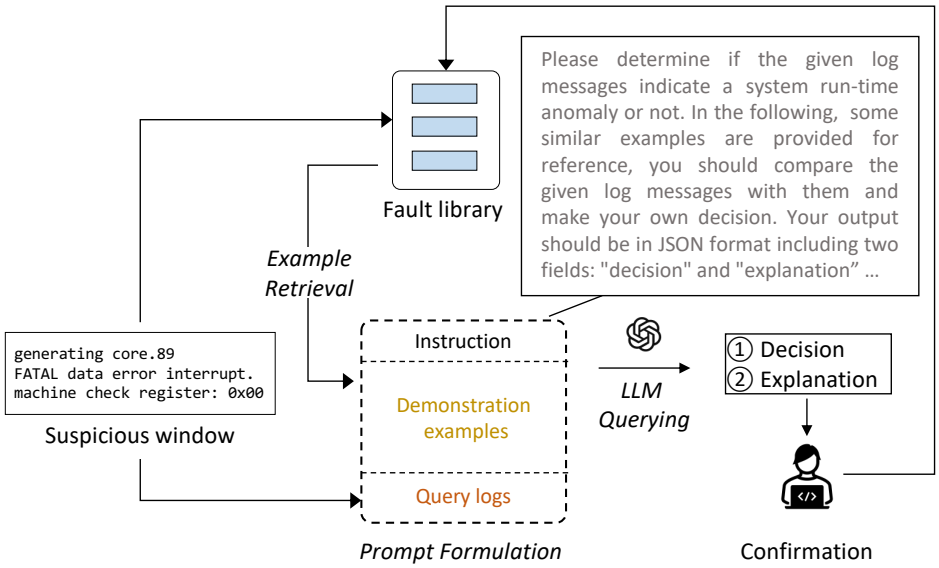


Figure 5.4: Processing Suspicious Windows with Backbone Analyzer

Prompt Formulation

The prompt for LLM comprises three components, as depicted in Figure 5.4. First, there is the *instruction*, which defines the log-based anomaly detection task and establishes the desired output format for LLM. Specifically, we regulate the output of LLM using JSON format, enabling precise parsing of LLM's decisions and excluding any irrelevant text generated by LLM. (2) the *examples* that are used to provide demonstrations for LLM. In the example demonstration, we order the examples by placing the more similar examples close to the query. As highlighted by recent studies [40, 158], this approach can facilitate LLM with learning relationships between the examples and the query, thus improving the overall performance of LLM. (3) the *query* that should be predicted by LLM. The log messages for prediction are directly appended after the instructions and examples. Due to space limitations, we include the specific details of the prompt in our replication package.

LLM Querying

We then utilize the constructed prompt to query an LLM (*e.g.*, GPT-3.5). The responses obtained from the backbone include a decision regarding whether the input is classified as an anomaly or normal. Those anomalies are reported to on-site engineers for prompt mitigation. Additionally, the responses provide corresponding explanations that aim to aid on-site engineers in confirming the classifications. After a case is confirmed, the input log window and its confirmed label are stored in the fault library to facilitate future analysis. Additionally, we allow engineers to annotate each log template as an anomaly or not, which is saved in the mark within the log cluster, further enhancing the adaptiveness of Sealog.

Despite our empirical observations in Section 5.2.2 suggesting a minimal overlap in semantics among log entries, our methodology essentially differs from prior research [172]. The retrieved examples are instrumental in illustrating the anomaly detection process and the expected structure of the output [40]. Our strategy actively seeks out the most closely related examples to maximize the effectiveness of the ICL process to enhance LLM.

5.3.4 Deployment

Real-world deployment of Sealog consists of two phases: offline warming-up and online detection. The *offline warming-up* phases provide the detection agent and backbone with labeled log data for training. Specifically, for the detection agent, these log data are processed to construct the initial NPT structure and LRU sketches. For the backbone analyzer, these log data are used to build the candidate library. During the *online detection* phase, the NPT continuously processes streaming log messages and raises suspicious logs to the backbone analyzer. The backbone analyzer conducts ICL-based reasoning by referring to cases retrieved from

the candidate library. Once a decision is made by the backbone analyzer and confirmed by on-site engineers, they are provided as feedback for future analysis. In Section 5.5, we share more details regarding our practical experience with the real-world deployment of Sealog.

5.4 Evaluation

We answer the research questions (RQs) in this section.

- RQ1: How effective is Sealog under the *offline* setting?
- RQ2: How effective is Sealog under the *online* setting?
- RQ3: How does the number of queries affect the performance of Sealog?
- RQ4: What is the time and space efficiency of Sealog?

5.4.1 Experimental Setting

Dataset

We evaluate Sealog on two widely-used public benchmarking datasets and one industrial dataset from the production environment of Huawei Cloud. Table 5.2 provides the statistics of these datasets. Specifically, the BGL (Blue Gene/L) dataset is a supercomputing system log dataset collected by Lawrence Livermore National Labs (LLNL) [56,121]. The Thunderbird dataset originates from a Thunderbird supercomputer at Sandia National Labs (SNL) [56, 121]. Although existing studies often use 10 million continuous lines from the Thunderbird dataset for evaluation [79, 80], they do not specify which 10 million logs they employed; therefore, we use the first 10 million logs in this study. The industrial dataset is collected from the production environment of Huawei Cloud. It contains one month-long log data generated by 20 internal microservices that offer various functionalities such as API Gateway,

user management, auto-scaling, load balancing, and more. Experienced engineers from our site reliability engineering (SRE) team assist us in labeling the dataset based on their historical diagnostic reports. These reports record the starting times of actual system issues, which facilitate accurately identifying the anomalous log messages manually. After a thorough labeling process, we obtained a log data dataset with approximately 1.5 million lines of log messages covering a wide range of system problems. We list the details of 103 types of failures in our replication package.

Evaluation Metrics

We calculate four measures: TP (true positive), which is the number of correctly predicted anomaly windows; FP (false positive), which is the number of predicted anomaly windows that are actually normal; TN (true negative), which is the number of correctly predicted normal windows; and FN (false negative), which is the number of predicted normal windows that are actually anomalous. Based on these numbers, we calculate precision (PC)=$\frac{TP}{TP+FP}$ and recall (RC) =$\frac{TP}{TP+FN}$ and F1 scores=$\frac{2\times PC\times RC}{PC+RC}$.

Comparative Methods

We have selected the following state-of-the-art studies as our comparative methods:

- **Loglizer (IF/LR/DT)** [55] is a log-based anomaly detection library encompassing a variety of machine learning (ML) methods, including isolation forest (IF), linear regression (LR), decision tree (DT), etc.

- **DeepLog** [35] employs a Long Short-Term Memory (LSTM) model as its core component. DeepLog takes windows of log messages as input and predicts the next log event. Anomalies are detected if the prediction differs from the actual event.

Table 5.2: Statistics of Evaluation Datasets

Dataset	BGL	Thunderbird	Industry
# Log messages	4,747,963	10,000,000	1,488,073
# Templates	456	1,504	3,241
# Train windows (anomaly ratio)	2,884 (21%)	416 (55%)	3,048 (13%)
# Test windows (anomaly ratio)	722 (24%)	105 (30%)	933 (18%)

- **LogAnomaly** [108] aims to detect anomalies by combining log count vectors and log semantic vectors, which also utilizes a forecasting-based mechanism to reflect anomalies.
- **LogRobust** [172] aims to capture the meaning of log messages using word embeddings to handle the challenge of constantly changing log messages. Each log message is encoded as a representation based on word vectors using an attention-based Bidirectional LSTM.
- **NeuralLog** [79] aims to bypass the log parsing step; instead, NeuralLog extracts semantic meaning from raw log messages and represents them as vectors to detect anomalies using a Transformer-based classification model.

Implementation Details

We implemented the prototype of Sealog with approximately 1000 lines of Python code for easy use in Huawei Cloud. We utilize ChatGPT (gpt-turbo-3.5-0301) as the LLM in the backbone analyzer, which is widely recognized for its advanced language understanding capabilities. For the baselines, NeuralLog and Loglizer, we directly utilized the code provided by the respective authors. Additionally, for DeepLog, LogAnomaly, and LogRobust, we utilized the DeepLoglizer library [26] for implementation. We set

Table 5.3: Experimental Results of End-to-end Offline Anomaly Detection

Method	BGL			Thunderbird			Industry		
	Precision	Recall	**F1 score**	Precision	Recall	**F1 score**	Precision	Recall	**F1 score**
IF	0.125	0.615	0.208	0.291	0.968	0.448	0.176	0.994	0.299
LR	0.738	0.437	0.549	0.842	0.516	0.640	0.818	0.655	0.727
DT	1.000	0.570	0.726	1.000	0.839	<u>0.912</u>	0.942	0.788	0.858
DeepLog	0.241	0.895	0.380	0.295	1.000	0.456	0.358	0.909	0.513
LogAnomaly	0.268	0.862	0.409	0.307	1.000	0.470	0.360	0.927	0.519
RobustLog	0.942	0.961	<u>0.951</u>	1.000	0.710	0.830	0.984	0.764	0.860
NeuralLog	0.881	0.886	0.883	0.713	0.719	0.715	0.889	0.895	<u>0.887</u>
Sealog	0.994	0.991	**0.993**	1.000	0.903	**0.949**	1.000	0.931	**0.964**

K=3 and evaluate the settings of important parameter θ_{query} for Sealog in Section 5.4.2. For baseline methods, we utilize the default parameters provided by the original source code. We use the same window setting for the public dataset as in [80], namely, using a fixed window size of one hour without overlapping. For the industrial dataset, we use 10 minutes as the window size for real-time anomaly detection.

5.4.2 Experimental Results

Effectiveness under Offline Setting (RQ1)

In this RQ, we evaluate the effectiveness of Sealog in an offline setting, where the training and testing sets are fixed. We evaluate the overall performance of Sealog as well as the performance of each of its components. Following the setting of previous studies [79, 80], we split the whole dataset in chronological order, with the first 80% data for training and the remaining 20% data for testing, and keep them fixed. We use the training data for warming up Sealog as described in 5.3.4 and training the baseline methods. Testing data is used for evaluation. The statistics of training and testing data are shown in Table 5.2.

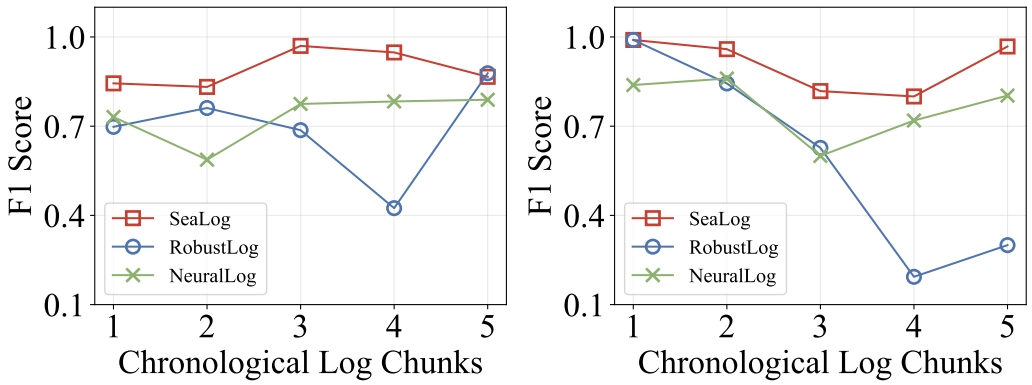
The evaluation results are presented in Table 5.3, the highest F1 score is emphasized in **bold**, and the second-best score is <u>underlined</u>. We can make the following observations: (1) Unsupervised methods (IF, DeepLog, and LogAnomaly) show signifi-

cantly lower effectiveness than supervised approaches in terms of F1 score. This is mainly because they assume that all training data is normal. As a result, during the inference phase, these methods classify every window containing unseen log templates as anomalies, resulting in high recall but low precision. (2) Supervised methods achieve a better balance between recall and precision by learning from labeled data. Furthermore, with sufficient labels, ML-based methods (i.e., DT) can still achieve comparable or better performance (on Thunderbird) compared to DL-based methods. The above two observations are consistent with the recent empirical studies [26, 80] that benchmark existing DL-based methods. (3) Sealog attains the highest F1 score across all three datasets, ranging from 0.949 to 0.993. This result demonstrates the effectiveness of the collaboration of the detection agent and the backbone analyzer. We then discuss their individual effectiveness in section 5.5.

Answer to RQ1. Sealog achieves the best overall performance among all state-of-the-art baselines across three benchmarking datasets, demonstrating that Sealog can effectively fuse the results of the detection agent and the backbone analyzer.

Effectiveness under Online Setting (RQ2)

In this RQ, we would like to evaluate whether Sealog can adapt to evolving log data within an online setting. Based on the results of RQ1, we select the most effective model (i.e., RobustLog and NeuralLog) as strong baselines for this RQ. To simulate a scenario where new log messages are received, we divide the BGL and Industry datasets into six even chunks (numbered from 0 to 5) in chronological order, each containing log messages with different templates. Note that the results of the Thunderbird dataset are similar to those of the BGL dataset, so it is excluded to save space. In practice, on-call engineers must verify suspicious logs reported

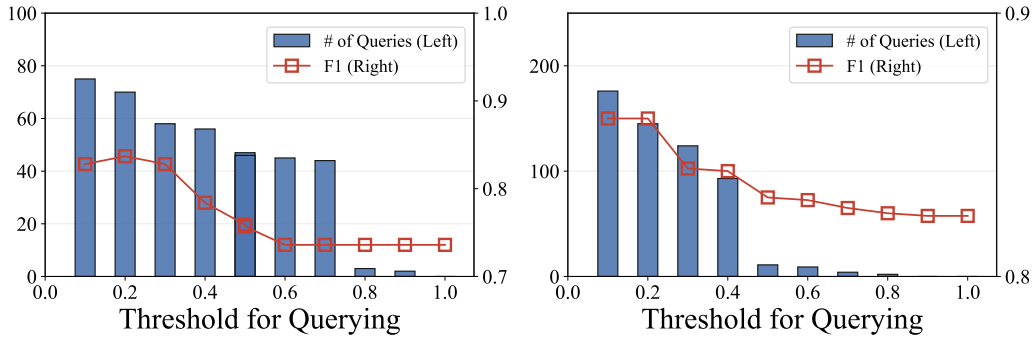


(a) BGL (b) Industry

Figure 5.5: Experimental Results of Online Anomaly Detection

by an anomaly detector, which naturally provides labeled data allowing us to continue training a model on the fly. To mimic this scenario, we train the models using one chunk and then evaluate the trained models on the next chunk (*e.g.*,, training with chunk 0 and testing with chunk 1), resulting in five evaluation results. While RobustLog and NeuralLog were not initially designed for receiving streaming feedback, we enable them to perform online training by continuously adapting to the incoming chunk, ensuring a fair comparison with Sealog.

Figure 5.5 presents the experimental results. Our observations are as follows: (1) Sealog consistently achieves the highest F1 scores across all chunks for both datasets, demonstrating its successful adaptation to new chunks and maintaining stable performance. (2) RobustLog's performance significantly declines from chunk 1 to chunk 4 in both datasets, indicating that it experiences catastrophic forgetting [37] when accommodating new chunks. Nevertheless, its performance improves in the 5th chunk on the BGL dataset, as chunks 4 and 5 share considerable data overlap, enabling RobustLog to detect the most recent anomalies. Conversely, the Industry dataset is more complex and has less overlap between chunks, resulting in substantially lower perfor-



(a) BGL (b) Industry

Figure 5.6: Anomaly Detection Performance w.r.t. the Number of Queries

mance for RobustLog compared to other models. (3) NeuralLog shows more consistent performance than RobustLog because of its transformer architecture, which has more parameters to remember observed data and address the issue of catastrophic forgetting.

Answer to RQ2. Sealog consistently surpasses the strong baseline methods (RobustLog and NeuralLog) under the setting that logs are evolving. The results demonstrate that Sealog can meet the requirements of being adaptive.

Impact of Query Number (RQ3)

In this RQ, we aim to study how the number of queries affects the effectiveness of Sealog. Specifically, we warm up the detection agent with only 50% of the training data for BGL and industry datasets, mimicking the scenario that the model might not be well trained initially. Then, we use the same test set as in RQ1 to evaluate the detection agent. While testing, we vary the number of queries sent to the backbone analyzer by tuning the threshold to issue a query, *i.e.*, θ_{query} in the range of [0, 1] with a step size of 0.1.

