Intelligent Reliability Monitoring and Engineering for Online Service Systems

CHEN, Zhuangbin

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
January 2023

Thesis Assessment Committee

Professor LO Chi Lik Eric (Chair)

Professor LYU Rung Tsong Michael (Thesis Supervisor)

Professor LEE Pak Ching Patrick (Committee Member)

Professor TAO Xie (External Examiner)

Abstract of thesis entitled:

Intelligent Reliability Monitoring and Engineering for Online Service Systems

Submitted by CHEN, Zhuangbin
for the degree of Doctor of Philosophy
at The Chinese University of Hong Kong in January 2023

The proliferation of distributed internet applications has imposed stringent reliability, performance, and scaling requirements on online service systems. Cloud systems, where online services reside, consist of a large variety of both hardware (e.g., networking equipment, servers, data storage) and software (e.g., virtualization software, service programs) components. Since any given component possesses a small but non-zero failure probability, the large number of components means that service incidents (interruptions or performance degradation of a service or product) are inevitable. Unfortunately, not all incidents are easily detectable within cloud environments.

To gain visibility into the internals of online service systems, various monitoring data are accumulated to reflect their runtime state from different perspectives, including logs, metrics, alerts/events, and topologies. Data-driven algorithms based on the massive amount of monitoring data have been proven an effective means to accelerate and automate problem detection and resolution in complex cloud systems. This thesis introduces our research of intelligent reliability monitoring and engineering for online service systems.

First, it involves an empirical investigation into the current

status of incident management at Microsoft. Based on field studies and our analysis of two years of incident tickets, we identify a series of pain points of incident handling in production systems: imprecise failure impact estimation, redundant engineering efforts, flooding alarms, and gray failures. We discuss the reasons behind them from the perspective of cloud system design and operation. We also present Microsoft's attempts to leverage data-driven techniques to cope with these challenges. Our study reveals some crucial problems in this line of research that have not yet been addressed by existing work. We are motivated to fill these significant gaps.

Second, we conduct a systematic review and evaluation of deep learning techniques for log anomaly detection. Log anomaly detection plays a vital role in software reliability management. While neural network models have substantially boosted their performance in recent years, a comprehensive benchmark and an open-source toolkit are lacking. Therefore, we implement six representative methods and compare their performance in terms of accuracy, robustness, and efficiency. We further share our experience of industrial deployment and our vision of important future directions. Our study enables better integration and customization of log anomaly detection in the industry, alleviating the problems of flooding alarms and gray failures.

Third, we propose ADSketch, a performance anomaly detection method based on metrics. ADSketch features the merits of interpretability and adaptability. In online services, similar anomalies often manifest as similar unusual metric patterns. ADSketch extracts such patterns efficiently, which in turn help interpret the anomaly type. To adapt to new patterns, ADSketch carefully compares inputs with the learned patterns and refreshes them accordingly. ADSketch has been deployed in production environments and demonstrated promising results in combating flooding alarms and gray failures.

Last, we propose GIRDLE, an unsupervised and unified alert aggregation framework based on graph representation learning. Alerts provide timely awareness of the problems in online service systems. Due to complex dependencies in cloud systems, incidents often come with flooding alerts. By identifying related alerts, GIRDLE helps estimate the failure scope and save duplicate engineering efforts in handling them. To this end, GIRDLE leverages multi-source information (i.e., alerts, metrics, and topologies) to profile the failure symptoms of services. Such a design is essential for tracing the propagation of failures precisely.

In summary, this thesis targets the use of data analytics and machine learning techniques to improve the reliability monitoring and engineering of online service systems and gain actionable insights to accelerate incident diagnosis. Extensive experiments on both public and industrial datasets validate the effectiveness and efficiency of our proposed algorithms.

論文題目：在線服務系統的智能可靠性監測與工程

作者　　：陳壯彬

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　　：

分布式互聯網應用的激增對在線服務系統提出了嚴格的可靠性、性能和可擴展性需求。在線服務所在的雲系統由多種硬件（如網絡設備、服務器、數據存儲）和軟件（如虛擬化軟件、服務程序）組成。由於任何組件都具有較小但非零的故障概率，因此組件數量多意味著服務事故（服務或產品的中斷或性能下降）不可避免。不幸的是，並非所有事故都能在雲環境中輕易檢測到。

為了了解在線服務系統的內部情況，需要積累各種監控數據，以從不同的角度反映其運行時狀態，包括日誌、度量、告警/問題事件和拓撲。基於大量監控數據的數據驅動算法已被證明是在復雜雲系統中加速和自動化問題檢測和解決的有效手段。本論文介紹了我們對在線服務系統可靠性的智能監測與工程的研究。

首先，它涉及對Microsoft事故管理現狀的實證調查。基於實地研究和我們對兩年事故單的分析，我們確定了生產系統中事故處理的一系列痛點：不精確的故障影響評估、冗余的運維工作、洪水告警和灰色故障。我們從雲系統設計和運行的角度討論了它們背後的原因。我們還介紹了Microsoft利用數據驅動技術應對這些挑戰的嘗試。我們的研究揭示了這一研究領域的一些關鍵問題，這些問題尚未被現有工作解決。我們致力於填補這些重要空白。

其次，我們對用於日誌異常檢測的深度學習技術進行了系統的回顧和評估。日誌異常檢測在軟件可靠性管理中起著至關重要的作用。盡管近年來神經網絡模型使其性能得到了大幅提升，但全面的基準探究和開源工具包仍然缺乏。因此，我們實現了六種具有代表性的方法，並在準確性、魯棒性和效率方面

比較了它們的性能。我們進一步分享我們的工業部署經驗和我們對未來重要方向的願景。我們的研究使得業界能夠更好地集成和定制日誌異常檢測的方法，緩解洪水警報和灰色故障的問題。

第三，我們提出了ADSketch，一種基於度量的性能異常檢測方法。ADSketch具有可解釋性和適應性的優點。在在線服務中，類似的異常通常表現為類似且罕見的異常度量模式。ADSketch能夠有效地提取這些模式，進而幫助解釋異常類型。為了適應新的模式，ADSketch仔細比較其輸入與已學習的模式，並相應地更新它們。ADSketch已部署在生產環境中，並在應對洪水告警和灰色故障方面取得了良好的效果。

最後，我們提出了Girdle，一個基於圖表征學習的無監督統一告警聚合框架。告警可及時了解在線服務系統中的問題。由於雲系統中的復雜依賴關系，事故通常伴隨著洪水告警。通過識別相關告警，Girdle有助於估計故障範圍，從而節省處理這些告警的重復運維工作。為此，Girdle利用多源信息（即告警、度量和拓撲）來描述服務的故障癥狀。這種設計對於精確跟蹤故障傳播至關重要。

綜上所述，本論文的目標是使用數據分析和機器學習技術來改進在線服務系統的可靠性監控與工程，並獲得可實施的洞察以加速事故診斷。在公共和工業數據集上的大量實驗驗證了我們提出的算法的有效性和效率。

Acknowledgement

First and foremost, I would like to acknowledge Prof. Michael R. Lyu, for all of his support and guidance over the years. Michael is a very generous person who believes everyone deserves a second chance. His generosity and wisdom made my achievements possible. What I have learned from Michael, beyond the technical stuff, are kindness, honesty, and rigorousness that I could use in my entire life.

In addition, I would like to acknowledge the role played by my thesis committee, Prof. Eric Lo and Prof. Patrick Lee, for their constructive comments and valuable suggestions on this thesis and all my term presentations. Great thanks to Prof. Tao Xie from Peking University, who kindly serves as the external examiner for my thesis.

Over the years, I have had the chance to collaborate with several talented researchers and engineers from Huawei Cloud and Microsoft Research on various projects that appear within this thesis. I also had a great time working with Prof. Hongyu Zhang from The University of Newcastle. I would like to express my thanks and gratitude to all of them. Acknowledgments are also due to my friends at ARISE Lab, for making this journey joyful and enriching. And this is important for me.

Significant acknowledgment goes to my family, including my mother, father, brother, sister, and sister-in-law, for being supportive and understanding throughout this process.

Dedicated to my mother, my beloved family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

With the emergence of cloud computing platforms (e.g., Microsoft Azure, Amazon Web Services, and Google Cloud Platform), many traditional software systems have been migrated to the cloud as online services. They have benefited many enterprises by taking over the maintenance of IT and infrastructure, allowing them to improve their core business competence. Different from conventional shrink-wrapped software, online services need to serve millions of customers worldwide. Thus, the reliability of online service systems is an important quality attribute. Service incidents, such as unplanned interruptions and outages, could cause performance degradation (e.g., slow response time) and service unavailability if not mitigated timely and properly, leading to economic loss and user dissatisfaction. For example, in July 2018, an hour episode of downtime on Amazon's annual Prime Day led to a loss of $100+ million[1]. In October 2021, Facebook and its subsidiaries became globally unavailable for a period of six to seven hours, resulting in a nearly 5% drop in the company's shares and at least a $60 million loss in advertising revenue[2]. Experience indicates that online service systems are rife with

[1] https://techcrunch.com/2018/07/18/amazon-prime-day-outage-cost/

[2] https://en.wikipedia.org/wiki/2021_Facebook_outage

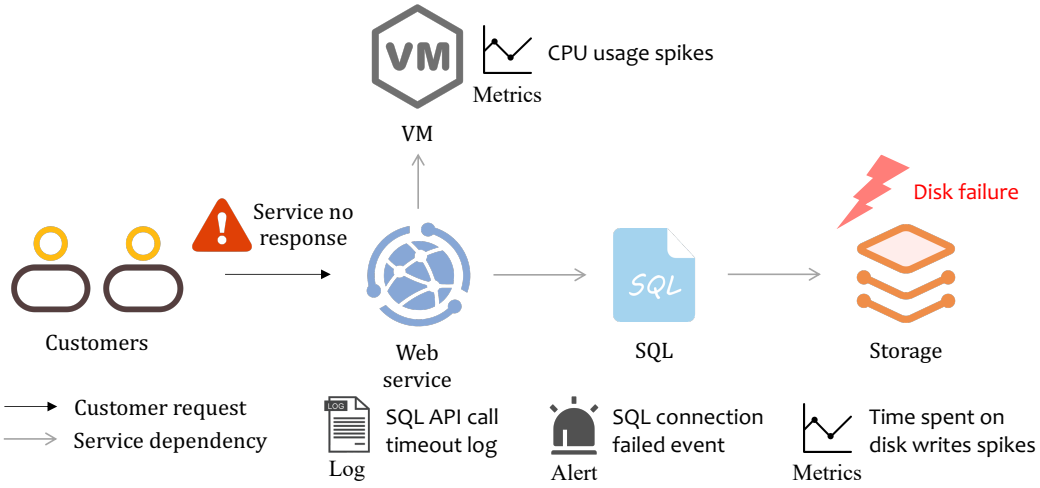


Figure 1.1: A Case of Monitoring Data Generation in Online Service Systems

hardware and software failures, due to the use of commodity hardware (exhibiting low but non-zero failure rates) coupled with software bugs, misconfigurations, and failures caused by service deployment and upgrade [107, 67]. Therefore, intelligent reliability monitoring and engineering have become a core selling point of online service providers. In this thesis, we use "incident" and "failure" interchangeably. Cloud vendors who have promoted the automation of reliability management have already received real gains in availability, efficiency, and agility [29, 37, 107].