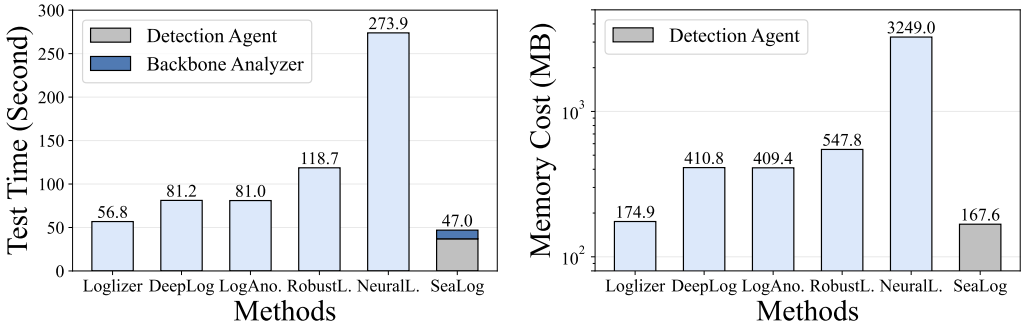
The results are presented in Figure 5.6. We can obtain the

following observations: (1) Fewer queries are launched when increasing the threshold θ_{query}. With fewer queries, the performance of Sealog is degraded. However, even if no query is launched (i.e., setting $\theta_{query} = 1$), the detection agent can still achieve around 0.73 F1 score for BGL and 0.82 for industry data, respectively. (2) To achieve the highest F1, the detection agent only queries the backbone analyzer fewer than 80 and 190 times on BGL and Industry datasets with millions of log lines, respectively. For the industry dataset, it indicates at most 6 queries per day can be raised. This shows that the detection agent can filter out most normal cases and involve the compute-intensive backbone analyzer conservatively.

Answer to RQ3. The performance of the detection agent improves as the amount of feedback increases. Owing to the NPT-based anomaly detection approach, most normal log messages are filtered out, resulting in only 190 queries in total (*i.e.*, 6 queries per day) for the industry dataset.

Time and Space Efficiency (RQ4)

(1) Time efficiency We measure the time required to perform anomaly detection on the complete BGL test set in comparison with all baseline methods. For DL-based methods, we set them to evaluation mode without backpropagation. Additionally, since online-generated logs are not pre-parsed, we include the time required to parse raw log messages using Drain [53] for baseline methods. Note that I/O time is excluded for all methods, and all methods are executed without GPU acceleration, which may not be available when resources are constrained. For Sealog, we account for the time taken by the detection agent and backbone analyzer. The detection agent's time includes all steps from preprocessing to anomaly detection. To simulate a real-world deployment scenario, all queries sent to the backbone analyzer are



(a) Time Efficiency Comparison (b) Space Efficiency Comparison

Figure 5.7: Experimental Results of Time and Space Efficiency Comparison

conducted asynchronously, allowing the execution of the detection agent to proceed without being blocked. The backbone analyzer's time is calculated as the total time needed to process all queries. Note that the communication time between the detection agent and the backbone analyzer is negligible due to the limited number of small-size queries launched.

Figure 5.7(a) presents the comparison results. Since LR, IF, and DT present similar efficiency, their values are averaged and denoted as Loglizer in the figure. We can observe that: (1) Sealog achieves comparable efficiency to Loglizer, being 2 to 5 times faster than other deep learning-based methods. While Sealog relies on a costly LLM as the backbone for accuracy, it also ensures efficiency because the LLM only needs to process a small number of queries from the detection agent. As shown in Figure 5.7(a), it takes around 10 seconds to process all the queries asynchronously. (2) DeepLog and LogAnomaly with similar LSTM-based architectures exhibit comparable performance. In contrast, RobustLog and NeuralLog require more time than others, as they involve a greater number of parameters.

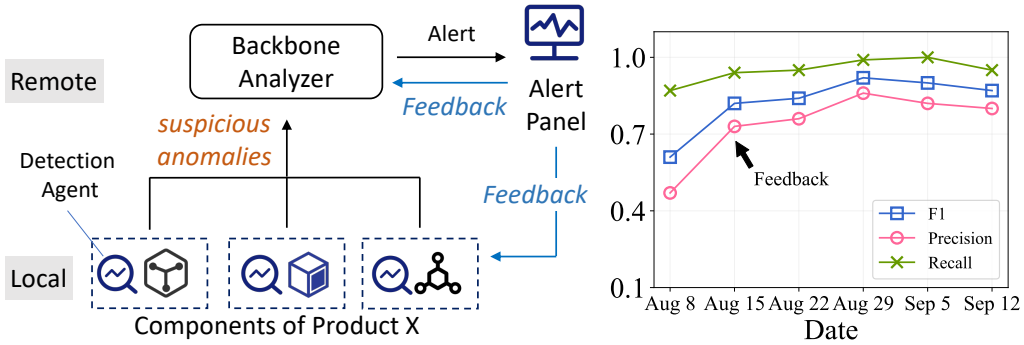
(2) Space efficiency We use Memory Profiler [36] to monitor the memory usage of each method when processing a single log window, as anomaly detection is usually performed on a window-by-

window basis. Figure 5.7(b) displays the comparison results. Sealog demands the least memory during log processing, using only 5% to 41% of the memory consumed by DL-based methods. This is primarily because Sealog only needs to store the NPT structure in memory, which is typically sparse. Loglizer demonstrates comparable space efficiency to Sealog, since it only stores the count vector of a log window, and the subsequent ML-based method (e.g., LR) is memory-efficient. DL-based methods require storing a large number of parameters (e.g., network parameters), resulting in higher memory usage. NeuralLog, which adopts a heavy transformer architecture, consumes the most memory among the methods.

Answer to RQ4. When compared to recent deep learning-based methods, Sealog is 2 to 5 times faster and requires only 5% to 41% of the memory consumed by the baselines. These results demonstrate that Sealog meets the requirement of being lightweight for practical anomaly detection.

5.5 Industrial Experience

We share our experience deploying Sealog in product X, a unified data center management. This platform offers multiple functionalities, including streamlined service orchestration, robust security measures, and access control mechanisms. Product X comprises multiple components, each consisting of *hundreds of microservices*. In the early stage of product X, we conduct log analysis in a centralized manner, where all log data are transmitted to a high-performance server for manual analysis or utilizing tools like log parser [181] to assist in the analysis process. This approach was suitable when the product had a smaller scale, generated a limited amount of log data, and had a smaller number of issues to address. However, with the deployment of the product and



(a) Deployment of Sealog in Product X

(b) Deploying Performance of Sealog

Figure 5.8: Practical Application of Sealog in Huawei Cloud

the increased customer number, the requirements for log analysis become more demanding (as described in Section 5.2.2).

To meet these practical requirements, we have adopted Sealog for monitoring product X. Currently, *135* microservices within product X have been equipped with Sealog over a period of *twelve months*. The deployment of Sealog in Product X is illustrated in Figure 5.8(a). The detection agent's time and space efficiency allows for local deployment alongside each microservice. Once a suspicious anomaly is detected, the agent forwards it to a centrally deployed backbone analyzer for detailed analysis. For confidentiality reasons, during the online deployment, we utilize an internal LLM maintained by Huawei Cloud. Finally, if the LLM agrees that these anomalies are true, these anomalies together with explanations are raised as alerts to a central alert panel, enabling on-site engineers to take prompt mitigation measures. After the on-site engineers confirm the reported anomalies, these anomalies serve as valuable feedback to continuously improve the overall framework. During the past twelve months, Sealog has resulted in less than 10% false positives, and the interpretable results have been well-recognized by the on-site engineers for convenient confirmation.

To assess the performance of Sealog in deployment, we gathered log data generated by these microservices over a six-week period (08/08/2023 to 12/09/2023). Then, we compared the prediction results of Sealog with manually annotated groundtruth provided by on-site engineers. The results are shown in Figure 5.8(b). In the first week, we solely deployed the detection agent, which resulted in a high recall but relatively low precision. This outcome was anticipated as we aimed to set a lenient threshold to capture most anomalies, regardless of the cost of increased false positive predictions. Starting from the second week, we deployed the feedback mechanism and the backbone analyzer. After that, the overall performance of Sealog improved, indicating the effectiveness of the feedback loop and the backbone analyzer.

Ideally, LLMs would provide direct feedback to the local agent and avoid the need for human intervention. However, even with advanced LLMs like ChatGPT, correct predictions cannot always be guaranteed. Therefore, we still involve on-site engineers for confirmation in our real-world deployment. However, with the assistance of the LLM-based backbone analyzer, the manual effort required for confirmation has been significantly reduced. This is particularly beneficial for newly employed engineers who may not be familiar with the system yet. Looking ahead, we believe that the proposed Sealog framework can benefit from advancements in LLM to automate the operation process of large-scale cloud systems.

5.6 Threats to Validity

We identify the following threats to the validity of our study.

The cloud system under study. Our motivation for a lightweight and adaptive method is driven by real-world scenarios in a single cloud system. However, Huawei Cloud is a representative world-leading cloud provider with a vast scale. Besides, our

findings in Section 5.2.2 are consistent with previous studies reported by other cloud vendors *e.g.*, Azure [146]. Additionally, we also adopt public datasets for evaluation. Hence, the evaluation results should be compelling.

The LLMs under study. We use LLM as a backbone analyzer in the Sealog framework. It is important to note that the performance and characteristics of LLMs can differ between implementations. We chose the most representative and publicly available LLM (GPT-3.5) to ensure reproducibility. However, Sealog is designed as a general framework, allowing LLMs to be replaced as necessary, thus enabling Sealog to benefit from advancements in LLM technology.

Implementation and parameter settings. We have implemented several measures to mitigate these threats. First, we have employed peer code review while implementing Sealog, and for the baseline methods, we have utilized open-sourced code released by the original paper or highly-rated replications on GitHub. Second, we have used the hyperparameters recommended by the original authors wherever available. Third, to ensure the reproducibility of our results, we have made our code and partial data publicly available.

5.7 Summary

In this chapter, we presented Sealog, a lightweight and adaptive log-based anomaly detection framework designed for practical cloud systems. Sealog utilizes a detection agent for lightweight anomaly detection in a streaming manner. We also incorporate large language models (LLMs) as a backbone analyzer, to provide more accurate and adaptive analysis. Experimental results on two public datasets and an industrial dataset from CloudX showed that Sealog is effective, achieving F1 scores between 0.949 and 0.993. Moreover, Sealog maintained high and consistent accuracy,

even in the presence of evolving log data *i.e.*, adaptive. Additionally, Sealog is 2× to 5× faster and requires only 5% to 41% memory resources compared to existing methods *i.e.*, lightweight. With the advantages of Sealog, we have deployed this framework in our production environment for twelve months. Our analysis shows that Sealog can continuously maintain a high performance (∼0.9 F1 score) over time. To benefit the community, we share our experience in deploying Sealog in Huawei Cloud and make Sealog publicly available.

□ **End of chapter.**

Chapter 6

Incident-aware Ticket Aggregation

Customers of cloud systems frequently submit support tickets to seek assistance from cloud providers when they encounter technical issues. To maintain user satisfaction, it is crucial for cloud providers to address these tickets promptly. However, during incidents, the overwhelming volume of customer tickets can make it challenging for support engineers to manage them effectively. A significant portion of these tickets may be duplicates, and traditional semantic similarity-based methods often fall short of identifying all duplicates due to the diversity and complexity of the issues. In this chapter, we introduce iPACK, a solution designed to aggregate duplicate tickets during incidents by leveraging cloud-side monitoring information (i.e., alerts). The remainder of this chapter is organized as follows. Section 6.1 provides the background of the problem and outlines our contributions. In Section 6.2, we present a motivating example to highlight the challenges and inspire the design of our methodology. Section 6.3 details the design of the proposed iPACK solution. Section 6.4 discusses the evaluation results of iPACK, based on real-world data collected from Azure. Following this, Section 6.5 showcases industrial case studies that demonstrate the practical applications of iPACK. Finally, Section 6.7 summarizes the chapter.

6.1 Problem and Contributions

In the era of Cloud Computing, cloud platforms such as Amazon AWS, Microsoft Azure, and Google Cloud Platform serve millions of users worldwide. When customers encounter a technical problem with the platform; they often resort to cloud providers for help by submitting a *support ticket* (ticket for short), which consists of a textual issue description and some basic attributes (e.g., date and product name). From the cloud provider's perspective, once a ticket is received, it is essential to provide timely assistance to customers to avoid user dissatisfaction and financial loss [6] [112].

In practice, *incidents* (i.e., unexpected service interruptions) are inevitable for large-scale cloud platforms [96] [30]. Though much effort has been devoted to ensuring the reliability of cloud systems [86] [174] [133], customers could still be impacted by incidents. For a large-scale cloud platform serving millions of customers, incidents could trigger a large number of tickets, among which many could be duplicate as the tickets are reported in a distributed and uncoordinated manner. To reduce the burden of support engineers, it is essential to *precisely and comprehensively aggregate the tickets, i.e., clustering the duplicate tickets caused by the same incident.* By doing this, the support team can resolve the tickets more efficiently.

To aggregate the tickets caused by the same incident, a common practice is to check if multiple tickets with similar symptom descriptions are reported within a short period. The intuition behind this is that customers using the same functionalities or services tend to encounter similar problems if they are caused by the same incident (e.g., service unavailability). Most existing studies on duplicate issue report detection measure the semantic similarity between two reports based on their textual descriptions, using natural language processing techniques such as word frequency [138] [137] [177], word embedding [166] [14], topic mod-

eling [15], and pretrained model [49]. Such semantic similarity-based approaches work well for traditional software systems (e.g., NetBeans [78], Eclipse and Firefox [77]). However, they are sub-optimal for aggregating duplicate tickets in cloud systems due to the large-scale and heterogeneous architecture of cloud systems [30] [164] [147]. The main reason is that customers of cloud systems could encounter distinct issues (with distinct symptoms) caused by the same incident. On the one hand, customers using the same service may experience different issues due to various usage scenarios. For example, when the control plane of the virtual machine (VM) service is problematic, the customers could complain about various problems related to VM creation, upgrade or deletion, depending on their particular scenarios. On the other hand, multiple services can be impacted by the same incident due to the notorious failure propagation problem [86] [147] [28] in cloud systems. For example, when an infrastructure-level service (e.g., a storage service) is interrupted, other services depending on it (e.g., VM and Web application) can be impacted too. As a result, customers using different services may observe different symptoms and submit tickets with dissimilar descriptions. Consequently, it is insufficient to tackle this problem by solely utilizing textual descriptions of tickets.

To address existing studies' limitations, we propose introducing cloud-side runtime information, i.e., *alerts*, to facilitate ticket aggregation in cloud systems. Modern cloud systems widely adopt monitors to continuously detect anomalies (unexpected behaviors) of cloud systems [4] [111] [46]. Once an anomaly is detected, an alert describing the anomaly will be fired to notify on-call engineers for inspection promptly. The services (and their internal components) are interdependent in cloud systems [164] [147]; therefore, when an incident impacts multiple components or services, multiple alerts will be triggered within a short period [174] [25], that is, these alerts are correlated with each other (i.e., **alert-**

alert relation). According to our study in Azure, the correlated alerts caused by most (93%) incidents are fired within four hours. On the other hand, a particular issue of a component (e.g., problematic API for VM allocation) in cloud systems can reflect a particular customer-side issue (e.g., cannot create a VM). So, it is possible to find a *responsible alert* within the component that captures the issue resulting in the ticket (i.e., **ticket-alert relation**). In Azure, we find that for 92% of customer tickets; the alert system has already fired responsible alerts that cause these tickets before the tickets are submitted.

Motivated by these two kinds of relations, we propose to formulate the ticket aggregation problem in cloud systems as a two-stage linking problem, i.e., alert-alert linking and ticket-alert linking. Intuitively, if the same incident triggers multiple interlinked alerts and these alerts are further linked to different tickets, then we consider these tickets should be aggregated (i.e., caused by the same incident). In doing this, it is possible to aggregate semantically different tickets via alert-alert links.

However, designing such a framework mainly faces two challenges originating from the large scale and complexity of cloud systems: First, alerts are massive and noisy. The main reason is that cloud systems consist of a large number of interdependent services. Each service adopts comprehensive monitors to capture any abnormal patterns to ensure its reliability [19]. These monitors could be sensitive. As a result, various alerts are continuously fired every second [86], so it is challenging to correctly identify and link alerts that are relevant to the ongoing incident. Second, the features of both alerts and tickets have high cardinality, which means each of their features consists of too many unique values. When considering linking alerts and tickets, the number of feature combinations grows exponentially due to the high cardinality. Consequently, it is hard to identify effective feature combinations between them and conduct correct correlation.