To understand the health condition of service systems, different types of monitoring data are generated to continuously reflect the system state from different perspectives, including logs [68, 70], metrics [12], alerts/events [97, 191], topologies [121, 7, 149]. Figure 1.1 presents a case of monitoring data generation in online service systems. A disk failure happens to the Storage service, whose impact propagates along the topology of service dependencies. Consequently, customers' requests may receive no response. In this process, different services produce certain types of monitoring data reporting the failure symptoms. For example, to serve customer requests, the Web Service needs to query the

database for user authentication, transaction recovery, etc. Failing to do so results in a log saying "SQL API call timeout." As a result, customers will send multiple requests, consuming more CPU resources of the VMs where the Web Service resides. Based on the pre-defined alerting policy, the SQL service may generate an alert describing the connection failure. In the Storage service, we may also see spikes in the metric monitoring the time spent on disk writes. Traditionally, engineers rely on manual inspection of the data for system troubleshooting. For example, they perform simple keyword searches (e.g., "exception," "failed," "error", and "timeout") to mine suspicious logs that might be associated with the underlying problems, e.g., a failed read/write operation. Also, they resort to setting fixed thresholds on metrics to detect unhealthy moments of the system, e.g., long latency. However, today's cloud systems have imposed considerable challenges on such human-dependent IT operations:

- *Large scale and complexity.* Due to the ever-increasing scale and complexity of today's online services, the volume, variety, and velocity of the monitoring data have reached an unprecedented level. The traditional approaches fall short for being labor-intensive and error-prone.
- *Fast development iteration.* As the demand for new software and features skyrockets in the digital age, services are being developed rapidly through frequent iterations and continuous user feedback. Human-defined rules for failure detection cannot keep pace with newly emerging failure patterns.
- *Complicated service dependencies.* Online services often depend on each other to provide and support complete applications. As a result, minor issues could propagate their impact and escalate into system-wide outages. Service dependencies significantly increase the difficulty of manually maintaining the well-being of online services (e.g., root cause localization and impact scoping).

Therefore, inspecting heterogeneous IT data for troubleshooting often requires a decent knowledge of the entire online service system, which often goes beyond the ability of a single engineer. This situation serves as a catalyst for the algorithmic analysis of monitoring data toward effective and intelligent service management. Specifically, it leverages big data, analytics, and machine learning techniques to gain actionable information and uncover powerful insights from an immense quantity of data to enhance cloud IT operations.

In this context, we conduct research on intelligent reliability monitoring and engineering for online service systems based on various IT data, i.e., logs, metrics, alerts, and topologies. As shown in the left part of Figure 1.2, these data are generated by services in different cloud layers. The right part of Figure 1.2 shows our contributions in this thesis, which comprise four parts. First, we provide an empirical study on industrial incident management at Microsoft. We highlight two critical challenges of incident handling (i.e., resource health assessment and resource dependency discovery) and investigate the reasons behind them. They serve as an important guideline for the following studies of service reliability. Second, we present a systematic review and evaluation of popular log anomaly detection methods powered by deep learning techniques. Researchers and practitioners can benefit from such a study as it enables them to quickly understand the characteristics of these methods and select appropriate ones for their problems. Third, we propose ADSketch, a performance anomaly detection method based on metrics. It features the ability to interpret the detected anomalies and learn new patterns online. These efforts tackle the challenge of resource health assessment by pursuing more accurate anomaly detection based on logs and metrics. Finally, we propose GIRDLE, an alert aggregation framework based on graph representation learning. We make use of multi-source monitoring data (i.e., alerts, met-

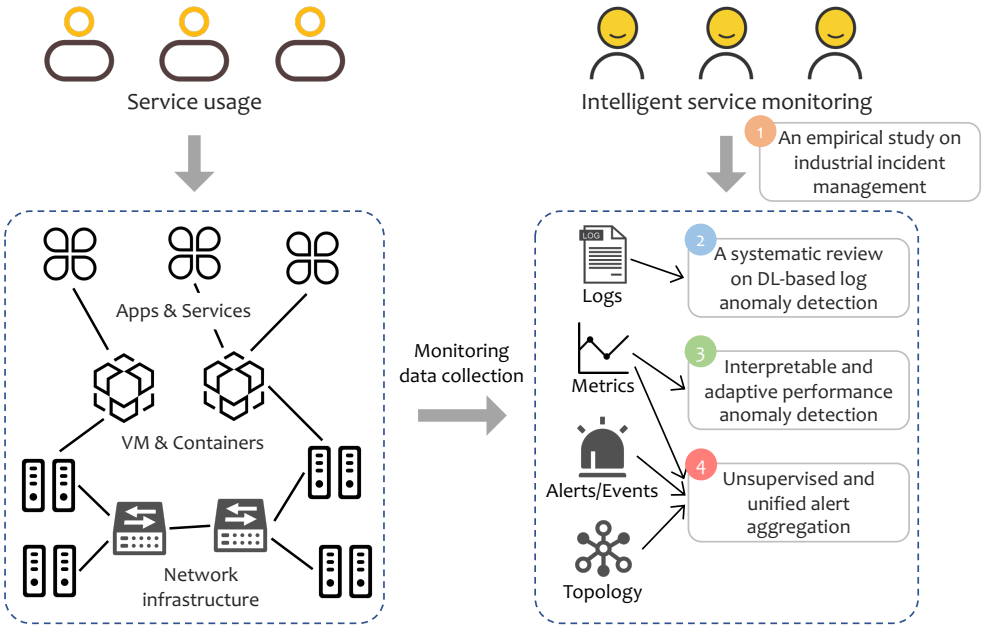


Figure 1.2: Intelligent Service Monitoring Studies

rics, and topologies) to eliminate the effort of manual labeling, making GIRDLE unsupervised. GIRDLE promotes the discovery of resource dependencies in a data-driven manner.

1.2 Thesis Contributions

In this thesis, we make the following contributions toward intelligent monitoring for online service systems.

1. **An empirical study of incident management.**

   While modern services have employed an army of support engineers to ensure their efficient operation, in practice, we still observe critical incidents that occur unexpectedly, resulting in a multiple-hour goose chase to identify the root causes. Our study aims to understand the pain points of support engineers in handling such critical incidents and propose dedicated algorithms to alleviate them. Specifically, based on over two years of incident tickets, we report the

current status of incident management at Microsoft. We study the key challenges that result in the inefficient and error-prone management workflow for the critical incidents. We also explain the challenges from the perspective of cloud system design and operation. We present representative real-world cases and statistical evidence for our findings. The insights and lessons learned from the investigation reveal some essential problems of the current data-driven service management, which we try to address in this thesis.

2. **An evaluation of DL-based log anomaly detection.**

   Log-based anomaly detection has been an essential means of service reliability monitoring. By resorting to deep learning (DL) techniques, it has achieved remarkable performance in recent years [40, 44, 189]. However, there is currently a lack of rigorous comparison among representative methods of this kind. This renders the difficulty for researchers and practitioners to choose proper methods for the problems at hand. Also, it is unclear how they can be integrated into production systems. The industry has pointed out that full-spectrum engineers with a mix of engineering and AI/ML/Data Science skills are vital for accelerating AIOps (AI for IT Operations) innovations and adoptions. Thus, we provide the first comprehensive study on DL-based log anomaly detection to help engineers easily customize and integrate end-to-end solutions into cloud services. Specifically, we conduct a systematic evaluation that benchmarks the effectiveness and efficiency of six representative DL-based log anomaly detectors and share our experience of deploying them into production systems. An open-source toolkit is released to allow easy reuse for the community.

3. **Interpretable and adaptive metric-based anomaly detection.**

When performing anomaly detection over metrics, existing methods [75, 167, 190, 139] often lack the merit of interpretability. That is, they are unable to explain what type of anomaly has happened, which requires manual effort to find out. Thus, interpretability is vital for engineers and analysts to trust the model's outputs and take remediation actions immediately. Moreover, they often lack the adaptability to effectively accommodate the ever-changing services in an online fashion. Online service systems have a vast number of components that continuously undergo updates [94]. In this case, unseen metric patterns could emerge [52], degrading the performance of the model in use. In production systems, setting a fixed threshold on metrics remains the most widely-used method for anomaly detection, which needs to consider the trade-off between false positives and false negatives. All these problems require anomaly detection to have the ability of online learning to adapt to unprecedented metric patterns.

We propose ADSketch, an interpretable and adaptive approach for service performance anomaly detection. ADSketch offers a way to characterize service performance issues using monitoring metrics. It achieves interpretability by identifying groups of anomalous metric patterns. Each of them represents a particular type of performance anomaly. The type of detected anomalies can then be immediately recognized if similar patterns show up again. New metric inputs are carefully compared to the existing patterns for online updates. Experiments conducted on both public and industrial metric data from Huawei Cloud have demonstrated our design's effectiveness.

4. **Unsupervised and unified alert aggregation.**

   Existing practices for pinpointing service system faults typ-

ically involve watching for system alerts (e.g., API call timeout, high CPU usage rate). Such methods treat alerts independently and only see a fragment of the failure. Therefore, in isolation, knowledge of these alerts is of limited utility. Support engineers need to gradually gather related alerts to recover the complete picture of the failure. Due to the cascading effect, cloud incidents often come with overwhelming alerts from dependent services and devices. This dramatically increases the difficulty of failure understanding and root cause analysis. To pursue efficient incident management, related alerts should quickly be aggregated to narrow down the problem scope. This requires leveraging more than one source of data to precisely profile the trajectory of failure propagation among services. However, existing work fails to integrate multi-source information for alert aggregation in a unified way, leading to poor performance. To address this problem, we design GIRDLE, an alert aggregation framework based on graph representation learning. GIRDLE combines alerts, metrics, and topologies to measure the behavioral similarity between services when a failure happens to search the boundary of its cascading effect. A feature vector for each unique type of alert is learned in an unsupervised and unified fashion, encoding its interactions with other alerts in temporal and topological dimensions. We conduct experiments on industrial data from Huawei Cloud to evaluate GIRDLE, showing promising results.

1.3 Thesis Organization

This remainder of this thesis is organized as follows.

- **Chapter 2: Online Service Systems and Monitoring**
  Chapter 2 scopes the problem that we try to solve and

provides context on online service systems and their monitoring. Specifically, we first brief online service systems in Section 2.1. Then, Section 2.2 introduces four types of typical IT data, i.e., logs, metrics, alerts, and topologies. Finally, in the remaining sections of the chapter, we provide some background for the intelligent service monitoring studies conducted in this thesis.

- **Chapter 3: Literature Survey on Intelligent Service Monitoring**

  Chapter 3 examines some contemporary work of online service monitoring. They are representative in the literature and closely related to ours. In particular, we discuss the shortcomings of the approaches proposed by previous work before presenting the algorithms that we develop to address them in this thesis.

- **Chapter 4: Intelligent Incident Management**

  Incident management plays a critical role in the assurance of reliability and quality of service systems. Chapter 4 discusses Microsoft's incident management practices, focusing on figuring out the challenges of handling critical failures and digging into the reasons behind them. Specifically, we first introduce the background of incident management and our contributions in Section 4.1. Then, Section 4.2 discusses our study methodology and the identified incident characteristics. Section 4.3 presents our main findings and conclusions in this study, including the key challenges of incident management and the underlying reasons. Section 4.4 introduces an AIOps framework of Microsoft for incident management. Finally, we summarize this chapter in Section 4.5.

- **Chapter 5: Deep Log Anomaly Detection**

  The industry has pointed out that the lack of the right

talents is an important challenge of data-driven service management. Support engineers should have strong engineering capabilities and mindset with enough AI/ML/Data Science skills. To help engineers with the customization and deployment of log-based service monitoring, Chapter 5 presents our systematic study on DL techniques for log anomaly detection. We first introduce the motivation and our contributions in Section 5.1. The problem formulation of log anomaly detection is summarized in Section 5.2, together with a review of six representative methods powered by neural networks. Section 5.3 talks about the experiments conducted on two widely-used datasets, containing nearly 16 million log messages and 0.4 million anomaly instances in total. We evaluate them in terms of three quality perspectives: accuracy, robustness against noise, and efficiency. In Section 5.4, we share our experience of industrial deployment and vision on promising future research directions. Finally, Section 5.5 summarizes this chapter.

- **Chapter 6: Adaptive Performance Anomaly Detection**

  Time-series anomaly detection is a traditional topic in both academia and industry. Many existing algorithms fail to gain engineers' trust due to a lack of interpretability. The ever-changing online services also demand them to self-adapt to unseen anomaly patterns. To achieve these goals, Chapter 6 introduces ADSketch, an interpretable and adaptive algorithm for metric-based performance anomaly detection. Specifically, we first introduce the main problems of existing research in this field and our main ideas to address them in Section 6.1. Section 6.2 elaborates on the algorithmic details for metric pattern extraction and how we use them to achieve interpretability. We also introduce our design of

online pattern updates. Section 6.3 presents experimental results on both public and production data from Huawei Cloud. Section 6.4 shares our success stories and some case studies. Finally, we summarize this chapter in Section 6.5.

- **Chapter 7: Graph-based Alert Aggregation**

  Online services often come with a set of alerting policies that are used to detect problems in the system. When isolated, the generated alerts only provide scattered information about failures. Chapter 7 investigates the integration of heterogeneous monitoring data for alert aggregation, which facilitates the understanding and impact scoping of failures. Specifically, we first talk about the cascading effect of service failures in Section 7.1, which raises the need for alert aggregation. We then describe the proposed framework Girdle for this end in Section 7.2, which considers the correlations between alerts in both temporal and topological dimensions. Section 7.3 evaluates the performance of Girdle based on industrial data from Huawei Cloud. We discuss our success story and lessons learned from practice in Section 7.4. Finally, Section 7.5 summarizes this chapter.

- **Chapter 8: Conclusion and Future Work**

  We conclude this thesis in Chapter 8 and talk about interesting future work. We focus especially on modeling failure propagation across different cloud layers (including services, middleware, virtual networks, and physical networks) to achieve full-stack cloud monitoring.

□ **End of chapter.**

Chapter 2

Online Service Systems and Monitoring

2.1 Online Service Systems

Online services, also called cloud services or web services, refer to a wide variety of services delivered on demand to companies and customers over the Internet. These services are designed to provide access to applications and resources in a simple and affordable way. They have been increasingly integrated into our daily lives and become indispensable, such as emails, search engines, and instant messaging apps. The internal infrastructure (e.g., physical servers and networks) of the online service systems are fully managed by cloud computing vendors and service providers. Therefore, there is no need for an enterprise to host applications on its own on-premises hardware.

As shown in Figure 2.1, there are generally three basic types of online services.

- **Software as a Service.** Software as a Service (SaaS) is the most widely recognized type of online service. This broad category encompasses a variety of services, such as web-based email, file storage and backup, and online documentation software. There are three popular types of architecture for service development, i.e., application, microservice, and

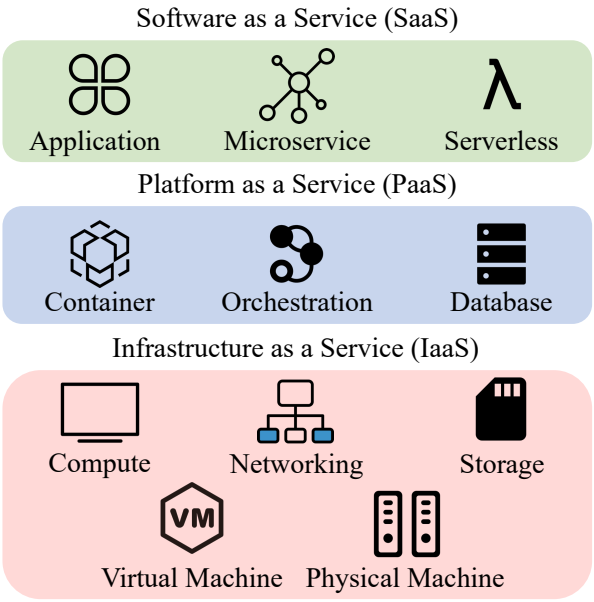


Figure 2.1: Online Service Systems

serverless function. Examples of SaaS providers include Google Suite, Microsoft Office 365, and Dropbox.

- **Platform as a Service.** Platform as a service (PaaS) is a complete development and deployment environment in the cloud, where developers can build cloud apps. PaaS provides programming languages, containers and their orchestration, database management systems, and middlewares that enable organizations to manage the complete lifecycle of cloud apps, including building, testing, deploying, managing, and updating. Many IaaS vendors, including the examples listed below, also offer PaaS capabilities.
- **Infrastructure as a Service.** Infrastructure as a service (IaaS) provides essential infrastructure (e.g., compute, networking, storage) on demand that many cloud service providers need to provision their applications. IaaS allows organizations to bypass the cost and complexity of buying and managing physical servers and data center infrastructure, and also offers load balancing and application

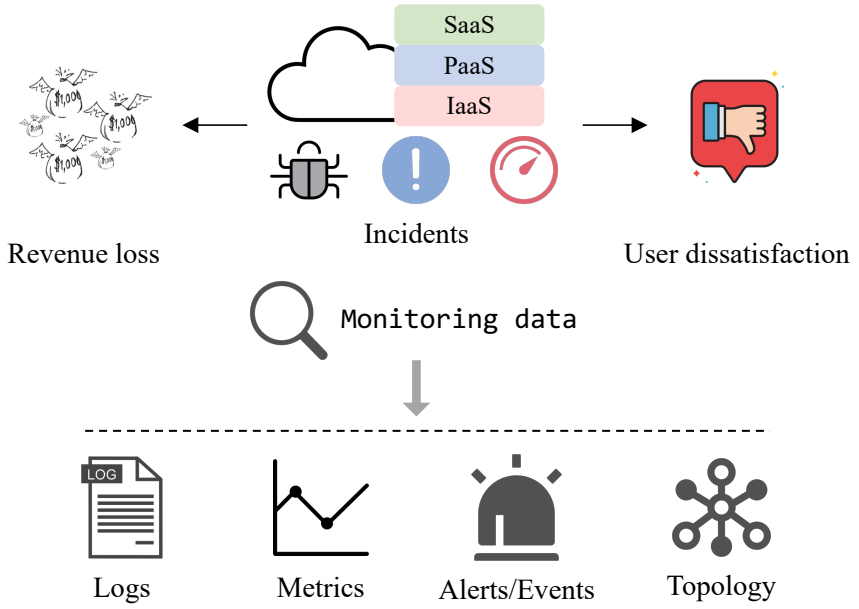


Figure 2.2: Online Service System Monitoring Data

firewall capabilities. Examples of IaaS are Amazon Web Services (AWS), Microsoft Azure, and Google Compute Engine. Many well-known SaaS providers run on IaaS platforms.

2.2 Online Service Monitoring Data

As shown in Figure 2.2, to ensure online service systems deliver a high-quality and continuous experience to customers, a variety of monitoring data are accumulated to reflect their health state from different perspectives, including:

- **Logs**. Logs are semi-structured text messages printed by logging statements in source code, composed of constant strings and variable values. Logs record critical events and operations during the runtime of the software.
- **Metrics.** Metrics are measurement of system status that are sampled uniformly, including performance counters, response time, CPU usage, memory usage, etc.