In this chapter, we propose iPACK to address these challenges. Specifically, iPACK mainly consists of three steps, i.e., *alert parsing*, *incident profiling* and *ticket-event correlation*. The first two steps address the first challenge, and the third step addresses the second challenge. In the *alert parsing* step, we preprocess (parse) alerts as more coarse-grained *events* to reduce redundant alerts. Next, in the *incident profiling* step, we propose GIP (graph-based incident profiling) to automatically filter noisy events and link events caused by the same incident. As a result, each incident is represented as an event graph by considering alert-alert relations. Afterward, in the *ticket-event correlation* step, we propose AIN (attentive interaction network) to correlate a ticket to a responsible event by considering ticket-alert relations. Finally, we aggregate these tickets that are linked to the events within the same event graph (i.e., incident), which are provided to CSS (customer support service) team to accelerate processing the tickets.

This work makes the following major contributions:

- We are the first to propose to introduce cloud runtime information (i.e., alerts) to aggregate duplicate tickets. We propose iPACK to leverage the alert-alert relations and ticket-alert relations to achieve this goal.

- We evaluate iPACK on three datasets collected from the production environment of Azure. The evaluation results show that iPACK outperforms state-of-the-art methods by 12.4%$\sim$31.2%, which confirm the effectiveness of iPACK. We also share our industrial experience of applying iPACK in a large-scale cloud platform, Azure.

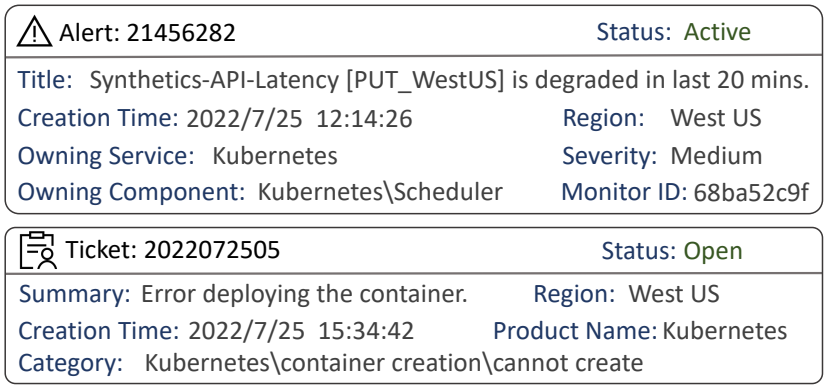


Figure 6.1: An Example of An Alert and Its Resultant Ticket

6.2 Motivation

6.2.1 Background

Alert Alerts are fired by monitors that continuously detect anomalies in cloud systems, which automatically notify on-call engineers for investigation [164] [165] [25]. An alert has many attributes as presented in Fig. 6.1 (top), including *alert ID*, *title*, *creation time*, *region*, *owning service*, *owning component*, *severity*, *monitor ID*, etc. The *title* is generated by following a template pre-defined by engineers. The *severity* indicates how serious the issue is, which has three levels, i.e., low, medium and high. A service (*owning service*) consists of many components (*owning component*), where each component has its own functionality or feature.

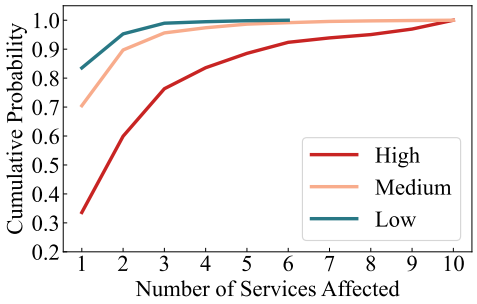
Alert-Ticket Linkage As presented in Fig. 6.1 (bottom), a support ticket usually contains attributes such as *ticket ID*, *creation time*, *summary*, *region*, *product name*, and *category*. The *summary* is free text written by customers in natural language. The *region* is where the customer's product is deployed. The *category* is a coarse-to-fine text description initially selected by the customer, which facilitates triaging a ticket to a proper support engineer. In addition, a ticket may also include a long detailed

description (hidden in the figure).

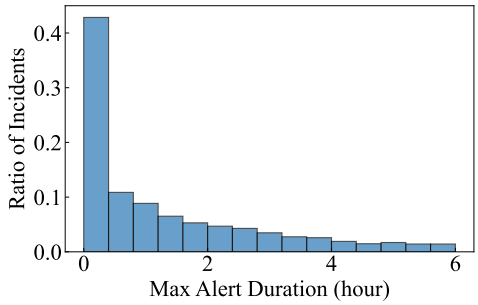
Modern cloud platforms adopt similar schemes of alerts and support tickets described above. For example, CloudWatch of AWS [4], Alerting of GCP [46] and Azure Monitor [111] share similar alerting mechanisms, and their alerts carry similar attributes. Besides, their ticket management systems require similar attributes from customers as in Fig. 6.1, i.e., AWS Support [5], Google Support Hub [45], and Azure Support [45]. In this work, we only leverage the *common* features that all these popular cloud platforms own to ensure generalizability.

6.2.2 Alert-Alert Relation

The alert-alert relation denotes that two alerts could be correlated if they are caused by the same incident. The relation originates from the hierarchical structures of modern cloud systems that consist of inter-dependent components or services [19]. When an incident happens, multiple components or services could be impacted due to failure propagation [147] [25], which will fire alerts within a short period associated with the same incident. During the diagnosis of an incident, in Azure, on-call engineers will manually mark these alerts and assess the severity of the incident according to the number of customers impacted. According to the diagnosis history in Azure from 2020/01/01 to 2022/06/01, as shown in Fig. 6.2(a), we found incidents with a higher severity tend to affect more services. Especially, 70% of high-severity incidents affect more than one service. We studied the resultant alerts of historical incidents. We calculated the max alert duration of the incidents (i.e., the time interval between the earliest and the latest alerts triggered by the incident). As shown in Fig. 6.2(b), we found that the max alert duration of 93% of incidents is within four hours. This serves as evidence to automatically identify the correlated alerts within an incident (in Section 6.3.3).



(a) The number of services impacted by incidents



(b) Distribution of max alert duration of incidents

Figure 6.2: Statistics of Alert and Incident Data in Azure

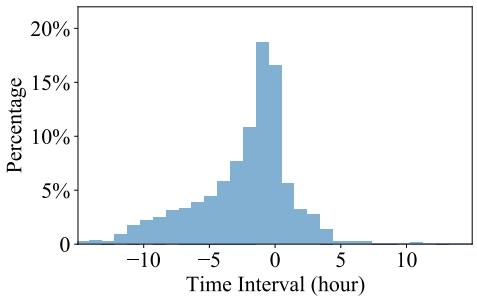


Figure 6.3: Time Interval between Alerts And Resultant Tickets

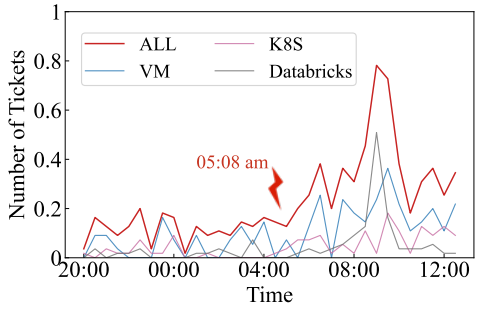


Figure 6.4: Ticket Number Trend during An Incident

6.2.3 Ticket-Alert Relation

The ticket-alert relation denotes that a ticket can correlate with a responsible alert inside the cloud systems. When a particular type of issue happens inside the cloud system (alerts are also fired), the customer could experience particular problems. Fig. 6.1 presents an example. If the API PUT (for container allocation) in the Kubernetes services is degraded, the customer can experience an error when deploying a container. In Azure, if a ticket is related to a cloud-side issue, the support engineers are required to annotate the responsible alert ID after diagnosis. Based on the annotated alert-ticket pairs collected from 2020/01/01 to 2022/06/01, we study the time interval between alert generation and ticket sub-

mission. Fig. 6.3 shows the results, where a negative time interval indicates that an alert is fired before the ticket is submitted. We found around 92% of tickets have responsible alerts fired before customers submit the tickets. This allows us to correlate responsible alerts for most tickets in runtime (in Section 6.3.4).

6.2.4 A Motivating Example

We present a real-world incident in July 2021 in Azure and its resultant tickets as a motivating example. The impact of the incident started at 05:08 AM (UTC). It was caused by the availability loss of the DiskRP (disk resource provider) service that provides a control plane service for managed disks. Since its gateway queue was full, a large proportion of incoming requests were rejected. As a consequence, services relying on DiskRP experienced interruptions. On-call engineers' diagnosis confirmed that three services were impacted, i.e., virtual machine (VM), Databricks, and Kubernetes (K8S). Customers using these services were affected, which led to overwhelming tickets. As shown in Fig. 6.4, the ticket numbers of the services simultaneously increased right after the impact started, which implies the three services could be impacted by the same incident concurrently. In particular, the CSS team received around *four* times the number of tickets than usual within a short period and assigned *twice* the number of support engineers to handle these tickets. We list some samples of alerts and tickets related to this incident in Table 6.1. These tickets ($t_1 \sim t_8$) carry dissimilar semantics due to different use scenarios and services for different customers. Therefore, it is hard to know that these tickets are actually caused by the same incident, rendering the difficulty for support engineers to group them and handle the burst of tickets efficiently.

We propose to aggregate these tickets by simultaneously leveraging the aforementioned alert-alert relations and ticket-alert rela-

Table 6.1: Alerts Caused by The Same Incident and The Resultant Tickets

Service	Tickets		Alerts	
	Category	Summary	Component	Title
VM	VM/Scale Update	t_1: Virtual machine scale sets resize issue.	Resource Provider	a_1 : VMStart Failures exceed 300 times.
	VM/VM Start	t_2: Server did not start on time.		
Databricks	Databricks/Job Issue	t_3: Unable to open cluster of Databricks.	Control Plane	a_2 : Databricks cluster creation fails.
	Databricks/Cluster Launch	t_4: Unable to provision clusters.		
K8S	K8S/Cluster Update	t_5: Unable to autoscale.	Resource	a_3 : The PUT operation success rate <80%.
	K8S/Cluster Update	t_6: Cannot upgrade node pool, stuck.	Scheduler	a_4 : CPU utilization exceeds 90%.

tions. We take Table 6.1 as an example to elaborate our intuition. First, we need to know what alerts are triggered by an incident, i.e., profiling the incident. In this example, we link the alerts $a_1 - a_2 - a_3$ via capturing the alert-alert relations (i.e., they are caused by the same incident). Second, we need to know what tickets are caused by these alerts, namely, linking $a_1 - (t_1, t_2)$, $a_2 - (t_3, t_4)$, and $a_3 - (t_5, t_6)$. Finally, because the alerts $a_1 \sim a_3$ are linked as an incident and $t_1 \sim t_6$ are further linked to these alerts, we can aggregate $t_1 \sim t_6$ as the same cluster even though they possess dissimilar semantics.

Challenges. To achieve this, iPACK should address the following two challenges originated from the large scale and complicated architecture of cloud systems [86] [147] [19].

Challenge 1: Massive and noisy alerts. Cloud systems could contain thousands of interdependent services. These services are closely monitored from various aspects to capture any unexpected behaviors. For example, there could be hundreds, even thousands of high-severity alerts reported in Azure per day. Some alerts are **regular alerts** that are reported frequently (due to sensitive monitoring rules) and periodically (due to periodical monitoring). These regular alerts are generally not related to a particular cloud incident and only report usual system runtime status such as CPU/memory usage rate (e.g., a_4 in Table 6.1). In contrast, **indicative alerts** are caused by an actual problem of cloud systems. For example, the alerts $a_1 \sim a_3$ in Table 6.1 are indicative alerts. It is challenging to identify the indicative alerts and

correctly link them among massive and noisy alerts.

Challenge 2: High feature cardinality. High feature cardinality refers to a situation where a feature has a large number of unique values. For example, the feature *category* of a ticket has more than 3,000 options, and the features *component* and *monitor ID* of alerts have more than 2,000 and 10,000 options, respectively. Using traditional one-hot encoding [89] methods to process these features would lead to a high-dimensional feature space, resulting in the curse of dimensionality [150]. Additionally, linking alerts to tickets requires the consideration of various combinations of features between them. However, due to the high feature cardinality, the number of possible combinations grows exponentially, making it difficult to identify the most effective combinations that accurately reflect the correlation between alerts and tickets. This constitutes a significant challenge in our work.

6.3 Methodology

6.3.1 Overview of iPACK

The goal of iPACK is to aggregate duplicate tickets that are caused by the same cloud incident among all tickets. Due to the large scale and heterogeneous architecture [30] [164] [147] of cloud systems, it is insufficient to solely consider the textual similarity of tickets to achieve this goal. To address this problem, we introduce cloud run-time information (i.e., alerts) and formulate it as a two-stage linking problem. Intuitively, iPACK first finds links between alerts by leveraging alert-alert relations. These inter-linked alerts constitute a graph to represent an incident. Then iPACK identifies the tickets that are caused by these alerts according to ticket-alert relations. The tickets linked to the alerts within the same graph (i.e., incident) are aggregated. Thus, we can aggregate the tickets with dissimilar semantics via the bridge of alert-alert links.

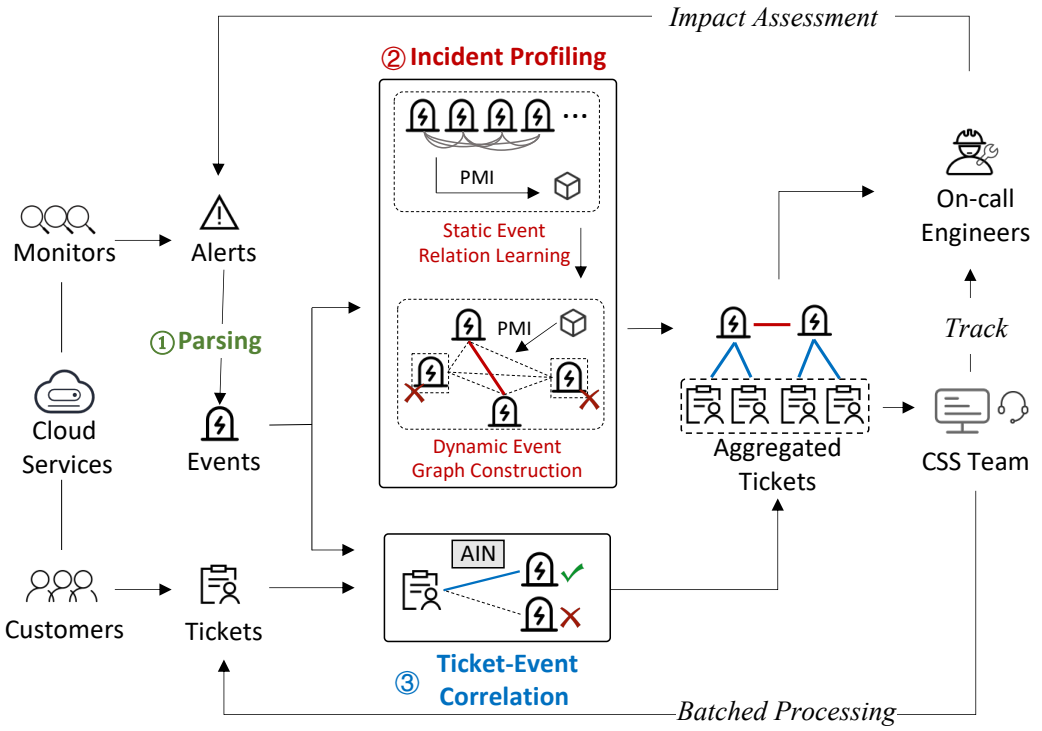


Figure 6.5: The Overall Framework of iPACK.

As shown in Fig. 6.5, iPACK consists of three steps: *alert parsing*, *incident profiling* and *ticket-event correlation*. In the *alert parsing* step, we parse alerts as more coarse-grained *events* to reduce redundant alerts. Next, in the *incident profiling step*, we propose a graph-based incident profiling (GIP) method to remove the regular events (i.e., parsed regular alerts) and link correlated indicative events. Then, in the *ticket-event correlation*, we propose an attentive interaction network (AIN) to correlate a ticket to an event. Finally, if two tickets are correlated to the events within the same event graph (i.e., the same incident), we aggregate the tickets as the same cluster. The results of the ticket aggregation are presented to the CSS (Customer Support Services) team to streamline the ticket processing process and improve efficiency. This allows support engineers to send out batch notifications to potentially affected customers and provide quick guidance for ser-

vice recovery. Additionally, the results can aid on-call engineers in conducting impact assessments, including identifying affected services and determining the extent of customer impact caused by the incident (e.g., number of affected customers).