- **Alerts.** Alerts (also generally referred to as events or alarms) raise timely awareness of problems in cloud applications so they can be resolved quickly. The format of an alert contains the following information: ID, time, severity, textual message, source, etc. In cloud systems, alerts are fired by monitors (which are always-on programs) based on some policies describing the circumstances under which customers want to be alerted. Common alerting policies of monitors include checking resource status, thresholding metrics, and anomaly detection algorithms based on metrics or logs.
- **Topologies.** Topologies are graphical data in cloud systems, which have different meanings when it comes to different objects. For example, microservices depend on each other to provide and support network-based services and applications, forming service dependencies. Such relations can be retrieved from API documentation [4], configurations [7], or trace data [149, 175]. Network topology describes the physical connections among devices, such as switches, routers, and hosts. Topologies provide information regarding the relations among different cloud components, which play an important role in root cause analysis, failure impact scoping, visualization, etc. [162, 185].

By profiling service failures, these data provide clues for engineers to conduct incident management. For example, in a database failure, we may see a log message saying that the software experiences a failed connection, a latency metric showing a spike, and an alert reporting low throughput issue. The service dependencies would suggest that other functions like login and searching may also get impacted. Therefore, intelligent monitoring algorithms based on these data serve as a crucial means to promote the reliability and resilience of online service systems.

In the remaining sections, we provide some background and

challenges for the intelligent service monitoring studies conducted in this thesis. We start from the general incident management process and then introduce the pipeline of log-based anomaly detection. Next, we elaborate on the metric patterns that we observe and use for performance anomaly detection. Finally, we talk about the cascading effect of service failures, which motivates us to perform alert aggregation.

2.3 Incident Management for Online Services

2.3.1 Incidents

In cloud systems, an incident is any unplanned interruption or performance degradation of a service or product, which can lead to service shortages at all service levels, i.e., SaaS, PaaS, and IaaS. For example, a bad HTTP request, security vulnerability, resource throttling, power outage, or customer-reported error. In particular, each incident has a severity level, which is set according to its potential impact on customers. Every organization has different classification criteria, but many follow such patterns: *Low*, *Medium*, *High*, and *Critical*. For example, a data center power failure may bring down dozens of services, which should be treated as a *Critical* incident. Typically, one service relies on many supporting services, such as SQL Database and Domain Name System (DNS), to function properly. Such dependency quickly increases the chances of incidents, as any component along the dependency graph can be the source of failure.

2.3.2 Incident Management

In this subsection, we first elaborate on the typical incident management procedure and then introduce the metrics widely used to measure its performance.

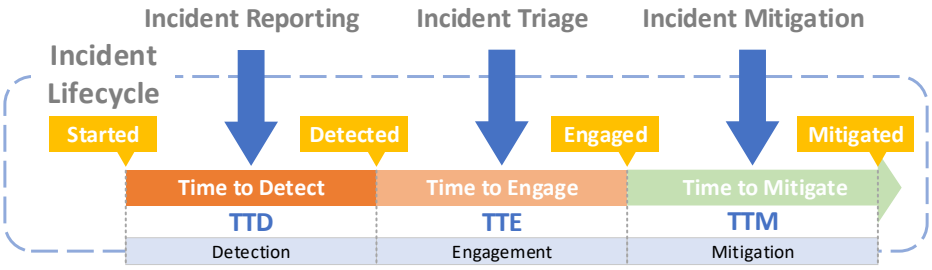


Figure 2.3: The TTx Metrics

Incident Management Procedure

Incident management is the process of detecting a live service problem, creating an incident, determining the cause, restoring the service to full operation, and capturing knowledge to prevent the incident from happening again. Typically, there are three steps involved [28, 20]: incident reporting, incident triage, and incident mitigation, as shown in Figure 2.3.

- **Incident Reporting.** Incident reporting is to detect service violations or performance degradation and create a ticket to record relevant information. In cloud systems, engineers can manually submit incident tickets if abnormal system behaviors are detected, or customer-reported failure messages are confirmed. Meanwhile, monitors can detect incidents by periodically monitoring the runtime information of service systems, e.g., software/system logs, performance counters, and process/machine/service-level events.
- **Incident Triage**. Incident triage is to engage the responsible service team for problem investigation. However, due to cloud systems' high complexity and dependencies, incidents are frequently assigned to the wrong responsible teams, significantly prolonging service downtime.
- **Incident Mitigation**. Incident mitigation is the process of bringing problematic service back to normal, so it can continue to serve customers. In practice, some temporary

workarounds (e.g., server rebooting and service redeployment) will be applied first to mitigate the impact quickly, as a short period of downtime could become an expensive drain on company revenue and user trust.

TTx Metrics of Incident Management

Incident management is critical for cloud vendors to pursue their ultimate goal: Service Level Agreement (SLA). Cloud SLA is a commitment of a cloud service provider to its customers, which guarantees a minimum level of system/service availability, reliability, and responsiveness. Similarly, objectives are also set for different phases of incident management, which are described by the TTx metrics, as shown in Figure 2.3. The goal of improving incident management is to minimize these TTx, efficiently mitigate the incident impact, and reduce operation loads.

- *Time to Detect (TTD)*: The time it takes to detect an incident from the start of its impact (SLA breached).
- *Time to Engage (TTE)*: The time it takes to engage the correct responsible service team from incident detection.
- *Time to Mitigate (TTM)*: The time it takes to mitigate customer impact and re-establish SLA (SLA re-established).

2.4 Log-based Anomaly Detection

Logs have been an imperative resource for detecting anomalies [70, 68, 194]. The overall pipeline of log-based anomaly detection is illustrated in Figure 2.4, which mainly consists of four phases, i.e., *log collection*, *log parsing*, *feature extraction*, and *anomaly detection*.

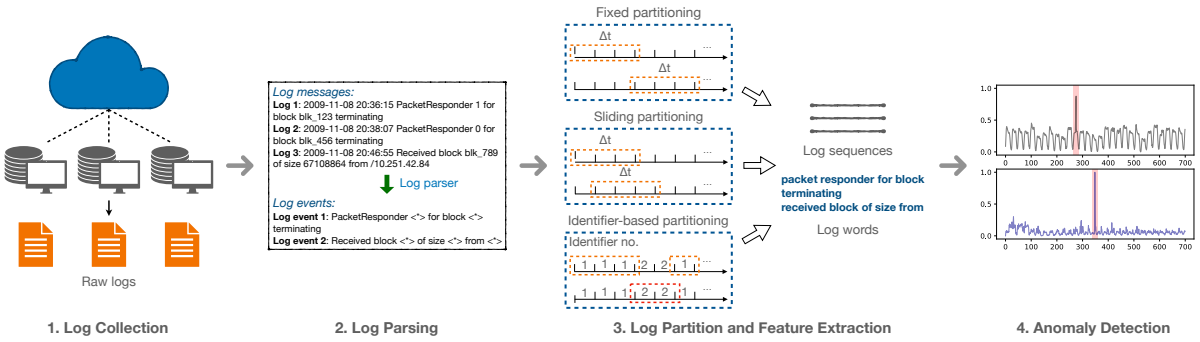


Figure 2.4: The Overall Pipeline of Log-based Anomaly Detection

2.4.1 Log Collection

Software systems routinely print logs to the system console or designated log files to record runtime status. In general, each log is a line of semi-structured text printed by a logging statement in source code, which usually contains a timestamp and a detailed message (e.g., error symptom, target component, IP address). In large-scale systems such as distributed systems, these logs are often collected. The abundance of log data has enabled a variety of log analysis tasks, such as anomaly detection and fault localization [40, 182].

2.4.2 Log Parsing

After log collection, raw logs are often semi-structured and need to be parsed into a structured format for downstream analysis. This process is called log parsing [203]. Specifically, log parsing tries to identify the constant/static part and variable/dynamic part of a raw log line. The constant part is commonly referred to as log event, log template, or log key (we use them interchangeably hereafter); the variable part stores the value of the corresponding parameters (e.g., IP address, thread name, job ID, message ID), which could be different depending on specific runs of the system. For example, in Figure 2.4 (phase two), a log excerpt collected from Hadoop Distributed File System

(HDFS) on Amazon EC2 platform [169] "*Received block blk_789 of size 67108864 from /10.251.42.84*" is parsed into the log event of "*Received block <*> of size <*> from <*>*", where all parameters are replaced with the token "<*>". Zhu et al. [203] evaluated 13 methods for automated log parsing and released a toolset.

2.4.3 Log Partition and Feature Extraction

As logs are textual messages, they need to be converted into numerical features such that ML algorithms can understand them. To this end, each log message is first represented with the log template identified by a log parser. Then, log timestamp and log identifier (e.g., task/job/session ID) are often employed to partition logs into different groups, each representing a log sequence. In particular, timestamp-based log partition usually includes two strategies, i.e., fixed partitioning and sliding partitioning.

Fixed Partitioning. Fixed partitioning has a pre-defined partition size, which indicates the time interval used to split the chronologically sorted logs. In this case, there is no overlap between two consecutive fixed partitions. An example is shown in Figure 2.4 (phase three), where the partition size is denoted as Δt. Δt could be one hour or even one day, depending on the specific problems at hand.

Sliding Partitioning. Sliding partitioning has two parameters, i.e., partition size and stride. The stride indicates the forwarding distance of the time window along the time axis to generate log partitions. In general, the stride is smaller than the partition size, resulting in the overlap between different sliding partitions. Therefore, the strategy of sliding partitioning often produces more log sequences than fixed partitioning, depending on both the partition size and stride. In Figure 2.4 (phase three), the partition size is Δt, while the stride is $\Delta t/3$.

Identifier-based Partitioning. Identifier-based partitioning sorts logs in chronological order and divides them into different sequences. In each sequence, all logs share a unique and common identifier, indicating they originate from the same task execution. For instance, HDFS logs employ *block id* to record the operations associated with a specific block, e.g., allocation, replication, and deletion. Particularly, log sequences generated in this manner often have varying lengths. For example, sequences with a short length could be due to early termination caused by abnormal execution.

After log partition, many traditional ML-based methods [70] generate a vector of log event count as the input feature, where each dimension denotes a log event, and the value counts its occurrence in a log sequence. Different from them, DL-based methods often directly consume the log event sequence. In particular, each element of the sequence can be simply the index of the log event or a more sophisticated feature such as a log embedding vector. The purpose is to learn logs' semantics to make more intelligent decisions. Specifically, the words in a log event are first represented by word embeddings learned with *word2vec* algorithms, e.g., FastText [80] and GloVe [129]. Then, the word embeddings are aggregated to compose the log event's semantic vector [189].