6.3.2 Alert Parsing

The title of an alert is generated following an engineer-specified template. Monitors may be triggered multiple times during an incident causing massive redundant alerts. To reduce the volume of alerts and avoid redundancy, we parse each alert to its corresponding template and aggregate the alerts sharing the same template as an **event**. Take a_1 in Table 6.1 as an example; multiple similar alerts can fire concurrently such as "VMStart Failures exceed 100/150/200/250 times", which are aggregated as "VMStart Failures exceed $*$ times".

We formulate this problem as the well-studied log parsing problem [181] following [147]. We propose to customize a widely-adopted log parsing algorithm, Drain [53] to parse the alerts into templates (events). Drain works by extracting the common parts of alert titles from each group of alerts, where the group is determined by calculating the overlap of words. To enhance Drain in our scenario, we observe that if two alerts are reported by different monitors or belong to different components, the two alerts must have distinct templates. Therefore, we first divide all alerts into different partitions according to both *monitor ID* and *owning component*. We then apply Drain in each partition to extract the templates. In this way, we can reduce the noises in each partition and also accelerate the processing by parallel computing. Finally, each alert is parsed as an event, which introduces two features, i.e., *event template* and *event ID* (a hash value of its template). Within a fixed time window (Section 6.3.3), for events sharing the same template, we reserve the latest event and discard the rest of

the events to reduce its volume. The following steps are applied to events instead of raw alerts.

6.3.3 Incident Profiling

The goal of this step is to represent an incident via *linking the correlated events that are caused by the same incidents*. In doing this, the linked events can then be used to bridge semantically different tickets in the next step (Section 6.3.4).

To learn relations between events, some existing solutions leverage manual annotations [19] [23], which are not practical because such labels are hard to obtain and usually insufficient in real-world practice. While there are unsupervised solutions [174] [28], they require prior knowledge (e.g., the precise topology of cloud services) to estimate alert relations. However, such prior knowledge is usually inaccurate and requires extensive efforts to collect, update and validate [164] [28] [19].

We propose an unsupervised approach, i.e., Graph-based Incident Profiling (GIP), which does not rely on prior knowledge. The input is a series of events within a time window, and the output is one or multiple graphs of the events. Each graph profiles an incident containing indicative events related to the incident. GIP has a *static event relation learning* step and a *dynamic event graph construction* step. Intuitively, if two events are correlated, these events are more likely to be triggered within a short period frequently in the history [28] [19]. We model such frequent patterns in the first *static event relation learning* step. Then, in the *dynamic event graph construction* step, we dynamically link the events possessing the learned frequent patterns and remove regular events in the runtime.

Static Event Relation Learning

In this step, we assign a static score to each event pair weighing how likely they co-occur in history. To this end, we first collect a series of historical events in chronological order. Then we apply a *four-hour-long* sliding time window on these events with a step size of one hour. We adopt *four* hours as the window length because it can cover most alerts within an incident according to our study in Section 6.2.2. The one-hour step size allows us to introduce enough new events for learning the static event relations and avoid separating co-occurred events into two different windows. Each window w_i contain multiple events, i.e., $w_i = [e_1, e_2, e_3, ...]$. If two events appear in the same window, we count it as a co-occurrence. Based on these windows, we compute the point-wise mutual information (PMI) score [154] for each event pair, which is a popular metric to measure co-occurrence associations [126] [167]. Formally, the PMI value for the event pair (e_i, e_j) is :

$$PMI(e_i, e_j) = log\frac{p(e_i, e_j)}{p(e_i)p(e_j)}, \tag{6.1}$$

where $p(e_i, e_j) = \frac{C(e_i,e_j)}{M}$, $p(e_i) = \frac{C(e_i)}{M}$. $C(e_i, e_j)$ denotes the number of windows that contain both e_i and e_j, and $C(e_i)$ is the number of windows that contain e_i. M is the total number of windows. A higher PMI value indicates two events are more likely to co-occur in history, and a positive PMI value indicates they are more likely to co-occur than appear individually. We use $d(e_i, e_j)$ to denote the pre-computed PMI value for the event pair (e_i, e_j).

Dynamic Event Graph Construction

We then dynamically construct event graphs in the runtime by utilizing the learned static PMI values. The input to this step is the events collected within the latest four-hour-long time window. The output is one or more event graphs, each of which contains correlated events caused by the incident.

Algorithm 2: Dynamic Event Graph Construction

Input: Pre-computed PMI values in d, a window of latest events $w_j = [e_1, e_2, e_3, ...]$, hyper-parameter $\mu \in [0,1]$
Output: $g_o = \{g_1, g_2, ...\}$
Init: g ← Empty undirected graph; r ← Empty list

```
1  for i ← 1 to l do
2      for j ← i to l do
3          if d(e_i, e_j) > 0 then
4              g.AddWeightedEdge((e_i, e_j), weight=d(e_i, e_j))
5          end
6      end
7  end
8  for each node e_i ∈ g do
9      W = GetWeightsOfOutEdges(e_i)
10     AscendingSort(W)
11     γ = SearchKneePoint(W) // Kneedle algorithm
12     if γ < μ then
13         g.RemoveNode(e_i)
14     end
15 end
16 g_o ← GetSubGraphs(g)
```

Intuitively, we aim to link the events with high PMI values because they are possibly caused by the same ongoing incident in the runtime, considering they frequently co-occur in history. However, regular (noisy) events tend to co-occur with various types of events because they frequently appear regardless of whether there is an incident. In contrast, indicative events only frequently co-occur with only a small portion of events. Based on the difference between regular events and indicative events, we propose a novel algorithm to prune the regular events automatically, and the remaining indicative events are correlated. The pseudocode of the algorithm is shown in Algorithm 2. First, we link every pair of events with positive PMI values constituting a single initial event graph g with the PMI values as weights of edges (line $1 \sim 7$). Then, for each node, we calculate a knee point (i.e., γ in Algorithm 2) based on the PMI values of all its out edges, i.e., $\mathcal{W}$ (line

$9 \sim 11$). Specifically, we adopt the Kneedle algorithm [131] to calculate γ. A small γ for a node denotes that most PMI values of its linked neighbors are large, namely, the node frequently co-occurs with many neighbors (events). This implies that the node is more likely to be a regular event. Therefore, we remove the node if its γ is less than a threshold μ (line $12 \sim 14$). As revealed by previous studies [174] [20], regular events make up a large portion of all events. Therefore, we empirically set $\mu = 0.8$ to remove most events aggressively, which turns out to be effective in our scenario (Section 6.4.2). Finally, we extract subgraphs (i.e., connected component [149]) from the the pruned graph g (line 16).

6.3.4 Ticket-Event Correlation

After profiling incidents as several event graphs (i.e., event-event linking), we correlate each ticket to the event that captures the internal cloud issue resulting in the ticket (i.e., event-ticket linking). If two tickets are correlated to inter-linked events (i.e., they are caused by the same incident), we can then aggregate them as the same cluster.

We mainly address the challenge caused by the high cardinality of features of tickets and events (Section 6.2.4). Inspired by factorization machine [130] in the field of recommendation systems, we propose an attentive interaction network (AIN), which decomposes feature combinations as Hadamard products of low-dimension feature embeddings. In this way, we bypass directly encoding the exponentially-growing feature combinations with high-dimension feature vectors. The input of AIN is a ticket-event pair and the output is a probability representing how likely the input pair is correlated. Fig. 6.6 shows the overall framework of AIN composed of three layers, i.e., *embedding layer*, *attentive interaction layer*, and *prediction layer*, which are elaborated as follows.

Embedding Layer. The embedding layer represents all features (f_i for a ticket feature and $\hat{f}_i$ for an event feature) as trainable vectors (i.e., embeddings) denoted as $\mathbf{v_i} \in \mathbb{R}^k$, where k is a user-defined hyper-parameter. For *summary* of tickets and *event template* of events (denoted as f_1 and $\hat{f}_1$ in Fig. 6.6), we resort to the power of pretrained model BERT (Bidirectional Encoder Representations from Transformers) [33] to embed their semantics as vectors. We exclude the detailed ticket description since it potentially introduces noises, and the summary already provides the essential part [166] [49] [145]. The remaining features are initialized as random vectors.

Attentive Interaction Layer. After each feature is associated with an embedding vector, the attentive interaction layer models the feature combination of two features as the Hadamard product (i.e., element-wise product denoted as $\odot$) of their corresponding embedding vectors. For $\mathbf{u} = \mathbf{x} \odot \mathbf{y}$ we have $u_i = x_i y_i$. The attentive interaction layer models combinations of features across a ticket and an event, formally,

$$\mathbf{z} = \sum_{i}^{n} \sum_{j}^{m} a_{ij}(\mathbf{v}_i \odot \mathbf{v}_j), \tag{6.2}$$

where n and m are the numbers of ticket and event features, respectively. To identify the effective feature combinations for different ticket-event pairs, AIN computes an importance score a_{ij} for each combination result $v_i \odot v_j$ in Equation (6.2). Afterwards, these feature combinations are summarized as a single representation $\mathbf{z} \in \mathbb{R}^k$ by computing their weighted average. The importance weight a_{ij} is calculated as follows:

$$\hat{a}_{ij} = \mathbf{h}^T \phi(\mathbf{W}(\mathbf{v}_i \odot \mathbf{v}_j) + \mathbf{b}), \tag{6.3}$$

$$a_{ij} = \frac{e^{\hat{a}_{ij}}}{\sum_i^n \sum_j^m e^{\hat{a}_{ij}}}, \tag{6.4}$$

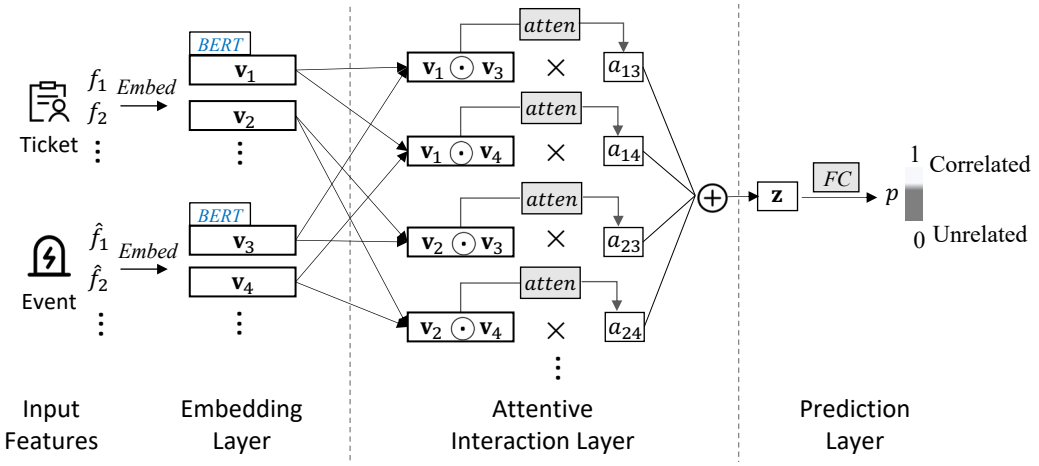


Figure 6.6: The Overall Framework of AIN.

Equation (6.3) denotes a fully-connected (FC) neural network that takes the combination of two features as input and outputs their (unnormalized) importance weight. where $\phi(x) = max(0, x)$ is the ReLU activation function. $\mathbf{h}^T \in \mathbb{R}^r$, $\mathbf{W} \in \mathbb{R}^{(r \times k)}$ and $\mathbf{b} \in \mathbb{R}^r$ are trainable parameters. r is a hyper-parameter that denotes the size of the hidden layer. Equation (6.4) normalizes the importance weights to $[0, 1]$. The importance weights control how much each feature combination contributes for prediction. For example, in Equation (6.2), for a_{ij} close to 1, its corresponding feature combination will dominate the summarized vector $\mathbf{z}$. This means that the prediction mostly depends on the feature combination of $\mathbf{v_i}$ and $\mathbf{v_j}$. In addition, the weights are automatically learned by the FC in Equation 6.3, we actually force AIN to select the effective feature combinations when learning from the data.

Prediction Layer. We formulate ticket-event correlation as a binary classification problem. Particularly, to calculate the correlation probability p, an FC neural network is applied on $\mathbf{z}$, i.e., $p = \sigma(\mathbf{w}_o^T \mathbf{z} + b_o)$, where $\mathbf{w}_o \in \mathbb{R}^k$ and $b_o \in \mathbb{R}$ are trainable parameters, and $\sigma(x) = \frac{1}{1+e^{-x}}$ is the Sigmoid function producing a probability within the range of $[0, 1]$. To update all trainable parameters, we utilize the popular Adam optimizer [75] to minimize

the following binary cross-entropy loss $\mathcal{L}_{BCE}$ via fitting training data with N ticket-event pairs.

$$\mathcal{L}_{BCE} = -\sum_{i}^{N} \big(y_i log(p_i) + (1 - y_i) log(1 - p_i)\big), \tag{6.5}$$

where $y_i = 1$ for positive (i.e., correlated) ticket-event pairs and $y_i = 0$ for negative (i.e., unrelated) pairs. The positive samples are collected by extracting the responsible alert ID of a ticket from its resolution text written by support engineers (Section 6.2.3). So, such data is gradually accumulated during the daily work routine of support engineers, which does not incur additional manual effort for data labeling. We then randomly sample the same number of negative pairs. The features used are *event template*, *event ID*, *severity*, *monitor ID*, *owning service*, *owning component* for events, and *product name*, *category*, *summary* for tickets.

6.3.5 Deployment

iPACK consists of offline parts (pre-computed) and online parts (serving online continuously) for deployment in the could systems. The offline parts include alert parsing, static event relations learning and AIN training. The online parts conduct alert parsing, dynamic event graph construction, and ticket-event correlation utilizing the trained AIN. The details are as follows.

Offline Parts

Intuitively, the offline parts leverage the historical data to prepare intermediate data (e.g., PMI values) or model (e.g., AIN) for online use. Specifically, iPACK parse all collected alerts to events (Section 6.3.2). Then, static event relations learning is conducted (Section 6.3.3), which computes PMI values for all event pairs. The PMI values are then stored in a Redis database for reference. After that, AIN is trained using historical ticket-event

pairs. Azure continuously collects the alert and ticket data; in order to capture the latest system update (e.g., new alerts), the offline parts are executed periodically (e.g., once every month).

Online Parts

In the online deployment, iPACK is periodically executed (e.g., every five minutes) and pushes its latest analysis results to the CSS team. Support engineers can also manually trigger iPACK when needed (e.g., a large volume of tickets are received). Considering cloud services and customers are physically isolated in different regions, iPACK is applied separately in different regions. Once executed, iPACK collects the latest alerts and tickets within the latest four-hour-long time window to analyze. We can reduce the great volume of ticket-event pairs by filtering with region and time. The tickets and alerts in the same time window and region constitute a *chunk*.

In each chunk, after parsing alerts as events, GIP is applied to link events as event graphs (i.e., incidents). Then, we apply AIN to link each ticket to one of the events. For each ticket, AIN recommends a list of events ranked by the associated correlation probabilities. Note that we exclude the tickets whose largest probability in the ranked list is smaller than a confidence threshold $\theta = 0.8$, because they are more likely caused by a customer-side issue (e.g., incorrect configurations). Next, tickets that are correlated to the events within the same event graph are aggregated as a cluster. Based on the aggregation results, on the one hand, on-call engineers can conduct impact assessment (i.e., how many customers are impacted) for an incident; on the other hand, the CSS team can avoid duplicate manual inspection and make batched communication to customers. (e.g., provide the latest mitigation progress of the internal incident).

6.4 Evaluation

We answer the following research questions (RQs) to evaluate the performance of iPACK:

- **RQ1**: How effective is iPACK in aggregating duplicate tickets caused by the same incident?
- **RQ2**: How effective is AIN in correlating a ticket to the responsible event?
- **RQ3**: How does graph-based incident profiling (GIP) impact the effectiveness of iPACK?

6.4.1 Experimental Setting

Dataset

We collect the datasets from the production environment of Azure from 2020/01/01 to 2022/06/01. To evaluate the generality of iPACK, we collect three datasets from different physically isolated regions (i.e., $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$), which cover 81 services serving different numbers of customers. Each dataset is collected from tens of services and includes hundreds of incidents and hundreds of thousands of alerts. For each incident, the datasets contain tens of to hundreds of resulting tickets. Note that we hide the specific figures of the dataset statistics due to the confidential policy of Azure. We use the data before 2022/01/01 to compute PMI values (Section 6.3.3) and train AIN (Section 6.3.4). The data after the date is used for evaluation.

Comparative solutions

Recent studies have been working on user feedback analysis such as duplicate bug report detection [145] [119] [178] [15] [17] and emerging issue detection [38] [177] [39]. We select the following state-of-the-art approaches as our comparative solutions:

Table 6.2: Effectiveness of Aggregating Duplicate Tickets Caused by the Same Cloud Incident

Methods	Dataset $\mathcal{A}$			Dataset $\mathcal{B}$			Dataset $\mathcal{C}$		
	Precision	Recall	**F1 score**	Precision	Recall	**F1 score**	Precision	Recall	**F1 score**
Categorization	0.930	0.205	0.336	0.943	0.373	0.535	0.925	0.207	0.338
iFeedback	0.901	0.590	<u>0.713</u>	0.876	0.473	0.614	0.886	0.626	0.733
LWE	0.862	0.453	0.594	0.824	0.515	0.634	0.861	0.672	<u>0.755</u>
BERT	0.884	0.587	0.705	0.854	0.710	<u>0.775</u>	0.843	0.629	0.720
LinkCM	0.931	0.507	0.657	0.892	0.538	0.671	0.901	0.628	0.740
LinkCM w/ GIP	0.900	0.685	0.778	0.886	0.756	0.816	0.899	0.809	0.852
iPACK	0.912	0.960	**0.935**	0.882	0.861	**0.871**	0.899	0.888	**0.894**

Categorization. We aggregate tickets by referring to their feature *category*, i.e., if two tickets share the same category, then they are aggregated into the same cluster.

iFeedback. iFeedback is proposed and adopted by WeChat in their production environment [177], which targets aggregating similar user feedback by identifying frequent word combinations (and groups of combinations). For example, if the word combination of "pay" and "fail" bursts, an issue may happen to the payment feature of the product.

LWE. LWE [15] is a method integrating Latent Dirichlet Allocation (LDA) and word embeddings to leverage the advantages of both techniques. LWE first utilizes LDA to represent all tickets and roughly identify candidates of duplicated tickets. Then, the candidates are represented using word embeddings to conduct more fine-grained clustering.

BERT. BERT [33] is a popular pretraining model in natural language processing and has shown its power in capturing the semantics of user feedback in recent studies [49] [97] [155]. Because these studies do not directly aggregate user feedback, in this work, we adopt BERT to first represent the tickets as dense vectors, based on which we use agglomerative hierarchical clustering [151] to aggregate tickets.

LinkCM. LinkCM [47] is proposed to facilitate the triage of a customer-reported alert by matching it with an alert of cloud

systems. LinkCM learns the correlation by purely fusing the titles between the report and alert via a decomposable attention mechanism and transfer learning. In our scenario, if two tickets are correlated to the same event by LinkCM, they are grouped together. LinkCM can also link a ticket to an event as AIN does, so we combine GIP with LinkCM (i.e., **LinkCM w/ GIP**) as a strong baseline for comparison.

Implementation Details

We have implemented iPACK with approximately 3000 lines of Python code and packaged it as a serverless function [164] for ease of use in Azure. The iPACK system is deployed on a CentOS Linux server with 60GB of RAM and an Intel(R) Core(TM) i7-5930K CPU @ 3.50GHz. The AIN component of iPACK is trained and tested with the GPU acceleration of an NVIDIA GeForce GTX TITAN X. We have set the default hyper-parameters of the AIN as k=128 and r=256, and the model is trained until its training loss stops decreasing for ten continuous epochs, using an early stopping approach. As for the comparative solutions, as they are not open-sourced, we have followed the implementation in their respective papers and leveraged well-established libraries to ensure accuracy. For example, we have used AllenNLP [3] for LinkCM, scikit-learn [132] and gensim [127] for LWE, and HuggingFace [59] for BERT.

Evaluation Metrics

Metrics for evaluating ticket aggregation (RQ1 and RQ3). Given a sequence of tickets, our approach assigns a unique cluster ID, denoted as "incident-number" to tickets that are caused by the same incident. Tickets that are not related to a cloud-side issue are marked with the cluster ID "non-incident". To evaluate the accuracy of our ticket aggregation, we use the widely accepted

Rand Index [1,41,128] for pair-wise comparison in clustering. We conduct pair-wise comparisons between the ground-truth cluster ID and the predicted cluster ID for all tickets. The results are used to calculate the following metrics: True Positives (TP), which are pairs of duplicate tickets correctly predicted to have the same cluster label; True Negatives (TN), which are pairs of non-duplicate tickets correctly predicted to have different cluster labels; False Positives (FP), which are pairs of non-duplicate tickets wrongly predicted to have the same cluster label; and False Negatives (FN), which are pairs of duplicate tickets wrongly predicted to have different cluster labels. Based on the results, we use the following metrics to evaluate the aggregation results: $precision = \frac{TP}{TP+FP}$, $recall = \frac{TP}{TP+FN}$, and $F1\ score = 2 \cdot \frac{precision \cdot recall}{precision\ +\ recall}$.

Metrics for evaluating ticket-event correlation (RQ2). The correlation of tickets with an event, referred to as AIN in Section 6.3.4, is a crucial component of iPACK. This component generates a ranked list of potential responsible events for a given ticket based on the probability scores (as determined by AIN's output) in descending order. To assess the accuracy of this step, we use the metric Acc@K (accuracy@K). For each ticket, if the actual ground-truth event appears within the top-K positions of the list, we consider the ticket to be a "hit". The Acc@K metric is calculated as the ratio of the number of hit tickets to the total number of tickets, represented as $Acc@K = \frac{\#\ of\ hit\ tickets}{\#\ of\ all\ tickets}$. In our evaluation, we consider three values of K (i.e., 1, 2, and 3) and also compute the average of these three metrics to provide a comprehensive assessment.

6.4.2 Experimental Results

The Effectiveness of iPACK (RQ1)

In this RQ, we aim to evaluate how accurately iPACK can aggregate the duplicate tickets by comparing it with all comparative so-

lutions (Section 6.4.1). The evaluation is conducted using datasets $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ and the results are reported in terms of precision, recall, and F1 score. Precision reflects the degree of correctness in the clustering results, while recall represents the completeness of the results. The F1 score is a balance between precision and recall and provides a comprehensive measure of the overall performance of the approach. The results are presented in Table 6.2. The highest F1 score is emphasized in **bold**, and the second-best score is underlined.

We can make the following observations: (1) iPACK achieves the best F1 score across all three datasets, i.e., 0.935, 0.871, and 0.894, outperforming the second-best methods by 31.2%, 12.4% and 18.4% in dataset $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, respectively. (2) Categorization can achieve the highest precision (0.930∼0.943) although its recall is considerably low (0.205∼0.373). The reason is that the ticket feature *category* is defined in a fine-grained manner by support engineers in Azure. Therefore, it tends to aggressively split the complete set of duplicate tickets into many small groups, leading to a low recall score. However, tickets in each such small group share precisely similar semantics as evidenced by the high precision. (3) iFeedback, LWE, and BERT show lower precision but higher recall than Categorization. The reason is that these methods can capture more coarse-grained semantic similarity between tickets. Consequently, they can generate larger clusters (higher recall) but introduce additional noises (lower precision) (4) LinkCM can achieve a higher precision among all baseline methods except Categorization. Moreover, after combining with GIP, LinkCM w/ GIP can increase its recall because more tickets are aggregated together through event-event linking. However, it still underperforms iPACK in terms of the overall F1 score because LinkCM cannot correlate a ticket to an event as accurately as iPACK does (will show in RQ2). For instance, LinkCM may associate a cluster of similar tickets with the wrong event. Therefore, even though

related events are linked together, similar tickets are separated into different clusters, resulting in high precision but low recall.

Answer to RQ1. iPACK achieves the best F1 score among all state-of-the-art baselines across three datasets collected from different regions. iPACK slightly sacrifices precision compared with the Categorization method but achieves the highest F1 score 0.871∼0.935, outperforming state-of-the-art methods by 12.4%∼31.2%.

The Effectiveness of Ticket-event Correlation (RQ2)

In this RQ, the focus is on evaluating the accuracy of the ticket-event correlation step of iPACK, i.e., the proposed attentive interaction Network (AIN). The performance of AIN is compared with LinkCM [47] and four popular machine learning algorithms: LR (logistic regression), SVM (support vector machine), RF (random forest), and LightGBM (light gradient boosting machine). Additionally, the contribution of the attentive feature interaction component to AIN is studied.

To ensure a fair comparison, categorical features are represented as one-hot vectors, which are then concatenated with the representation of textual features extracted using BERT. This allows for a consistent input feature representation for all models compared. A variant of AIN is also developed by removing its attentive feature interaction component (referred to as "AIN w/o atten." in Table 6.3). This variant instead concatenates all feature embeddings into a single feature vector as the input for the prediction layer, as illustrated in Fig. 6.6. For clarity, this experiment is conducted using all pairs of ticket-event data from datasets $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$. We compare AIN with the baselines and its variant in terms of Acc@1, Acc@2, Acc@3 and the average of these metrics.

We can make the following observations in the results shown in Table 6.3: (1) The proposed AIN model outperforms all base-

Table 6.3: Effectiveness of Correlating a Ticket to An Event

Models	Acc@1	Acc@2	Acc@3	Average
LR	0.519	0.657	0.733	0.636
SVM	0.332	0.409	0.493	0.411
RF	0.563	0.684	0.761	0.669
LightGBM	0.658	0.723	0.832	0.712
LinkCM	0.743	0.769	0.882	0.798
AIN w/o atten.	0.673	0.762	0.824	0.753
AIN	**0.817**	**0.907**	**0.936**	**0.887**
Δ(%)	+21.4%	+19.0%	+13.6%	+17.8%

line models in terms of all four evaluation metrics. Notably, AIN achieves the highest Acc@1 score of 0.817, indicating its superior ability in accurately linking tickets to events and facilitating more effective ticket aggregation. (2) The introduction of the attentive feature interaction component results in significant improvements in AIN's performance, with a 21.4% increase in Acc@1 and a 17.8% increase in the average accuracy. This demonstrates that the component plays a crucial role in identifying effective feature combinations for accurate ticket-event linking. (3) Interestingly, AIN w/o atten. underperforms LinkCM and achieves similar performance as LightGBM. The reason is that AIN w/o atten. adopts simple concatenation of feature embedding, which fails to capture effective feature combinations. (4) LinkCM can outperform other baseline methods since its decomposable attention mechanism is able to capture the semantic matching between tickets and events. On the other hand, the relatively low Acc@1 scores of LR, SVM, RF and LightGBM may be due to the sparsity and high dimensionality of the input features. However, RF and LightGBM exhibit improved accuracy over LR and SVM, as they alleviate these problems through feature selection.

Answer to RQ2. AIN outperforms all other baseline methods by a large margin in correlating a ticket to the event that causes

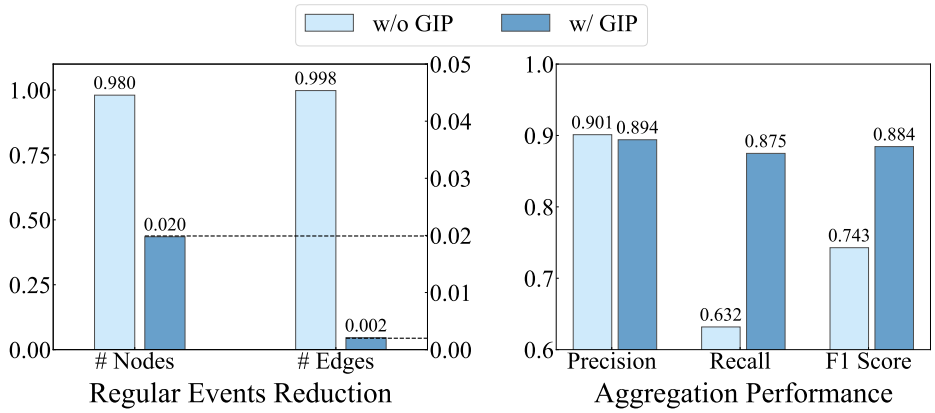


Figure 6.7: The Effectiveness of Graph-based Profiling (GIP)

it. The proposed attentive feature combination is the key to achieve the performance, which improves the average accuracy of AIN by 17.8%.

The impact of graph-based incident profiling (GIP) (RQ3)

We propose GIP to reduce regular events (noisy events) and link correlated indicative events to profile an incident, which bridges the tickets linked to the events even though they are semantically different. We evaluate its impact on iPACK using the union of all three datasets as in RQ2. We conduct the evaluation from the following two aspects:

(1) *The ratio of events reduced.* GIP builds a fully-connected event graph (link every two events with positive PMI values) and then prunes this graph via Algorithm 2. We measure the effectiveness of GIP with the ratio of nodes and edges that are pruned (reduced). Fig. 6.7 (left) presents the ratio of nodes and edges in the event graph without or with GIP (we normalize the ratio for better presentation). We can observe that only 2% of nodes and 0.2% of edges remain after using GIP, which shows GIP can reduce the large volume of events effectively.

(2) *The impact on the overall performance in aggregating du-*

plicate tickets. Though GIP can reduce the number of events, we aim to further evaluate whether it can accurately remove the regular events and link the correlated events as expected. To achieve this, we compare the ticket aggregation performance of iPACK with or without GIP. After removing GIP, we regard those tickets linked to the same event by AIN as belonging to the same cluster. The results are shown in Fig. 6.7 (right). We can observe that after applying GIP, its precision drops slightly, but the recall is largely improved. As a result, the overall F1 score is improved by 18.9%, from 0.743 to 0.884. This indicates that only a small portion of events are not correctly linked; however, more duplicate tickets are accurately aggregated via event-event linking.

Answer to RQ3. GIP can greatly boost the overall performance of iPACK. On the one hand, GIP reserves only 2% nodes and 0.2% edges in the pruned event graph. On the other hand, GIP accurately reserves and links the indicative events and improves the F1 score from 0.743 to 0.884.

6.5 Industrial Experience

In this section, we share our industrial experience by presenting a success case and a failure case from the real-world deployment of iPACK in Azure.

A Success Case

In September 2021, a datacenter maintenance activity resulted in the accidental shutdown of a water tower pump, which is a critical component of cooling systems. To prevent overheating and potential damage to users' data, the maintenance personnel had to shut down the downstream storage hardware. This caused a storage service disruption, leading to cascading impacts on several

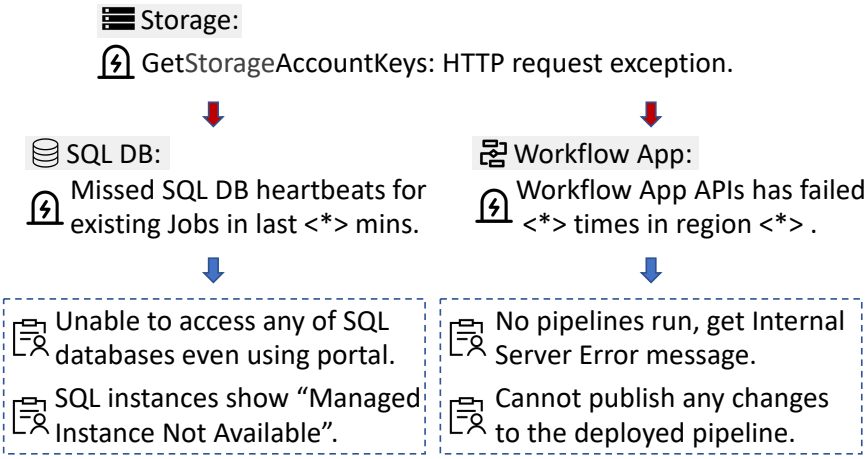


Figure 6.8: A Success Case of iPACK in Azure

dependent services such as the SQL DB and Workflow App, and triggering alerts.

The CSS team received a substantial number of tickets describing a wide range of issues in response to these events. To assist with the situation, iPACK continuously collected and analyzed the generated alerts and tickets. The partial output of iPACK's analysis is presented in Fig. 6.8. iPACK successfully linked the storage alert with corresponding alerts from SQL DB and Workflow App, as demonstrated by the **red** arrows in Fig. 6.8. Additionally, the tickets caused by these events were linked to their respective root cause events, as depicted by the **blue** arrows. This allowed the tickets to be aggregated, despite their semantic differences, and the results were pushed to the support engineers. With the information provided by iPACK, support engineers were able to initiate batch communications with potentially impacted customers and avoid duplicative manual inspections. Throughout the resolution process, the customers were continuously informed of the mitigation progress of the incident.