Then, the word embeddings are aggregated to compose the log event's semantic vector, denoted as $\mathcal{V}$. In this process, term frequency-inverse document frequency (TF-IDF) can be applied to compute the importance of words in log events. For a target word, its TF-IDF weight w is $TF(word) \times IDF(word)$, where $TF(word) = \frac{\#word}{\#total}$, $\#word$ is the number of the target word in a log event, $\#total$ is the number of words in the log event, $IDF(word) = log(\frac{\#E}{\#E_{word}})$, $\#E$ is the number of all log events, and $\#E_{word}$ is the number of log events containing the target word. Finally, the semantic vector of the log event can be

calculated as $\mathcal{V} = \frac{1}{N}\sum_{i=1}^{N} w_i \cdot v_i$, where N is the number of words in the log event, w_i and v_i are the weight and word vector for the $No.i$ word.

2.4.4 Anomaly Detection

Based on the log features constructed in the previous phase, anomaly detection can be performed, which is to identify anomalous log instances (e.g., logs printed by interruption exceptions). Many traditional ML-based anomaly detectors [70] produce a prediction (i.e., an anomaly or not) for the entire log sequence based on its log event count vector. Different from them, many DL-based methods first learn normal log patterns and then determine the normality for each log event. Thus, DL-based methods are capable of locating the exact log event(s) that contaminate the log event sequence, improving interpretability.

2.5 Metric-based Performance Anomaly Detection

2.5.1 Performance Anomaly Patterns

In online service systems, a large number of metrics are configured to monitor various aspects of both logical resources (e.g., a virtual machine) and physical resources (e.g., a computing server). Cloud systems often possess an abundance of redundant components, providing the ability of fault tolerance and self-healing (e.g., load balancing and availability zones). Consequently, the majority of service breakdowns tend to manifest themselves as performance anomalies first instead of fail-stop failures [74, 105]. We observed that when performance anomalies of similar types happen, their impacts tend to trigger similar reactions/symptoms on the metric time series, which we refer

to as metric patterns. For example, a level shift up on Interface Throughput may indicate slow service response, which could be caused by a load balancing failure; a level shift down on it may suggest service unavailability, and the culprit could be performance bugs (e.g., memory leak bugs). Similar observations have been made in [109, 43]. The rationale behind such a phenomenon is twofold. First, the design of the metrics is sophisticated and fine-grained, each dedicated to monitoring a specific problem, e.g., request timeout or high API error rate. Second, cloud systems widely employ the microservices architecture, where cloud applications employ lightweight container deployment, e.g., cloud-native applications and serverless computing. With this architecture, each microservice is designated for well-defined and modularized jobs, e.g., user login and location service. Thus, they tend to develop individual and stable patterns, which can manifest through their monitoring metrics.

2.5.2 Metric Pattern Mining

Metric patterns (i.e., repetitive time-series subsequences describing the misbehaving moments of metrics) can be leveraged to sketch the performance issues for anomaly detection. This is essentially profiling the mode of recurrent anomalies. For example, hardware failures often come with a sudden drop in the corresponding metrics, and the value remains zero for some time. If anomalies come into existence, they can be immediately identified by matching the established patterns. Such metric patterns can also facilitate problem mitigation. For example, when low service throughput and high CPU usage are detected, engineers can scale up the microservice (by adding local cores) to increase its capacity. The key challenge is how to automatically discover what anomalous patterns a metric time series has experienced. For each identified pattern, engineers can label the typical per-

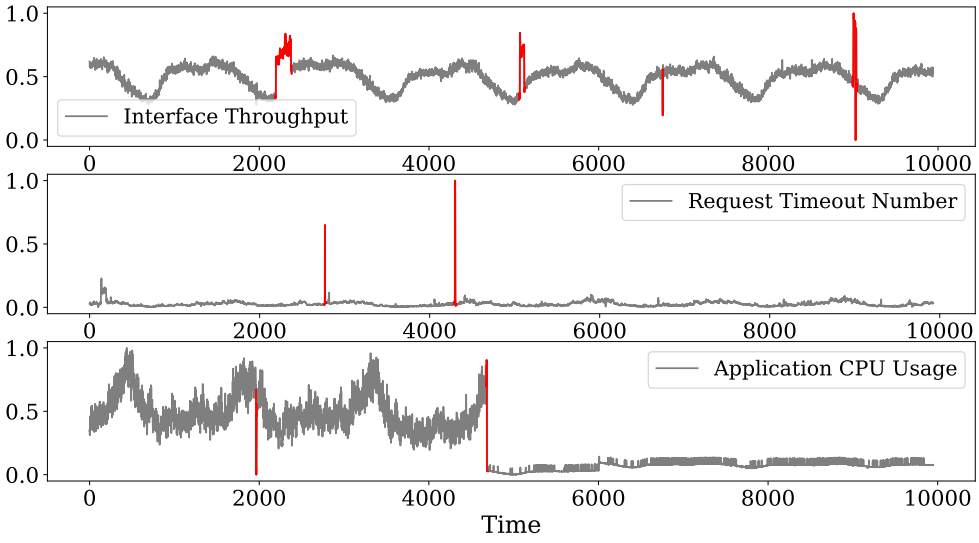


Figure 2.5: Examples of Performance Anomaly Patterns

formance issues it often associates with. In online scenarios, if a metric encounters any known anomalous patterns, the underlying performance issues can be recommended. Pattern sketching thus provides a means to accumulate and utilize engineers' knowledge.

In real-world scenarios, the patterns exhibited in metrics are extremely complicated and can have numerous variants in terms of scale, length, and combination. We have identified the following challenges for metric pattern discovery, which are illustrated in Figure 2.5. Each metric time series records around one week of monitoring data, whose anomalies are shown in red.

- **Background noise.** Although a large amount of metric time series is generated, a significant portion of them is trivial, recording only plain system runtime behaviors. Moreover, due to the dynamics of online services, some metrics may experience concept drift [52]. For example, the Application CPU Usage in Figure 2.5 drops abruptly, which could be caused by a role switch (e.g., from a primary node to a backup node) or user behavior change. How to distinguish anomalous patterns from normal ones is non-trivial.

- **Pattern variety.** A metric curve can possess multiple distinct patterns simultaneously. For example, in Figure 2.5, the Interface Throughput has two anomaly patterns, i.e., spike up and spike down. Also, the patterns can have different scales, as indicated by the two spikes in the Request Timeout Number. We need to consider the context of each metric for pattern extraction.
- **Varying anomaly duration**. Different performance issues may vary in duration. The first two anomalies in the Interface Throughput constitute such an example. Particularly, how long an anomaly lasts is also an important factor that engineers rely on to understand a service's health state. When characterizing the issues, such a fact should be considered appropriately.

2.6 Alert Aggregation for Online Services

2.6.1 Topologies in Large-scale Cloud Systems

Cloud vendors provide a variety of online services to customers worldwide, which often possess a hierarchical architecture, i.e., service (SaaS), platform (PaaS), and infrastructure (IaaS) layers. Different layers have different forms of topologies. In the SaaS layer, each service embodies the integration of code and data required to execute a complete and discrete functionality. Different services communicate through virtual networks using protocols such as HTTP and Remote Procedure Call (RPC). Service A is said to be dependent on service B if A needs to call B and requires the response to proceed with its operations. A call failure describes the case when a service calls another service but receives no or incorrect response. Such communications among services constitute the complex topology of online service systems [112, 98, 200, 136, 53]. The service dependency topology

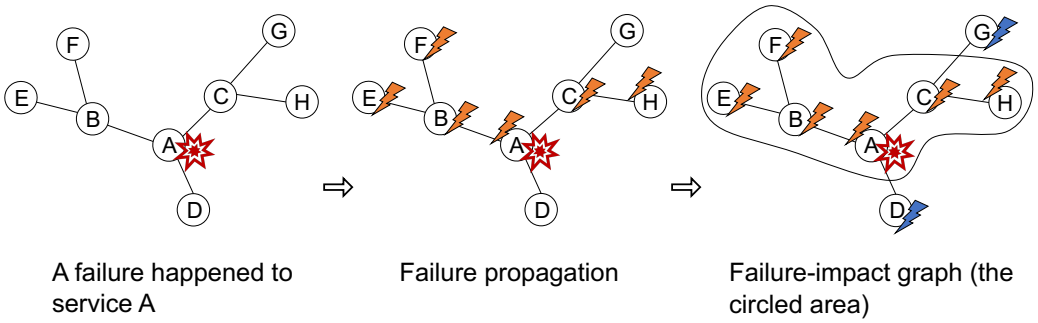


Figure 2.6: An Illustration of Service Failures' Cascading Effect (The circled area in the third subfigure shows the failure-impact graph.)

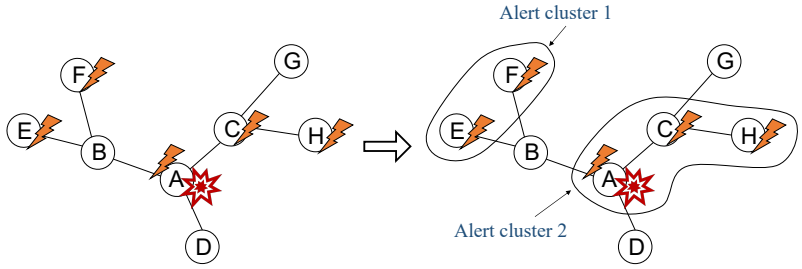


Figure 2.7: An Example of Incomplete Failure-impact Graph

can be built by parsing the REST documentation of services (in JSON format generated by Swagger[1] or RAML[2]) [4, 160] or analyzing the trace/invocation records between services [175, 149]. Some popular open-source microservice benchmarks, e.g., Train Ticket[3] and Online Boutique[4], directly expose the dependencies among its components. Another example is the cloud network topology in the IaaS layer, which includes the physical connections between different types of network devices (e.g., switches and servers) [156, 143] and the hosting relation between servers and Virtual Network Function (VNF) instances [137, 42]. These dependencies can be retrieved from the Configuration Management Database (CMDB) of cloud systems.

[1] `http://swagger.io`
[2] `http://raml.org`
[3] `https://github.com/FudanSELab/train-ticket`
[4] `https://github.com/GoogleCloudPlatform/microservices-demo`

2.6.2 Cascading Effect of Service Failures

Due to service dependencies, a failure occurring to one service tends to have a cascading effect across the entire system. Representative service failures include slow response, request timeout, service unavailability, etc., which could be caused by capacity issues, configuration errors, software bugs, hardware faults, etc. To quickly understand failure symptoms, a large number of monitors are configured to monitor the states of different services in a cloud system [29]. A monitor will render an alert when certain predefined conditions (e.g., CPU usage exceeds 80%) are met. When a failure happens, the monitors often produce a large number of alerts. These alerts are triggered by the common root cause and describe the failure from the perspectives of different services. Thus, they can be aggregated to help engineers understand and diagnose the failure.