A Failure Case

iPACK could sometimes fail when it cannot find responsible alerts in the cloud systems for a ticket. In August 2021, the CSS team received multiple tickets complaining of 503 (service unavailable) errors when the customers were using Web Services. Though the tickets were suspected to be caused by an internal issue due to their similar symptoms, iPACK did not correlate them with any alert. Only around five hours after the first ticket had been received, a related alert was fired and correlated by iPACK. According to the after-the-fact analysis of on-call engineers, the root cause of this incident turned out to be bad configurations of a Canary (gray) release for a few tenants. The developers did not configure a specific monitor for each of the tenants but monitored all tenants as a whole. As a result, the monitor was not sensitive enough and only triggered when most of the tenants' requests failed. Nevertheless, iPACK continuously runs and could still correlate the alert with the resultant tickets after the alert was finally fired. In this way, iPACK can potentially discover such under-monitoring cases and guide the configuration of monitors to improve system reliability [88]. Fortunately, such cases (tickets submissions before alerts) are rare in Azure with comprehensive monitoring according to our study (Section 6.2.3).

6.6 Threats to Validity

External Validity. The study's object is the primary external threat. The data was collected from Azure, as there is no publicly available dataset containing customer tickets and a large number of alerts. However, Azure is a world-leading cloud provider with a vast scale. The data covers a broad range of services from various regions (Section 6.4.1). Hence, the evaluation in Azure should be representative and convincing. Furthermore, iPACK leverages

the common features provided by the most popular cloud vendors, making it capable of generalizing to similar cloud systems, potentially benefiting cloud customers globally.

Internal Validity. Implementation and parameter setting are key internal threats to validity. As the baseline approaches are not open-sourced, we re-implemented them closely following the original papers, using mature libraries for core algorithms (Section 6.4.1). Both proposed and baseline methods underwent peer code review. For parameter setting, we conducted a grid-search to select the best results.

6.7 Summary

This chapter tackles the problem of aggregating duplicate customer support tickets for cloud systems. Previous solutions that mainly rely on customer-side information (i.e., textual similarity between tickets) are sub-optimal for tickets of large-scale cloud systems. The main cause is the complexity of cloud systems that consist of many inter-dependent services, where the customers may experience distinct issues even though they are affected by the same incident. To overcome this limitation, we propose iPACK to leverage alerts of cloud systems to facilitate ticket aggregation. Specifically, we propose graph-based incident profiling (GIP) to model alert-alert relations and attentive interaction network (AIN) to model alert-ticket relations, respectively. In this way, we can aggregate the tickets that are linked to the same incident (linked alerts) even though they carry dissimilar semantics. We evaluate iPACK based on three datasets collected from the real-world production environment in a large-scale cloud vendor, Azure. iPACK outperforms state-of-the-art methods by 12.4%∼31.2% across the three datasets.

□ **End of chapter.**

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Cloud systems have become immensely popular, significantly supporting our daily activities by providing scalable and accessible computing resources. As a result, ensuring their reliability is of paramount importance to maintain seamless and uninterrupted services. However, achieving high reliability in cloud systems is a formidable challenge due to their vast scale and inherent complexity. This thesis delves into our research efforts aimed at addressing these challenges, exploring innovative approaches and solutions to enhance the reliability of cloud systems in the face of their ever-growing demands and intricate architectures.

In chapter 4, we address the critical challenge of improving the observability of cloud systems, which is complicated by the virtualization techniques that obscure insights into system operations. Our study on Huawei Cloud reveals that instances with similar functionalities exhibit distinctive communication and resource usage patterns, enabling their identification despite limited access to tenant data. To leverage these patterns, we propose Prism, a framework that clusters instances based on their communication and resource usage data through a coarse-to-fine strategy. Extensive experiments demonstrate Prism's superior performance and scalability, achieving high accuracy in clustering instances. Prism

has been successfully validated in Huawei Cloud, where it has enhanced system observability and reliability by identifying vulnerable application deployments and aggregating minor errors to detect latent issues.

In chapter 5, we identify the limitations of existing log-based anomaly detection methods, which often struggle with resource constraints and adapting to evolving log data. Our study based on log data collected from Huawei Cloud highlights these issues, revealing that traditional methods are either too resource-intensive or insufficiently adaptive. To overcome these challenges, we propose SeaLog, a novel framework that integrates the efficiency of traditional machine-learning approaches with the adaptive capabilities of large language models. SeaLog consists of a lightweight detection agent and a backbone analyzer, which together provide accurate, resource-efficient, and adaptive anomaly detection. Our extensive experiments demonstrate SeaLog's superior performance in both fixed and evolving log data scenarios, and its successful deployment in Huawei Cloud over twelve months confirms its practical effectiveness.

In chapter 6, we address the challenge of aggregating duplicate support tickets in cloud computing platforms. We identify the limitations of traditional semantic similarity-based methods in handling diverse user reports of the same incident. To overcome this, we introduce iPACK, a solution that utilizes cloud-side runtime alerts to link and aggregate tickets through alert-alert and ticket-alert linking, enhancing deduplication accuracy. Evaluated on Azure datasets, iPACK demonstrated a significant improvement over existing methods, outperforming them by 12.4% to 31.2%. This innovative approach not only streamlines customer support operations but also offers a scalable solution for managing the complexities inherent in large-scale cloud systems. We also discussed success and failure cases in real-world usage of iPACK.

To summarize, this thesis addresses key research problems in

enhancing the reliability management of large-scale cloud systems. We begin by improving the observability of cloud systems through the inference of functional clusters using an efficient solution called Prism. Next, we introduce SeaLog, an LLM-enhanced log-based anomaly detection method, to produce more accurate alerts. Finally, we present iPACK, a comprehensive tool for managing support tickets and alerts to better understand incidents. These studies are conducted using real-world data collected from production environments of cloud systems. To benefit the community, we have made our data and code publicly available if possible.

7.2 Future Work

Ensuring the reliability of software systems is always crucial. This thesis primarily focuses on enhancing the monitoring and management of reliability data. As large language models (LLMs) are reshaping the world and gaining significant importance across various domains, new opportunities and challenges arise in this era of LLMs. Consequently, my future work will explore how to better utilize LLMs to improve the reliability of cloud systems and, conversely, how to enhance the reliability of LLM systems themselves. By addressing these interconnected aspects, we aim to advance the overall robustness and dependability of both cloud and LLM systems. Specifically, the first future work involves leveraging LLMs for more intelligent diagnosis of distributed systems, *i.e.*, *LLM-enhanced Distributed Diagnosis*. Given the extensive knowledge and powerful capabilities of LLMs, we can utilize and adapt this knowledge to enable automatic diagnosis for cloud systems, thus enhancing their reliability and efficiency. The second future work focuses on the better monitoring and optimization of LLM training systems, *i.e.*, *Monitoring and Diagnosis of LLM Training Systems*. Managing LLM systems presents unique challenges due to their complexity, scale, and resource demands. Therefore, it is

crucial to develop smarter monitoring and analysis techniques to ensure the optimal performance and stability of LLM training systems.

7.2.1 LLM-driven Distributed Diagnosis

Distributed systems, such as large-scale cloud systems, are composed of multiple interdependent services or components, making their diagnosis particularly complicated. Existing solutions have tackled these problems by utilizing machine learning-based methods to automate the diagnosis process of distributed systems [84, 135, 168]. However, such paradigm often struggle with addressing unseen cases and lack interpretability. To overcome these limitations, we plan to harness the power of large language models (LLMs) for diagnosing distributed systems. LLMs offer several advantages: (1) they possess a vast amount of knowledge that allows them to generalize and reason about unseen cases, (2) they can provide human-readable explanations for their outputs, enhancing interpretability, and (3) they can interact with real-world systems through techniques such as tool learning. However, there are several challenges to address in this LLM-driven paradigms, especially in large-scale cloud systems.

- First, current LLMs often lack domain-specific knowledge. While LLMs are trained on vast amounts of general data, they may not possess the nuanced understanding required for specific domains such as cloud infrastructure. For instance, diagnosing a complex issue in a distributed database system might require specialized knowledge about database internals, network protocols, and specific cloud service configurations, which general LLMs might not have. This limitation can hinder their ability to provide accurate diagnoses or actionable insights in highly specialized contexts.
- Second, current LLMs struggle to handle long contexts ef-

fectively. Distributed systems generate extensive logs and telemetry data that span large time frames and multiple components. For example, identifying a root cause might require correlating events from logs that span several days or involve multiple services. Current LLMs have limitations in processing such long sequences of data, often truncating or missing critical information, which can lead to incomplete or inaccurate analyses.

- Third, current LLMs are not efficient enough for real-time applications in large-scale cloud systems. The computational resources required to run LLMs are substantial, and their inference times can be prohibitive for real-time diagnostics. For example, during a critical system outage, waiting for an LLM to process and analyze logs could delay the resolution, exacerbating downtime and financial losses. Moreover, the energy consumption and costs associated with running these models at scale can be significant, making them impractical for continuous monitoring and diagnosis in a production environment.

To address these challenges, we plan to utilize a combination of Retrieval-Augmented Generation (RAG), tool learning, and the fusion of traditional machine learning-based methods. RAG can enhance domain-specific knowledge by retrieving relevant information from specialized databases and incorporating it into the LLM's responses, thereby improving its accuracy and relevance in cloud system diagnostics. Tool learning enables LLMs to interact with external tools, such as log analyzers and monitoring systems, allowing them to handle long contexts by delegating specific tasks to more specialized tools. This approach ensures that the LLM can focus on generating insights and explanations without being overwhelmed by extensive data. Additionally, by fusing LLMs with traditional machine learning-based methods, we can lever-

age the strengths of both paradigms. Traditional methods can efficiently process large volumes of data and identify patterns, while LLMs can provide interpretability and reasoning capabilities. This hybrid approach aims to create a more robust, efficient, and intelligent system for diagnosing and managing large-scale cloud systems.

7.2.2 Reliability of LLM Training Systems

The reliability of LLM training systems is both crucial and challenging due to the immense computational demands and the complexity of the infrastructure required. The vast number of parameters in modern LLMs necessitates significant computing power, which in turn requires larger computing clusters equipped with numerous high-performance devices and networks. Furthermore, as the size of model parameters and the number of training machines increase, so does the likelihood of failures during training. The synchronization requirements of the training process mean that even a local fault on a single GPU can cause the entire training process to fail. For example, Meta recently reported that during three months of training the OPT model, they encountered failures that led to over 35 manual restarts, 70 automatic restarts, and 100 cycling hosts, mostly due to infrastructure issues. These failures significantly impacted training efficiency and increased the workload for engineers, highlighting the critical need for more intelligent monitoring and diagnosis for LLM training systems.

To improve the reliability of LLM training systems, we plan to focus on two key directions: proactive fault tolerance and reactive fault management. Proactive fault tolerance aims to enhance system reliability through both system design and failure prediction. System design focuses on increasing redundancy in computation and training state, ensuring that the system can continue functioning even if some components fail. Additionally, failure prediction

will provide a proactive approach to mitigate the damages caused by potential failures, allowing for preemptive actions before issues escalate.

Reactive fault management will concentrate on quick fault diagnosis, involving two main components: Observability Data Collection and Optimization, and Knowledge-Empowered Fault Diagnosis. Observability Data Collection and Optimization will aim to reduce the cost and performance overhead associated with monitoring the system, ensuring that necessary data is gathered without significantly impacting performance. Knowledge-Empowered Fault Diagnosis will leverage accumulated knowledge to facilitate the analysis of recurring failures, enabling faster and more accurate identification and resolution of issues.

□ **End of chapter.**

Chapter 8

List of Publications

1. Jinyang Liu, Junjie Huang, Yintong Huo, Zhihan Jiang, Jiazhen Gu, Zhuangbin Chen, Cong Feng, Minzhi Yan, Michael R Lyu. "Scalable and Adaptive Log-based Anomaly Detection with Expert in the Loop." ArXiv preprint arXiv:2306.05032 (2024).

2. Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, Yintong Huo, Jiazhen Gu, Zhuangbin Chen, Jieming Zhu and Michael R. Lyu. "LILAC: Log Parsing using LLMs with Adaptive Parsing Cache." In Proceedings of the 32rd International Conference on Software Engineering (FSE 2024).

3. Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, Yintong Huo, Jiazhen Gu, Zhuangbin Chen, Jieming Zhu and Michael R. Lyu. "A Large-scale Evaluation for Log Parsing Techniques: How Far are We?." In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024).

4. Junjie Huang, Jinyang Liu, Zhuangbin Chen, Zhihan Jiang, Yichen Li, Jiazhen Gu, Cong Feng, Zengyin Yang, Yongqiang Yang, and Michael R. Lyu. "FaultProfIT: Hierarchical Fault Profiling of Incident Tickets in Large-scale Cloud Systems." In Proceedings of the 46th International Conference on Soft-

ware Engineering, Software Engineering in Practice track (ICSE-SEIP 2024).

5. Jinxi Kuang, Jinyang Liu, Junjie Huang, Renyi Zhong, Jiazhen Gu, Lan Yu, Rui Tan, Zengyin Yang, Michael R. Lyu. "Knowledge-aware Alert Aggregation in Large-scale Cloud Systems: a Hybrid Approach." In Proceedings of the 46th International Conference on Software Engineering, Software Engineering in Practice track (ICSE-SEIP 2024).

6. Jinyang Liu, Shilin He, Zhuangbin Chen, Liqun Li, Yu Kang, Xu Zhang, Pinjia He, Hongyu Zhang, Qingwei Lin, Zhangwei Xu, Saravan Rajmohan, Dongmei Zhang, Michael Lyu. "Incident-aware Duplicate Ticket Aggregation for Cloud Systems." In Proceedings of the 45h International Conference on Software and Engineering (ICSE 2023).

7. Jinyang Liu, Tianyi Yang, Zhuangbin Chen, Yuxin Su, Cong Feng, Zengyin Yang, Michael R. Lyu. "Practical Anomaly Detection over Multivariate Monitoring Metrics for Online Services." In Proceedings of the 34th IEEE International Symposium on Software Reliability Engineering (ISSRE 2023).

8. Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, Michael R. Lyu. "Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics." In Proceedings of the 34th IEEE International Symposium on Software Reliability Engineering (ISSRE 2023).

9. Jinyang Liu, Zhihan Jiang, Jiazhen Gu, Junjie Huang, Zhuangbin Chen, Cong Feng, Zengyin Yang, Yongqiang Yang, Michael R. Lyu. "Prism: Revealing Hidden Functional Clusters from Massive Instances in Cloud Systems." In Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE 2023).

10. Zhuangbin Chen, Jinyang Liu, Yuxin Su, Hongyu Zhang, Xiao Ling, Michael R. Lyu. "Adaptive Performance Anomaly Detection for Online Service Systems via Pattern Sketching." In Proceedings of the 44th International Conference on Software and Engineering (ICSE 2022).

11. Yichen Li, Xu Zhang, Shilin He, Zhuangbin Chen, Yu Kang, Jinyang Liu, Liqun Li, Yingnong Dang, Feng Gao, Zhangwei Xu, Saravan Rajmohan, Qingwei Lin, Dongmei Zhang, Michael R. Lyu. "An Intelligent Framework for Timely, Accurate, and Comprehensive Cloud Incident Detection." ACM SIGOPS Operating Systems Review, 2022.

12. Zhuangbin Chen, Jinyang Liu, Yuxin Su, Hongyu Zhang, Xuemin Wen, et al. Graph-based Incident Aggregation for Large-Scale Online Service Systems." In Proceedings of The 36th IEEE/ACM International Conference on Automated Software Engineering (ASE 2021).

13. Jinyang Liu, Jieming Zhu, Shilin He, Pinjia He, Zibin Zheng, and Michael R. Lyu. "Logzip: Extracting Hidden Structures via Iterative Clustering for Log Compression." In Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019).

14. Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, Michael R. Lyu. Tools and Benchmarks for Automated Log Parsing." In Proceedings of the 41st International Conference on Software Engineering, Software Engineering in Practice track (ICSE-SEIP 2019).

Chapter 4 is an adapted reprint of the publication "Prism: Revealing Hidden Functional Clusters from Massive Instances in Cloud Systems" that was published in the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE

2023). The thesis author is the primary author of this publication. This is a joint work with Zhihan Jiang, Jiazhen Gu, Junjie Huang, and Michael R. Lyu from The Chinese University of Hong Kong. Additionally, Zhuangbin Chen from the School of Software Engineering, Sun Yat-sen University, also played a significant role. The collaboration extends to Cong Feng, Zengyin Yang, and Yongqiang Yang from the Computing and Networking Innovation Lab at Huawei Cloud Computing Technology Co., Ltd.

Chapter 5 is an adapted reprint of the arXiv preprint "Scalable and Adaptive Log-based Anomaly Detection with Expert in the Loop". The thesis author is the primary author of this preprint. This is a joint work with Junjie Huang, Yintong Huo, Zhihan Jiang, Jiazhen Gu, and Michael R. Lyu from The Chinese University of Hong Kong. Additionally, Zhuangbin Chen from the School of Software Engineering, Sun Yat-sen University, also played a significant role. The collaboration extends to Cong Feng from the Computing and Networking Innovation Lab at Huawei Cloud Computing Technology Co., Ltd, and Minzhi Yan from the HCC Lab at Huawei Cloud Computing Technology Co., Ltd.

Chapter 6 is an adapted reprint of the paper "Incident-aware Duplicate Ticket Aggregation for Cloud Systems," presented at the 45th IEEE/ACM International Conference on Software Engineering (ICSE 2023). The thesis author is the primary author. This work is a collaboration with Shilin He, Liqun Li, Yu Kang, Xu Zhang, Qingwei Lin, Dongmei Zhang (Microsoft Research), Zhuangbin Chen, Michael R. Lyu (The Chinese University of Hong Kong), Pinjia He (The Chinese University of Hong Kong, Shenzhen), Hongyu Zhang (Chongqing University), Zhangwei Xu (Microsoft Azure), and Saravan Rajmohan (Microsoft 365).

□ **End of chapter.**

Bibliography

[1] E. Achtert, S. Goldhofer, H.-P. Kriegel, E. Schubert, and A. Zimek. Evaluation of clusterings–metrics and visual support. In *Proceedings of the 28th International Conference on Data Engineering (ICDE)*, pages 1285–1288, 2012.

[2] T. Ahmed, S. Ghosh, C. Bansal, T. Zimmermann, X. Zhang, and S. Rajmohan. Recommending root-cause and mitigation steps for cloud incidents using large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1737–1749. IEEE, 2023.

[3] AllenNLP. Allennlp. `https://allenai.org/allennlp`. Accessed: 2024-05-21.

[4] Amazon Web Services. Amazon cloudwatch documentation. `https://docs.aws.amazon.com/cloudwatch/index.html`. Accessed: 2024-05-21.

[5] Amazon Web Services. Aws support. `https://aws.amazon.com/premiumsupport/`. Accessed: 2024-05-21.

[6] Amazon Web Services. Aws support plan comparison. `https://aws.amazon.com/premiumsupport/plans/`. Accessed: 2024-05-21.

[7] Amazon Web Services. Logging ip traffic using vpc flow logs - amazon web services (aws). `https://docs.aws.amazon.com/vpc/latest/userguide/flow-logs.html`. Accessed: 2024-05-21.

[8] B. Arzani, S. Ciraci, S. Saroiu, A. Wolman, J. W. Stokes, G. Outhred, and L. Diwu. Privateeye: Scalable and privacy-preserving compromise detection in the cloud. In *NSDI*, pages 797–815, 2020.

[9] J. Audibert, P. Michiardi, F. Guyard, S. Marti, and M. A. Zuluaga. USAD: unsupervised anomaly detection on multivariate time series. In *Proceedings of the 26th SIGKDD Conference on Knowledge Discovery and Data Mining, (KDD)*, pages 3395–3404. ACM, 2020.

[10] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: automated classification of performance crises. In *Proceedings of the 5th European conference on Computer systems*, pages 111–124, 2010.

[11] G. E. Box and G. C. Tiao. *Bayesian inference in statistical analysis*. John Wiley & Sons, 2011.

[12] A. Z. Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (SEQUENCES)*, pages 21–29. IEEE, 1997.

[13] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[14] A. Budhiraja, K. Dutta, R. Reddy, and M. Shrivastava. Dwen: deep word embedding network for duplicate bug report detection in software repositories. In *Proceedings of the 40th International Conference on Software Engineering: companion proceeedings (ICSE-C)*, pages 193–194, 2018.

[15] A. Budhiraja, R. Reddy, and M. Shrivastava. Lwe: Lda refined word embeddings for duplicate bug report detection.

In *Proceedings of the 40th International Conference on Software Engineering: companion proceeedings (ICSE-C)*, pages 165–166, 2018.

[16] Y. Cao, H. Zhang, and S. Ding. Symcrash: Selective recording for reproducing crashes. In *Proceedings of the 29th International Conference on Automated software engineering (ASE)*, pages 791–802, 2014.

[17] O. Chaparro, J. M. Florez, U. Singh, and A. Marcus. Reformulating queries for duplicate bug report detection. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 218–229, 2019.

[18] 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 *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 111–120. IEEE, 2019.

[19] J. Chen, P. Wang, and W. Wang. Online summarizing alerts through semantic and behavior information. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, pages 1646–1657, 2022.

[20] J. Chen, S. Zhang, X. He, Q. Lin, H. Zhang, D. Hao, Y. Kang, F. Gao, Z. Xu, Y. Dang, et al. How incidental are the incidents? characterizing and prioritizing incidents for large-scale online service systems. In *Proceedings of the 35th International Conference on Automated Software Engineering (ASE)*, pages 373–384, 2020.

[21] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *Inter-*

national Conference on Autonomic Computing, 2004. Proceedings., pages 36–43. IEEE, 2004.

[22] Y. Chen, H. Xie, M. Ma, Y. Kang, X. Gao, L. Shi, Y. Cao, X. Gao, H. Fan, M. Wen, et al. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 674–688, 2024.

[23] Y. Chen, X. Yang, H. Dong, X. He, H. Zhang, Q. Lin, J. Chen, P. Zhao, Y. Kang, F. Gao, et al. Identifying linked incidents in large-scale online service systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 304–314, 2020.

[24] Y. Chen, X. Yang, Q. Lin, H. Zhang, F. Gao, Z. Xu, Y. Dang, D. Zhang, H. Dong, Y. Xu, et al. Outage prediction and diagnosis for cloud service systems. In *Proceedings of the 28th World Wide Web Conference (WWW)*, pages 2659–2665, 2019.

[25] Z. Chen, Y. Kang, L. Li, X. Zhang, H. Zhang, H. Xu, Y. Zhou, L. Yang, J. Sun, Z. Xu, et al. Towards intelligent incident management: why we need it and how we make it. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1487–1497, 2020.

[26] Z. Chen, J. Liu, W. Gu, Y. Su, and M. R. Lyu. Experience report: Deep learning-based system log analysis for anomaly detection. *arXiv preprint arXiv:2107.05908*, 2021.

[27] Z. Chen, J. Liu, Y. Su, H. Zhang, X. Ling, Y. Yang, and M. R. Lyu. Adaptive performance anomaly detection for

online service systems via pattern sketching. In *Proceedings of the 44th International Conference on Software Engineering*, pages 61–72, 2022.

[28] 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 *Proceedings of the 36th International Conference on Automated Software Engineering (ASE)*, pages 430–442, 2021.

[29] N. Cooper, C. Bernal-Cárdenas, O. Chaparro, K. Moran, and D. Poshyvanyk. It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, pages 957–969, 2021.

[30] D. Cotroneo, L. De Simone, P. Liguori, R. Natella, and N. Bidokhti. How bad can a bug get? an empirical analysis of software failures in the openstack cloud computing platform. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 200–211, 2019.

[31] H. Dai, H. Li, C.-S. Chen, W. Shang, and T.-H. Chen. Logram: Efficient log parsing using n n-gram dictionaries. *IEEE Transactions on Software Engineering (TSE)*, 48(3):879–892, 2020.

[32] D. Defays. An efficient algorithm for a complete link method. *The Computer Journal*, 20(4):364–366, 1977.

[33] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of*

(NAACL-HLT), pages 4171–4186. Association for Computational Linguistics, 2019.

[34] M. Du and F. Li. Spell: Streaming parsing of system event logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 859–864. IEEE, 2016.

[35] M. Du, F. Li, G. Zheng, and V. Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 1285–1298, 2017.

[36] Fabian Pedregosa and Philippe Gervais. Memory profiler. `https://github.com/pythonprofilers/memory_profiler`, 2022. Accessed: 2024-05-21.

[37] R. M. French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999.

[38] 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)*, pages 48–58, 2018.

[39] C. Gao, W. Zheng, Y. Deng, D. Lo, J. Zeng, M. R. Lyu, and I. King. Emerging app issue identification from user feedback: Experience on wechat. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 279–288, 2019.

[40] S. Gao, X.-C. Wen, C. Gao, W. Wang, H. Zhang, and M. R. Lyu. What makes good in-context demonstrations for code intelligence tasks with llms? In *Proceedings of the 38th International Conference on Automated Software Engineering (ASE)*, 2023.

[41] A. J. Gates and Y.-Y. Ahn. The impact of random models on clustering similarity. *Journal of Machine Learning Research (JMLR)*, 18:1–28, 2017.

[42] S. Ghosh, M. Shetty, C. Bansal, and S. Nath. How to fight production incidents? an empirical study on a large-scale cloud service. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 126–141, 2022.

[43] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio. Generative adversarial nets. In *Proceedings of the 27th Conference on Neural Information Processing Systems 2014, (NeurIPS)*, pages 2672–2680, 2014.

[44] Google Cloud. Cloud monitoring. `https://cloud.google.com/monitoring/`. Accessed: 2024-05-21.

[45] Google Cloud. Google support hub. `https://cloud.google.com/support-hub`. Accessed: 2024-05-21.

[46] Google Cloud. Introduction to alerting. `https://cloud.google.com/monitoring/alerts`. Accessed: 2024-05-21.

[47] J. Gu, J. Wen, Z. Wang, P. Zhao, C. Luo, Y. Kang, Y. Zhou, L. Yang, J. Sun, Z. Xu, et al. Efficient customer incident triage via linking with system incidents. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1296–1307, 2020.

[48] 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 (SOCC)*, pages 1–16, 2016.

[49] M. Haering, C. Stanik, and W. Maalej. Automatically matching bug reports with related app reviews. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, pages 970–981, 2021.

[50] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen. Logmine: Fast pattern recognition for log analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management (CIKM)*, pages 1573–1582, 2016.

[51] J. He, L. Xu, M. Yan, X. Xia, and Y. Lei. Duplicate bug report detection using dual-channel convolutional neural networks. In *Proceedings of the 28th International Conference on Program Comprehension (ICPC)*, pages 117–127, 2020.

[52] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu. An evaluation study on log parsing and its use in log mining. In *2016 46th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 654–661. IEEE, 2016.

[53] P. He, J. Zhu, Z. Zheng, and M. R. Lyu. Drain: An online log parsing approach with fixed depth tree. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*, pages 33–40. IEEE, 2017.

[54] S. He, Q. Lin, J.-G. Lou, H. Zhang, M. R. Lyu, and D. Zhang. Identifying impactful service system problems via log analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 60–70, 2018.

[55] S. He, J. Zhu, P. He, and M. R. Lyu. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th*

international symposium on software reliability engineering (ISSRE), pages 207–218. IEEE, 2016.

[56] S. He, J. Zhu, P. He, and M. R. Lyu. Loghub: A large collection of system log datasets towards automated log analytics. *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, 2020.

[57] J. Huang, J. Liu, Z. Chen, Z. Jiang, Y. Li, J. Gu, C. Feng, Z. Yang, Y. Yang, and M. R. Lyu. Faultprofit: Hierarchical fault profiling of incident tickets in large-scale cloud systems. *arXiv preprint arXiv:2402.17583*, 2024.

[58] 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)*, pages 150–155, 2017.

[59] HuggingFace. Transformers. `https://github.com/huggingface/transformers`. Accessed: 2024-05-21.

[60] K. Hundman, V. Constantinou, C. Laporte, I. Colwell, and T. Söderström. Detecting spacecraft anomalies using lstms and nonparametric dynamic thresholding. In *Proceedings of the 24th International Conference on Knowledge Discovery & Data Mining, (KDD)*, pages 387–395, 2018.

[61] Y. Huo, C. Lee, Y. Su, S. Shan, J. Liu, and M. R. Lyu. Evlog: Identifying anomalous logs over software evolution. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 391–402. IEEE, 2023.

[62] Y. Huo, Y. Su, C. Lee, and M. R. Lyu. Semparser: A semantic parser for log analytics. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 881–893. IEEE, 2023.

[63] M. Iliofotou, P. Pappu, M. Faloutsos, M. Mitzenmacher, S. Singh, and G. Varghese. Network monitoring using traffic dispersion graphs (tdgs). In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement (SIGCOMM)*, pages 315–320, 2007.

[64] N. Jain and S. Choudhary. Overview of virtualization in cloud computing. In *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*, pages 1–4. IEEE, 2016.

[65] James Lewis and Martin Fowler. Microservices: A definition of this new architectural term. `http://martinfowler.com/articles/microservices.html`, 2014. Accessed: 2024-05-21.

[66] Z. Jiang, J. Liu, Z. Chen, Y. Li, J. Huang, Y. Huo, P. He, J. Gu, and M. R. Lyu. Llmparser: A llm-based log parsing framework. *arXiv preprint arXiv:2310.01796*, 2023.

[67] Z. Jiang, J. Liu, J. Huang, Y. Li, Y. Huo, J. Gu, Z. Chen, J. Zhu, and M. R. Lyu. A large-scale benchmark for log parsing. *arXiv preprint arXiv:2308.10828*, 2023.

[68] Z. M. Jiang, A. E. Hassan, P. Flora, and G. Hamann. Abstracting execution logs to execution events for enterprise applications (short paper). In *2008 The Eighth International Conference on Quality Software*, pages 181–186. IEEE, 2008.

[69] P. Jin, S. Zhang, M. Ma, H. Li, Y. Kang, L. Li, Y. Liu, B. Qiao, C. Zhang, P. Zhao, et al. Assess and summarize: Improve outage understanding with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1657–1668, 2023.

[70] Y. Jin, E. Sharafuddin, and Z.-L. Zhang. Unveiling core network-wide communication patterns through application

traffic activity graph decomposition. *ACM SIGMETRICS Performance Evaluation Review*, 37(1):49–60, 2009.

[71] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[72] C. Jyoti and Z. Efpraxia. Understanding and exploring the value co-creation of cloud computing innovation using resource based value theory: An interpretive case study. *Journal of Business Research*, 164:113970, 2023.

[73] A. Kane and N. Shiri. Multivariate time series representation and similarity search using pca. In *Advances in Data Mining. Applications and Theoretical Aspects: 17th Industrial Conference, ICDM 2017*, pages 122–136. Springer, 2017.

[74] Z. A. Khan, D. Shin, D. Bianculli, and L. Briand. Guidelines for assessing the accuracy of log message template identification techniques. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, pages 1095–1106, 2022.

[75] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2015.

[76] B. Kucuk and E. Tuzun. Characterizing duplicate bugs: An empirical analysis. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 661–668. IEEE, 2021.

[77] A. Lamkanfi, J. Pérez, and S. Demeyer. The eclipse and mozilla defect tracking dataset: a genuine dataset for mining bug information. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*, pages 203–206, 2013.

[78] A. Lazar, S. Ritchey, and B. Sharif. Generating duplicate bug datasets. In *Proceedings of the 11th working conference on mining software repositories (MSR)*, pages 392–395, 2014.

[79] V.-H. Le and H. Zhang. Log-based anomaly detection without log parsing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 492–504. IEEE, 2021.

[80] V.-H. Le and H. Zhang. Log-based anomaly detection with deep learning: How far are we? In *Proceedings of the 44th International Conference on Software Engineering*, pages 1356–1367, 2022.

[81] V.-H. Le and H. Zhang. An evaluation of log parsing with chatgpt. *arXiv preprint arXiv:2306.01590*, 2023.

[82] V.-H. Le and H. Zhang. Log parsing with prompt-based few-shot learning. *arXiv preprint arXiv:2302.07435*, 2023.

[83] C. Lee, T. Yang, Z. Chen, Y. Su, and M. R. Lyu. Eadro: An end-to-end troubleshooting framework for microservices on multi-source data. *arXiv preprint arXiv:2302.05092*, 2023.

[84] C. Lee, T. Yang, Z. Chen, Y. Su, Y. Yang, and M. R. Lyu. Heterogeneous anomaly detection for software systems via semi-supervised cross-modal attention. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1724–1736. IEEE, 2023.

[85] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of massive data sets*. Cambridge university press (CAMB), 2020.

[86] L. Li, X. Zhang, X. Zhao, H. Zhang, Y. Kang, P. Zhao, B. Qiao, S. He, P. Lee, J. Sun, et al. Fighting the fog of war:

Automated incident detection for cloud systems. In *USENIX Annual Technical Conference*, pages 131–146, 2021.