We model the set of alerts triggered by a failure as its impact graph (i.e., failure-impact graph). As illustrated by Figure 2.6, service A encounters a failure, and the impact propagates to other services along the system topology. The circled area indicates the impact graph of the failure, where irrelevant alerts (in blue) in service D and G are excluded. These alerts are collected during the failure period, which can be determined by failure detection algorithms. In this period, failure can manifest its cascading effect completely. We do not impose stringent requirements on the clock synchronization among the distributed services, because the correlation between alerts can be learned appropriately if they show up during the failure (even if their local clocks drift slightly). Intuitively, it might seem that the impact graph can be easily constructed by tracing alerts along the topology. However, our industrial practices reveal that they are usually incomplete. An example is given in Figure 2.7, where service B occasionally fails to report any alert during the failure. Existing approaches may perceive it as two separate failures, which is undesirable.

We have summarized two main reasons for the missing alerts:

- System monitors that report alerts are configured with rules predefined by engineers. Due to the diversity of cloud services and user behaviors, the impact of a failure may not meet the rules of some monitors. For example, if an application triggers an alert when its CPU usage rate exceeds 80%, then any value below the threshold will be unqualified. As a consequence, the monitors will not report any alerts. Thus, the tracking of the failure's impact is blocked.
- To ensure continuity, cloud systems are designed to have a certain fault tolerance capability. In this case, service systems can bear some abnormal conditions, and thus, no alerts will be reported. Therefore, the impact of a failure may not manifest itself entirely over the system topology.

Recent studies on cloud incident management [29, 74] have demonstrated the incompleteness and imperfection of monitor design and distribution in online service systems. Thus, along the service dependency chain, some services in the middle may remain silent (i.e., report no alert), which impedes tracking failure's cascading effect. Therefore, although online service systems generate many alerts, they are often scattered.

□ **End of chapter.**

Chapter 3

Literature Survey on Intelligent Service Monitoring

In this chapter, we review existing work of intelligent monitoring for online service systems. We highlight their major limitations that we try to address in this thesis. Following the structure of the previous chapter, we start with the overall incident management of cloud services and then discuss related work on logs, metrics, and alerts.

3.1 Intelligent Incident Management

3.1.1 Reliability and Resilience of Online Services

There are many methods focusing on improving the reliability and resilience of cloud systems. Regarding failure prediction in cloud systems, Xu *et al.* [170] formulated the disk failure prediction problem (a major source of service incidents) as a ranking problem and adopted the FastTree algorithm to make predictions. He *et al.* [69] proposed a cascading clustering algorithm to identify the impactful problems by correlating the clusters of log event sequences with system KPIs. Zhang *et al.* [187] addressed the problem of general bug management in software systems. Hu *et al.* [73] automated this process by constructing a developer-

component-bug network, which models the relationship among developers, source code components, and the associated bugs.

Some approaches have been proposed to address the service dependency issues for cloud services. For example, Bahl *et al.* [7] presented Leslie Graph, an abstraction describing the complex dependencies between network, host, and application components in networked systems. In particular, they systematically discussed the challenges and difficulties of discovering service dependencies. Chen *et al.* [23] introduced Orion, a system that searches dependencies using packet headers and timing information in network traffic.

3.1.2 Empirical Study on Incident Management

Recently, cloud service incidents and their management are gaining more and more popularity. For example, Zhou *et al.* [197] performed an empirical study on the quality issues of a big data computing platform. They analyzed 210 real service quality issues and investigated their common symptoms, causes, and mitigation solutions. Their findings show that 21.0% of major issues encountered by customers are caused by hardware faults, 36.2% are caused by system-side defects, and 37.2% are due to customer-side faults. Huang *et al.* [74] studied the gray failures in production cloud-scale systems. They found this type of failure is hardly noticed by the system's failure detectors even when applications are afflicted by them. Chen *et al.* [20] studied the incident triage problem on 20 large-scale online service systems at Microsoft. Their results reveal the fact that incorrect assignment of incident reports frequently occurs, especially for incidents with high severity. Dang *et al.* [37] summarized the real-world challenges in building AIOps solutions and proposed a roadmap of AIOps-related research directions. Gunawi *et al.* [58] conducted a cloud outage study of 32 popular Internet

services. They provided answers to why outages still take place in cloud environments by analyzing 1,247 headline news and public postmortem reports which detail 597 unplanned outages.

While these papers only study one specific aspect of cloud systems' incident management, we present a comprehensive characterization of it. Moreover, we try to understand the reasons behind the challenges of incident management from the internal system design and operation principles.

3.2 Log-based Anomaly Detection

3.2.1 Log analysis

Logs have become imperative in the assurance of software systems' reliability and continuity because they are often the only data available that record software runtime information. Typical applications of logs include anomaly detection [70, 169, 40], failure prediction [146, 147], failure diagnosis [182, 200], etc. Most log analysis studies involve two main steps, i.e., log parsing and log mining. Based on whether log parsing can be conducted in a streaming manner, log parsers can be categorized into offline and online. Zhu et al. [203] conducted a comprehensive evaluation study on 13 automated log parsers and reported the benchmarking results. More recently, Dai et al. [36] proposed an online parser called Logram, which considers the n-grams of logs. The core insight is that frequent n-grams are more likely to be part of log templates.

Many efforts have also been devoted to log mining, especially anomaly detection due to its practical significance. They can be roughly categorized into two classes, i.e., traditional ML-based and DL-based methods. For example, Xu et al. [169] were the first to apply PCA to mine system problems from console logs. By mining invariants among log messages, Lou et al. [106] detected

system anomalies when any invariants were violated. Lin et al. [101] proposed LogCluster, which recommends representative log sequences for problem identification by clustering similar ones. He et al. [69] proposed Log3C to incorporate system KPIs into the identification of high-impact issues in service systems. Some work [50, 120] employs graph models such as finite state machines and control flow graphs to capture a system's normal execution paths. Anomalies are alerted if the transition probability or sequence violates the learned graph model.

In recent years, there has been a growing interest in applying neural networks to log anomaly detection. For example, Du et al. [40] proposed DeepLog, which is the first work to adopt an LSTM to detect log anomalies in an unsupervised manner. Meng et al. [114] proposed LogAnomaly to extend their work by incorporating logs' semantics. To address the issue of log instability, i.e., new logs may emerge during system evolution, Zhang et al. [189] proposed a supervised method called LogRobust, which also considers logs' semantic information. Wang et al. [173] addressed the issue of insufficient labels via probabilistic label estimation and designed an attention-based GRU neural network. Lu et al. [108] explored the feasibility of CNN for this task. Other models include the Transformer [122], LSTM-based generative adversarial network [166], etc.

3.2.2 Empirical studies on logs

Empirical studies are also an important topic in the log analysis community, which derives valuable insights from abundant research work in the literature and industrial practices. For example, Yuan et al. [183] studied the logging practices of open-source systems and provided developers with suggestions for improvement. Fu et al. [51, 202] focused on the logging practices on the industry side. The work done by He et al. [70] is the most related

study to ours, which benchmarks six representative log anomaly detection methods proposed before 2016. Different from them, we focus on the latest DL-based approaches and investigate more practical issues, such as unprecedented logs in testing data and inevitable anomalies in training data. Le et al. [87] followed our work to explore more characteristics of DL-based log anomaly detectors. More recently, Yang et al. [174] presented an interview study of how developers use execution logs in embedded software engineering, summarizing the major challenges of log analysis. He et al. [68] conducted a comprehensive survey on log analysis for reliability engineering, covering the entire lifecycle of logs, including logging, log compression, log parsing, and various log mining tasks. Candido et al. [15] provided a similar review for software monitoring. Other studies include cloud system attacks [82], cyber security applications [86], etc.

3.3 Metric-based Anomaly Detection

Performance anomaly detection on time series has been a hot topic. Monitoring metrics used to profile the runtime status of a system are usually denoted as multiple univariate time series. In the literature, anomaly detection methods on time series can be categorized into statistical, traditional machine learning, and deep learning approaches.

3.3.1 Statistical and ML-based Approaches

In industry, Autoregressive Moving Average Model (ARMA) [17] remains the most popular statistical method to detect apparent anomalies in univariate time series. To capture complex anomalous patterns, Ma et al. [109] summarized several type-oriented patterns from the metrics of cloud databases to diagnose the performance degradation in associated online services.

More complex pattern recognition methods utilize machine learning-based models. For example, unsupervised clustering methods can be used to detect anomalous points in time-series data. Like our work, Pang et al. [126] proposed a clustering-based statistical model called LeSiNN to detect anomaly patterns from history. However, it is not robust in real industry practices due to complicated parameter tuning. With the assumption that anomalous data should be in smaller numbers and isolated from an extensive number of normal observations, Isolation Forest (iForest) [102] employs multiple binary trees to distinguish anomalies in non-linear space. Extreme Value Theory (EVT) [150] studies the hidden state of a random variable around the tails of its distribution to adaptively enhance the performance of many statistical and machine learning methods. However, EVT heavily relies on hyperparameter tuning.

3.3.2 Deep Learning-based Approaches

In recent years, there has been an explosion of interest in applying neural networks to metric-based anomaly detection, which have been demonstrated to achieve better performance. For example, Zong et al. [206] proposed a deep autoencoding Gaussian mixture model (DAGMM) to detect anomalous data points from each observed data without considering the temporal dependencies in time series. To detect complex anomalies in spacecraft monitoring systems, LSTM-NDT [75] leverages Long Short-Term Memory (LSTM) networks with nonparametric dynamic thresholding to pursue interpretability throughout the systems. Zhao et al. [190] and Lin et al. [99] also employed LSTM to predict performance anomalies in software systems. Inspired by the Spectral Residual algorithm in other domains, Ren et al. [139] proposed SR-CNN to detect anomalies from seasonal metric data for large-scale cloud services, which contain the periodic recurrence of

fluctuations. DONUT [167] designs an unsupervised anomaly detection method based on the Variational Auto-Encoder (VAE) framework to detect anomalies from low-qualified seasonal metric time series with various patterns. DONUT provides a theoretical explanation compared to other deep learning methods. LSTM-VAE [127] combines LSTM networks and the VAE framework to reconstruct the probability distribution of observed data in time series. However, LSTM-VAE ignores the temporal dependencies in time series. OmniAnomaly [153] learns the normal patterns using a large collection of historical data. The anomalous patterns are located from the large margin of reconstruction loss to the normal patterns.

The aforementioned deep learning-based methods usually follow an end-to-end style and play as a black box inside. Due to poor interpretability, the detection results cannot provide engineers with actionable suggestions for fault diagnosis. Furthermore, all these methods have difficulties handling unseen metric patterns brought by the frequent updates of online services.

3.4 Alert Management in Cloud Systems

3.4.1 Problem Identification and Diagnosis

To provide high-quality online services, many researchers have conducted a series of investigations, including problem identification and diagnosis from runtime log data and alerts [101, 69, 78]. For example, to identify problems from a large volume of log data, Lin et al. [101] proposed LogCluster to cluster log sequences and pick the center of each cluster. Inspired by LogCluster [101], Zhao et al. [191] clustered online service alerts to identify the representative alerts to engineers. Different from the clustering techniques, Jiang et al. [78] proposed an alert prioritization ap-

proach by ranking the importance of alerts based on the metrics in alert data. The top-ranked alerts are more valuable for identifying problems. However, this approach has a limited scope of application because it is only practical to metric alerts generated from manually defined threshold rules. To conduct problem identification more aggressively, Chen et al. [26] proposed an incident diagnosis framework to predict general incidents by analyzing their relationships with different alerting signals. Zhao et al. [192] considered a more practical scenario where there are plenty of noisy alerts in online service systems. They proposed eWarn to filter out the noisy alerts and generate interpretable results for incident prediction. Wang et al. [162] constructed a real-time causality graph based on alerts for root cause analysis in industrial settings. To incorporate domain knowledge, they allow customized rules for graph construction. Such a design facilitate the rapid requirement changes in microservices.

3.4.2 Intelligent Alert Management

In recent years, alert management has become a hotspot topic in both academia and industry. Massive amount of effort has been devoted to alert detection [99, 192, 57, 97] and alert triage [57, 54, 20, 21]. For example, Lim et al. [96] utilized Hidden Markov Random Field for performance issue clustering to identify representative issues. Chen et al. [21] proposed DeepCT, a deep learning-based approach that is able to accumulate knowledge from alert discussions and automate alert triage. However, due to high manual examination costs, these methods cannot handle the overwhelming number of alerts. Many existing work [97, 168, 191, 193] addresses this problem by reducing the duplicated or correlated alerts. For example, Zhao et al. [193] aimed to recommend severe alerts to engineers. Lin et al. [97] proposed an alert correlation method to cluster semi-structured

alert texts to gain insights from the clustering results.

Similar to our method, Zhao et al. [191] conducted alert reduction by calculating their textual and topological similarity. The centroid alert of each cluster is then selected as the representative alert to engineers. Specifically, they first leveraged conventional methods to detect alert storms and the associated anomalous alerts and then adopted DBSCAN [41] to cluster alerts based on their textual and topological similarity. Another similar work is LiDAR proposed by Chen et al. [25], which links relevant alerts by incorporating the representation of cloud components. Their framework consists of two modules, a textual encoding module and a component embedding module. The first module learns a representation vector for alert descriptions in a supervised manner. The textual similarity between two alerts is measured by the cosine distance of their representation vectors. Similarly, the second module learns a vector for system components. The final similarity is calculated by leveraging two parts of information. However, these methods employ a simple weighted sum to combine the information from different sources and still hardly capture the relationship between alerts. Differently, our method utilizes sophisticated graph representation learning to obtain the semantic relationship of alerts from diverse sources, including temporal locality, topological structure, and metric data. Moreover, many existing alert management methods rely on supervised machine learning techniques to detect anomalies or conduct alert triage. More intelligent approaches with weak supervision or even unsupervised frameworks are still largely unexplored, which is addressed in our design.

□ **End of chapter.**

Chapter 4

Intelligent Incident Management

Once a service incident occurs, the service provider should immediately take action to diagnose the problem and bring the service back to normal, which is called *incident management*. Service providers have invested significant efforts in incident management to minimize service downtime and to ensure the high quality of the provided services. In this chapter, we present the practices of Microsoft in this direction. The remainder of this chapter is organized as follows. Section 4.1 provides the problem background and contributions of this study. Section 4.2 discusses our study methodology and the identified characteristics of incidents. Section 4.3 presents the key challenges of incident management and the underlying reasons. Section 4.4 presents the AIOps (Artificial Intelligence for IT Operations) framework for incident management at Microsoft. Finally, Section 4.5 summarizes this chapter.

4.1 Problem and Contributions

In practice, a typical incident management procedure goes as follows: when engineers or machine-based monitors detect a service incident, an incident ticket documenting relevant information

will be created in the incident management system. Based on the ticket, the incident will be assigned to the responsible service team to restore the service quickly.

Ideally, if the entire procedure of incident management goes smoothly, the service can be quickly recovered. With years of effort, Microsoft is now capable of alerting 97% of incidents automatically and controlling more than 90% of incidents by immediate mitigation. However, there are still severe and complex incidents that take a long time to handle. Our investigation reveals that the delay happens mainly in the following three scenarios. First, it is not rare that critical incidents cannot be immediately detected as they are often induced by unexpected system/customer behaviors. Second, failure's symptoms are sometimes hardly enough to directly pinpoint the responsible service team, as distinct problems could induce similar symptoms. Incidents are therefore reassigned multiple times. Third, engineers usually need a long time to identify the incident's root cause and the corresponding impact scope, i.e., impacted services and customers. Motivated by these observations, we conduct an empirical study to understand the key challenges bringing about the delay in these steps as well as the fundamental reasons behind them.

Microsoft runs worldwide cloud systems with thousands of services. Such a large scale makes it challenging to conduct incident management. We have summarized two critical challenges: (1) building dependencies among massive services/resources and (2) assessing the health state of numerous cloud resources to fire reasonable alerts. In large cloud enterprises, the performance and reliability of any particular application may rely on multiple services and resources, spanning many hosts and network components [119]. The dependency issue refers to the incompleteness and vagueness of such relationships across the entire system. Consequently, the culprits of the incident cannot be easily found.

Even provided with the dependencies, we still need to assess the health condition of resources to locate root causes. To help understand these issues, we carefully select some real-world counterintuitive incident cases to illustrate different types of pain points and why heuristic solutions would fail. Meanwhile, we present four exciting lessons learned. For example, while root cause localization stands as the core of impactful incident mitigation, addressing all impacted services could be equally important; a flood of alerts during impactful incidents is inevitable even if careful aggregations and tuned thresholds have been applied. In addition, we quantitatively analyze the incidents collected from six core large-scale services at Microsoft and conduct a series of experiments to derive statistical support for our findings.

Given that incident management is data-driven by nature, the concept of AIOps was proposed to address the challenges of IT operations with AI techniques [76, 37, 28]. We have seen its great potential in extracting patterns from recurrent incident symptoms to provide actionable recommendations. We present IcM BRAIN (BRAIN for short), our AIOps framework for incident management. First, we introduce different types of data utilized in the framework and the data preprocessing procedure. Then, we elaborate on the techniques for mitigating the aforementioned challenges. Finally, we share the application results to demonstrate the industrial benefits conveyed to the incident management of Microsoft.

4.2 Incident Ticket Analysis

4.2.1 Methodology

Raw Dataset

Microsoft provides thousands of online services running on a 24/7 basis. To ensure service reliability and availability, cloud

systems incorporate sophisticated monitoring mechanisms, where each monitor is tailored for a specific type of service-affecting symptom. The entity over which to perform health evaluation, monitoring, and alerting is called a *resource*, which can be a physical resource like a computer device or a logical resource like a virtual machine. Upon a violation of any predefined performance metric (e.g., availability and latency), the corresponding monitor will render an incident ticket with the timestamp, location, severity, involved services/teams, impacted resources/components, a title briefly describing the symptom, etc. Besides auto alerts, manual reporting (i.e., detected by customers or engineers) is another important source of incidents, in which an extra text snippet summarizing the cloud issue is included. During problem investigation, discussions conducted by On-Call Engineers (OCEs) will be continuously added to the ticket.

We have studied incidents from all services over two years. Among them, we select six core services at Microsoft, namely, Datacenter Management (DCM), Network, Storage, Compute, Database, and Web Service (WS), which are known as the fundamental basis that thousands of other services rely on. In this chapter, we report our findings by analyzing these services' incident tickets. We exclude the incidents that are intentionally generated for testing purposes. For over two years of operations, these core services have produced a large number of incidents and almost half of the impactful incidents at Microsoft. Due to privacy and security reasons, we do not release the dataset. Nevertheless, public incident tickets [58, 151] can help readers understand the main characteristics of our dataset. They both record cloud system failures and share certain similarity. At the same time, our dataset paints a more comprehensive landscape of the incidents in large-scale cloud systems and provides more details that are not publicly-available. These advantages allows us to conduct an in-depth analysis of incident management.

Study Approaches

After collecting incident tickets, we perform the following investigations to derive insights:

1) *Incident ticket analysis.* We calculate the distribution of incidents along multiple dimensions (e.g., severity and root cause) to obtain a clear view of their characteristics. Moreover, we manually study the impactful incidents as well as their postmortem reports to understand the issues of incident management that constitute the unique challenges of troubleshooting in cloud systems.

2) *Field studies.* Besides statistical analysis, we discuss with OCEs to collect first-hand information regarding the pain points of incident handling and empirically verify our hypotheses. In this process, we acquire much valuable feedback and suggestions, e.g., the selection of representative incident examples (Section 4.3.1).

3) *Validation.* To support our findings, we design dedicated experiments to obtain statistical evidence from the collected incidents. Moreover, to validate the effectiveness of our AIOps framework in production systems, we perform non-parametric hypothesis testing on incidents with and without BRAIN support.