[87] X. Li, H. Jiang, D. Liu, Z. Ren, and G. Li. Unsupervised deep bug report summarization. In *Proceedings of the 26th International Conference on Program Comprehension (ICPC)*, pages 144–14411. IEEE, 2018.

[88] Y. Li, X. Zhang, S. He, Z. Chen, Y. Kang, J. Liu, L. Li, Y. Dang, F. Gao, Z. Xu, et al. An intelligent framework for timely, accurate, and comprehensive cloud incident detection. *ACM SIGOPS Operating Systems Review*, 56(1):1–7, 2022.

[89] Z. Li, H. Li, T.-H. Chen, and W. Shang. Deeplv: Suggesting log levels using ordinal based neural networks. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, pages 1461–1472, 2021.

[90] Z. Li, C. Luo, T.-H. Chen, W. Shang, S. He, Q. Lin, and D. Zhang. Did we miss something important? studying and exploring variable-aware log abstraction. *arXiv preprint arXiv:2304.11391*, 2023.

[91] Z. Li, Y. Zhao, R. Liu, and D. Pei. Robust and rapid clustering of kpis for large-scale anomaly detection. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2018.

[92] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo. Failure prediction in ibm bluegene/l event logs. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, pages 583–588. IEEE, 2007.

[93] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen. Log clustering based problem identification for online service sys-

tems. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 102–111, 2016.

[94] D. Liu, Y. Feng, X. Zhang, J. Jones, and Z. Chen. Clustering crowdsourced test reports of mobile applications using image understanding. *IEEE Transactions on Software Engineering (TSE)*, 2020.

[95] F. T. Liu, K. M. Ting, and Z.-H. Zhou. Isolation forest. In *2008 eighth IEEE international conference on data mining*, pages 413–422. IEEE, 2008.

[96] 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)*, pages 155–162, 2019.

[97] H. Liu, M. Shen, J. Jin, and Y. Jiang. Automated classification of actions in bug reports of mobile apps. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 128–140, 2020.

[98] J. Liu, S. He, Z. Chen, L. Li, Y. Kang, X. Zhang, P. He, H. Zhang, Q. Lin, Z. Xu, et al. Incident-aware duplicate ticket aggregation for cloud systems. *arXiv preprint arXiv:2302.09520*, 2023.

[99] J. Liu, J. Zhu, S. He, P. He, Z. Zheng, and M. R. Lyu. Logzip: extracting hidden structures via iterative clustering for log compression. In *Proceedings of the 34th International Conference on Automated Software Engineering (ASE)*, pages 863–873, 2019.

[100] Y. Liu, X. Zhang, S. He, H. Zhang, L. Li, Y. Kang, Y. Xu, M. Ma, Q. Lin, Y. Dang, et al. Uniparser: A unified

log parser for heterogeneous log data. In *Proceedings of the ACM Web Conference 2022 (WWW)*, pages 1893–1901, 2022.

[101] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li. Mining invariants from console logs for system problem detection. In *USENIX annual technical conference*, pages 1–14, 2010.

[102] S. Lu, X. Wei, Y. Li, and L. Wang. Detecting anomaly in big data system logs using convolutional neural network. In *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*, pages 151–158. IEEE, 2018.

[103] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.

[104] W. Ma, L. Chen, X. Zhang, Y. Zhou, and B. Xu. How do developers fix cross-project correlated bugs? a case study on the github scientific python ecosystem. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, pages 381–392, 2017.

[105] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, pages 1255–1264, 2009.

[106] L. Malhotra, D. Agarwal, A. Jaiswal, et al. Virtualization in cloud computing. *J. Inform. Tech. Softw. Eng*, 4(2):1–3, 2014.

[107] P. Malhotra, A. Ramakrishnan, G. Anand, L. Vig, P. Agarwal, and G. Shroff. Lstm-based encoder-decoder for multi-sensor anomaly detection. *CoRR*, abs/1607.00148, 2016.

[108] 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 anomalies in unstructured logs. In *IJCAI*, volume 19, pages 4739–4745, 2019.

[109] S. Messaoudi, A. Panichella, D. Bianculli, L. Briand, and R. Sasnauskas. A search-based approach for accurate identification of log message formats. In *Proceedings of the 26th Conference on Program Comprehension*, pages 167–177, 2018.

[110] Microsoft Azure. Azure support options. `https://azure.microsoft.com/en-us/support/`. Accessed: 2024-05-21.

[111] Microsoft Azure. Overview of azure monitor alerts - azure monitor. `https://docs.microsoft.com/en-us/azure/azure-monitor/alerts/alerts-overview`. Accessed: 2024-05-21.

[112] Microsoft Azure. Support scope and responsiveness. `https://azure.microsoft.com/en-us/support/plans/response/`. Accessed: 2024-05-21.

[113] Microsoft Azure. What is azure? `https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure`. Accessed: 2024-05-21.

[114] M. Mizutani. Incremental mining of system log format. In *2013 IEEE International Conference on Services Computing*, pages 595–602. IEEE, 2013.

[115] J. Mohan, A. Phanishayee, J. Kulkarni, and V. Chidambaram. Looking beyond {GPUs} for {DNN} scheduling on {Multi-Tenant} clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 579–596, 2022.

[116] M. Müller. Dynamic time warping. *Information retrieval for music and motion*, pages 69–84, 2007.

[117] M. Nagappan and M. A. Vouk. Abstracting log lines to log event types for mining software system logs. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 114–117. IEEE, 2010.

[118] S. Nagaraja, P. Mittal, C.-Y. Hong, M. Caesar, and N. Borisov. Botgrep: Finding p2p bots with structured graph analysis. In *USENIX security symposium (USENIX security)*, volume 10, pages 95–110, 2010.

[119] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Proceedings of the 27th International Conference on Automated Software Engineering (ASE)*, pages 70–79, 2012.

[120] F. Nielsen and F. Nielsen. Hierarchical clustering. *Introduction to HPC with MPI for Data Science*, pages 195–211, 2016.

[121] A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *37th annual IEEE/IFIP international conference on dependable systems and networks (DSN'07)*, pages 575–584. IEEE, 2007.

[122] OpenAI. Openai embeddings. `https://platform.openai.com/docs/guides/embeddings`. Accessed: 2024-05-21.

[123] OpenAI. Introducing chatgpt. `https://openai.com/blog/chatgpt`, 2022. Accessed: 2024-05-21.

[124] W. Pang, S. Panda, J. Amjad, C. Diot, and R. Govindan. {CloudCluster}: Unearthing the functional structure of a cloud service. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1213–1230, 2022.

[125] J. Paparrizos and L. Gravano. k-shape: Efficient and accurate clustering of time series. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data (SIGMOD)*, pages 1855–1870, 2015.

[126] H. Qin, Y. Tian, and Y. Song. Relation extraction with word graphs from n-grams. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2860–2868, 2021.

[127] Radim Řehůřek. Gensim. `https://radimrehurek.com/gensim/`. Accessed: 2024-05-21.

[128] W. M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical association*, 66(336):846–850, 1971.

[129] S. Rastkar, G. C. Murphy, and G. Murray. Automatic summarization of bug reports. *IEEE Transactions on Software Engineering (TSE)*, 40(4):366–380, 2014.

[130] S. Rendle. Factorization machines. In *2010 IEEE International conference on data mining*, pages 995–1000. IEEE, 2010.

[131] V. Satopaa, J. Albrecht, D. Irwin, and B. Raghavan. Finding a" kneedle" in a haystack: Detecting knee points in system behavior. In *2011 31st international conference on distributed computing systems workshops*, pages 166–171. IEEE, 2011.

[132] Scikit-learn. Scikit-learn. `https://scikit-learn.org/`. Accessed: 2024-05-21.

[133] M. A. Shahid, N. Islam, M. M. Alam, M. M. Su'ud, and S. Musa. Towards resilient method: An exhaustive survey of fault tolerance methods in the cloud computing environment. *Comput. Sci. Rev.*, 40:100398, 2021.

[134] K. Shima. Length matters: Clustering system log messages using length of words. *arXiv preprint arXiv:1611.03213*, 2016.

[135] J. Soldani and A. Brogi. Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey. *ACM Computing Surveys (CSUR)*, 55(3):1–39, 2022.

[136] Y. Su, Y. Zhao, C. Niu, R. Liu, W. Sun, and D. Pei. Robust anomaly detection for multivariate time series through stochastic recurrent neural network. In *Proceedings of the 25th International Conference on Knowledge Discovery & Data Mining, (KDD)*, pages 2828–2837, 2019.

[137] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the 26th International Conference on Automated Software Engineering (ASE)*, pages 253–262, 2011.

[138] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo. A discriminative model approach for accurate duplicate bug

report retrieval. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 45–54, 2010.

[139] L. Tang, T. Li, and C.-S. Perng. Logsig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM international conference on Information and knowledge management (CIKM)*, pages 785–794, 2011.

[140] T. T. Tanimoto. Elementary mathematical theory of classification and prediction. 1958.

[141] R. Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM)(IEEE Cat. No. 03EX764)*, pages 119–126. Ieee, 2003.

[142] R. Vaarandi and M. Pihelgas. Logcluster-a data clustering and pattern mining algorithm for event logs. In *2015 11th International conference on network and service management (CNSM)*, pages 1–7. IEEE, 2015.

[143] H. Wang and D. Yeung. Towards bayesian deep learning: A framework and some existing methods. *IEEE Trans. Knowl. Data Eng.*, 28(12):3395–3408, 2016.

[144] L. Wang, N. Zhao, J. Chen, P. Li, W. Zhang, and K. Sui. Root-cause metric location for microservice systems via log anomaly detection. In *2020 IEEE international conference on web services (ICWS)*, pages 142–150. IEEE, 2020.

[145] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th international Conference on Software Engineering (ICSE)*, pages 461–470, 2008.

[146] X. Wang, X. Zhang, L. Li, S. He, H. Zhang, Y. Liu, L. Zheng, Y. Kang, Q. Lin, Y. Dang, et al. Spine: a scalable log parser with feedback guidance. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1198–1208, 2022.

[147] Y. Wang, G. Li, Z. Wang, Y. Kang, Y. Zhou, H. Zhang, F. Gao, J. Sun, L. Yang, P. Lee, et al. Fast outage analysis of large-scale production clouds with service correlation mining. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 885–896. IEEE, 2021.

[148] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding. Mlaas in the wild: Workload analysis and scheduling in large-scale heterogeneous gpu clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960. USENIX Association, 2022.

[149] Wikipedia. Component (graph theory). `http://en.wikipedia.org/w/index.php?title=Component_(graph_theory)`. Accessed: 2024-05-21.

[150] Wikipedia. Curse of dimensionality. `http://en.wikipedia.org/w/index.php?title=Curse_of_dimensionality`. Accessed: 2024-05-21.

[151] Wikipedia. Hierarchical clustering. `https://en.wikipedia.org/wiki/Hierarchical_clustering`. Accessed: 2024-05-21.

[152] Wikipedia. Jaccard index - wikipedia. `https://en.wikipedia.org/wiki/Jaccard_index`. Accessed: 2024-05-21.

[153] Wikipedia. Lidstone smoothing. `https://en.wikipedia.org/wiki/Additive_smoothing`. Accessed: 2024-05-21.

[154] Wikipedia. Pointwise mutual information. `http://en.wikipedia.org/w/index.php?title=Pointwise_mutual_information`. Accessed: 2024-05-21.

[155] H. Wu, W. Deng, X. Niu, and C. Nie. Identifying key features from app user reviews. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, pages 922–932, 2021.

[156] Y. Xing and Y. Zhan. Virtualization and cloud computing. In *Future Wireless Networks and Information Systems: Volume 1*, pages 305–312. Springer, 2012.

[157] H. Xu, W. Chen, N. Zhao, Z. Li, J. Bu, Z. Li, Y. Liu, Y. Zhao, D. Pei, Y. Feng, J. Chen, Z. Wang, and H. Qiao. Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web, (WWW)*, pages 187–196. ACM, 2018.

[158] J. Xu, R. Yang, Y. Huo, C. Zhang, and P. He. Prompting for automatic log template extraction. *arXiv preprint arXiv:2307.09950*, 2023.

[159] K. Xu, F. Wang, and L. Gu. Network-aware behavior clustering of internet end hosts. In *2011 proceedings ieee infocom (INFOCOM)*, pages 2078–2086. IEEE, 2011.

[160] W. Xu. *System problem detection by mining console logs*. University of California, Berkeley, 2010.

[161] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Largescale system problem detection by mining console logs.

In *Proceedings of SOSP*, volume 9, pages 1–17. Citeseer, 2009.

[162] L. Yang, J. Chen, Z. Wang, W. Wang, J. Jiang, X. Dong, and W. Zhang. Plelog: Semi-supervised log-based anomaly detection via probabilistic label estimation. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 230–231. IEEE, 2021.

[163] L. Yang, J. Chen, Z. Wang, W. Wang, J. Jiang, X. Dong, and W. Zhang. Semi-supervised log-based anomaly detection via probabilistic label estimation. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1448–1460. IEEE, 2021.

[164] 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 *Proceedings of the 36th International Conference on Automated Software Engineering (ASE)*, pages 653–665, 2021.

[165] T. Yang, J. Shen, Y. Su, X. Ren, Y. Yang, and M. R. Lyu. Characterizing and mitigating anti-patterns of alerts in industrial cloud systems. In *Proceedings of the 52st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2022.

[166] X. Yang, D. Lo, X. Xia, L. Bao, and J. Sun. Combining word embedding with information retrieval to recommend similar bug reports. In *Proceedings of the 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 127–137. IEEE, 2016.

[167] L. Yao, C. Mao, and Y. Luo. Graph convolutional networks for text classification. In *Proceedings of the AAAI conference on artificial intelligence (AAAI)*, pages 7370–7377, 2019.

[168] G. Yu, P. Chen, Y. Li, H. Chen, X. Li, and Z. Zheng. Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 553–565, 2023.

[169] C. Zhang, X. Peng, C. Sha, K. Zhang, Z. Fu, X. Wu, Q. Lin, and D. Zhang. Deeptralog: Trace-log combined microservice anomaly detection through graph-based deep learning. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, pages 623–634, 2022.

[170] S. Zhang, D. Li, Z. Zhong, J. Zhu, M. Liang, J. Luo, Y. Sun, Y. Su, S. Xia, Z. Hu, et al. Robust system instance clustering for large-scale web services. In *Proceedings of the ACM Web Conference 2022 (WWW)*, pages 1785–1796, 2022.

[171] S. Zhang, Z. Zhong, D. Li, Q. Fan, Y. Sun, M. Zhu, Y. Zhang, D. Pei, J. Sun, Y. Liu, et al. Efficient kpi anomaly detection through transfer learning for large-scale web services. *IEEE Journal on Selected Areas in Communications (JSAC)*, 40(8):2440–2455, 2022.

[172] 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 *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 807–817, 2019.

[173] G. Zhao, S. Hassan, Y. Zou, D. Truong, and T. Corbin. Predicting performance anomalies in software systems at run-time. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(3):1–33, 2021.

[174] N. Zhao, J. Chen, X. Peng, H. Wang, X. Wu, Y. Zhang, Z. Chen, X. Zheng, X. Nie, G. Wang, et al. Understanding and handling alert storm for online service systems. In *Proceedings of the 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 162–171, 2020.

[175] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[176] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G. Halfond. Recdroid: automatically reproducing android application crashes from bug reports. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 128–139, 2019.

[177] W. Zheng, H. Lu, Y. Zhou, J. Liang, H. Zheng, and Y. Deng. ifeedback: exploiting user feedback for real-time issue detection in large-scale online service systems. In *Proceedings of the 34th International Conference on Automated Software Engineering (ASE)*, pages 352–363, 2019.

[178] J. Zhou and H. Zhang. Learning to rank duplicate bug reports. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM)*, pages 852–861, 2012.

[179] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding. Fault analysis and debugging of microservice systems: In-

dustrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260, 2018.

[180] 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 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, pages 683–694, 2019.

[181] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 121–130. IEEE, 2019.

[182] B. Zong, Q. Song, M. R. Min, W. Cheng, C. Lumezanu, D. Cho, and H. Chen. Deep autoencoding gaussian mixture model for unsupervised anomaly detection. In *Proceedings of the 6th International Conference on Learning Representations, (ICLR)*, 2018.

[183] W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu. How practitioners perceive automated bug report management techniques. *IEEE Transactions on Software Engineering (TSE)*, 46(8):836–862, 2018.