4.2.2 Characteristics of Incident Tickets

Incident Severity

Table 4.1 shows the distribution of incident severity among six services. We can see that in all services, the *Low* and *Medium* incidents together take up more than 90% of the total. In Network and Storage, the numbers of these two types of incidents are similar, while in others, *Medium* incidents outnumber *Low* incidents by a substantial margin. The number of *High* incidents drops significantly, whose proportion ranges from 1.21% (Network) to 5.48% (DCM). Finally, incidents of *Critical* type account for a very small portion, i.e., $< 0.5\%$. However, such

incidents constitute a significant threat to the SLA of cloud vendors and, thus, should be addressed promptly and carefully.

Incident Fixing Time

We calculate incident fixing time and denote it as Time to Fix (TTF). Formally, it is defined as the time from the start of an incident to its final mitigation, i.e., $TTF{=}TTD{+}TTE{+}TTM$. In particular, to ignore infrequent peaks, we report the 90th percentile, which is chosen empirically. The results are shown in Table 4.2. Due to privacy concerns, we conceal the absolute results by dividing them by the smallest figure obtained in each experiment. A counter-intuitive observation is that the TTF of incidents with a lower severity (i.e., *Medium* and *Low*) is usually larger than that of incidents with a *High* severity. We find it is because low-severity incidents are usually trivial issues. Engineers will not address them immediately as they often could be mitigated by automatic routines. Meanwhile, except in Network, *Critical* incidents are always the most time-consuming incidents to mitigate. One reason is that the respective root causes (such as wrong configurations, software bugs, faulty devices, etc.) of *Critical* incidents will be fixed soon after postmortem analysis, and most of them will never re-occur. Every new *Critical* incident is likely to carry a brand-new failure, so OCEs need a long time for root cause identification and mitigation. Moreover, there exist hierarchical dependencies among these services. DCM is in charge of infrastructure maintenance and thus is a service at the lowest layer. On top of DCM are Network and then Storage, which are also fundamental services. Compute belongs to the next tier, followed by Database and WS, which have complicated dependencies on low-layer services. Therefore, high-layer services (i.e., Compute, Database, and WS) may have hierarchical root causes. The increased problem search space leads to a longer TTF.

Table 4.1: Distribution of Incident Severity

	DCM	Network	Storage	Compute	Database	WS
Critical	0.01%	0.01%	0.01%	0.31%	0.40%	0.07%
High	5.48%	1.21%	2.57%	5.27%	4.32%	3.33%
Medium	86.65%	46.90%	43.32%	74.19%	63.93%	84.52%
Low	7.86%	51.88%	54.10%	20.23%	31.35%	12.08%

Table 4.2: Distribution of Relative Incident Fixing Time

	DCM	Network	Storage	Compute	Database	WS
Critical	**38.33x**	8.46x	**10.06x**	**142.05x**	**209.97x**	**286.6x**
High	19.25x	3.18x	2.52x	2.56x	5.75x	3.56x
Medium	1x	**9.8x**	7.09x	2.95x	25.28x	12.93x
Low	3.01x	5.49x	1.09x	11.65x	2.41x	144.79x

Table 4.3: Distribution of Incident Root Causes

Root Cause	Dist.	Root Cause	Dist.
Network (Hardware)	**22.95%**	Human Error (Code Defect)	**19.23%**
Network (Connectivity)	2.24%	Human Error (Config.)	**7.45%**
Network (Config.)	0.89%	Human Error (Design Flaw)	5.66%
Network (Other)	4.47%	Human Error (Integration)	2.09%
Deployment (Upgrade)	5.22%	Human Error (Other)	2.83%
Deployment (Config.)	3.87%	External Issue (Partner)	2.83%
Deployment (Other)	1.19%	External Issue (Other)	1.64%
Capacity Issue	**6.56%**	Others	10.88%

Root Causes

To have an in-depth analysis of the incidents, we need to understand the reasons for their occurrence, which also helps characterize the failure patterns of cloud systems. Thus, we manually inspect the *Critical* incidents along with their postmortem reports and summarize their root causes into different categories. The results are shown in Table 4.3. We group incident root

causes into six categories, which are *Network Issue*, *Human Error*, *Deployment Issue*, *External Issue*, *Capacity Issue*, and *Others*. Furthermore, we summarize them into 16 subcategories. From Table 4.3, we can see that Network Issue with hardware failures and Human Error with code defects are the two dominant root causes, accounting for 22.95% and 19.23%, respectively. Meanwhile, Human Error with configuration issues and Capacity Issue are another two important causes.

4.3 Incident Management Understanding

In the management of high-impact incidents, we have observed an inefficient workflow of the system. Particularly, we have seen cases where a small-scale issue in one service yielded a more severe impact across multiple services before its official declaration. A natural problem then arises: why is the issue not detected in the first place? In the incident triage phase, we have noticed that incidents often require a long routine to find the correct responsible team, especially for incidents with high-level severity. Similar pain points can be seen in the incident mitigation phase. The connection between issue, cause, and impact can sometimes take a long time to establish.

In this section, we first summarize the key challenges that lead to the aforementioned pain points of incident management. Then, we investigate the reasons behind these challenges from the perspective of cloud system design and operations. Particularly, we design a series of validation experiments to derive statistical evidence from the raw dataset. The results are in relative value due to company policy. Moreover, to facilitate a better understanding of the challenges, we provide some interesting real-world incident examples, which are suggested by on-call engineers during field studies. Similarly, we hide sensitive information for privacy protection.

4.3.1 Key Challenges of Incident Management

We identify two fundamental challenges of incident management and the associated pain points, which are general across different cloud vendors because of the high resemblance in the design principles of cloud systems.

1) *Service/Resource Dependency Discovery.* Dependency is the relationship that an application relies on multiple services/microservices/APIs and physical/logical resources to function correctly. The dependencies can be either static or dynamic, which play an important role in the troubleshooting of distributed systems. However, thus far, the only proven approach to discovering these dependencies, especially fine-grained ones, is by gathering human expert knowledge across different service teams. More often than not, the dependency requires the confirmation of two related teams. This approach is not only inefficient, but also unscalable due to the numerous services and resources in large enterprises. In incident management, the absence of a complete and real-time dependency graph will mainly bring about two critical problems.

Imprecise Impact Estimation. When an incident occurs, OCEs need to estimate the impact scope of the failure and understand how the failure is propagated across the system. Such information is essential as a high-layer service (e.g., Database) needs to know which low-layer services (e.g., Storage) it depends upon are problematic for running corresponding diagnostic tools. However, delay happens as we are missing the whole fine-grained graph of how cloud systems are connected and affecting each other. Although upstream dependencies can be easily gathered (e.g., the Database knows which Storage nodes its components are deployed on and what services they call), downstream dependencies could be vague (e.g., Storage is not aware of how its APIs/resources are visited by other services). Precise impact estimation for an incident plays an important role in automatically identifying the affected customers, which is the main criterion

for deciding an incident's severity. Workarounds or solutions can then be delivered to the customers suffering from the failure promptly and proactively.

Meanwhile, impact estimation can accelerate the procedure of service restoration. Specifically, due to the complexity of distributed systems, even if the incident is resolved, the impacted services may not return back to normal automatically and demand manual checking and recovery for sick cloud resources. Improper planning of resource recovery would delay the restoration of critical high-layer services. The example shown in Figure 4.1 demonstrates that even if the root cause is found, we still need a big picture of which and how services/customers are impacted to prioritize the recovery of cloud resources. Precise service hierarchical dependencies can dramatically facilitate this process.

To understand how the estimation of an incident's impact is delayed, we carefully study the postmortem report of impactful incidents. Particularly, we define Time to Broadcast (TTB) as the time it takes to broadcast a failure to all impacted services and compare it with other incident management phases, i.e., TTD, TTE, and TTM. Table 4.4 presents the results, where, again, the absolute values are concealed. We can see that TTB has comparable values with TTM in almost all services, and they are the two dominant TTx in incident management workflow. Particularly, DCM, serving as a fundamental support to many services, owns the largest TTB. This is because serious failures happened to it often have a widespread impact.

> Lesson Learned 1. *Enumerating all impacted services based on dependencies and prioritizing the cloud resource recovery are as important as locating the root cause of high-impact incidents.*

Redundant Engineering Efforts. Services usually report

Incident ID Resolved Critical	***Multiple air handling unit failure***	
	Service: DCM	# of impacted requests: ~1,000,000
	Datacenter: DC #1	# of impacted accounts: ~1,500

Summary

An air conditioning system failure caused device clusters overheating, which brought down tens of thousands of storage nodes.

Diagnosis

When the air conditioning system was restored, a small portion of failure storage nodes (<1%) failed to recover automatically due to different errors (e.g., main board broken, CPU overheating, data inconsistency, etc.). These nodes demanded manual checking and recovery one by one. With the gradual recovery of storage nodes, many high-layer services confirmed mitigation. However, a Cloud Resource Management (CRM) service serving a large number of users failed to reconnect. It took some time to figure out that a specific node hosting Software Load Balancer (SLB) service was not back to normal. This caused impact to CRM as its load balancing was governed by the SLB node, which therefore deserved a higher priority when planning the order of node recovery. However, this was not the case because: (1) SLB team was not aware of which service instances were running on which SLB nodes, and (2) CRM team attributed the failures to Storage (instead of SLB) at the beginning. Although the dependencies between storage nodes and SLB service were clear, the second-degree dependency that SLB could constitute a single point of failure for CRM was not.

Figure 4.1: Incident Example 1

their own failures independently. The design purpose is to cover missing failures of other services. However, when separate incidents are being handled by different teams, it may not be immediately obvious that there exists a caused-by relation among some of them. This may lead to not only delay in mitigation, but also redundant engineering efforts as different teams are addressing the same problem. Figure 4.2 presents one such incident. A fine-grained dependency graph can dramatically improve the situation as we can correlate incidents by, for example, comparing their origins and tracing their impact. Another circumstance where redundant effort often happens is dealing with historically repeated incidents. In this case, incident correlation can also help by providing similar solutions.

Incident ID
Resolved
Critical

A high error rate of operation [API] has been seen

Service: CRM	# of impacted requests: ~1,000,000
Datacenter: DC #2	# of impacted accounts: ~10,000

Summary

Monitor has detected multiple VMs and web applications unavailable.

Diagnosis

Some operations of Cloud Resource Management (CRM) service suffered from a high error rate. Engineering team found the frontend web service was in a loop of crash and reboot. This resulted in customer requests being held for an extended period of time in web server request queue, leading to slow responses and request timeouts. More than five other services suffered from different failures such as login failures, request timeout errors, etc. The cascading effects and implicit service dependencies made the engineering team hard to know and notify all impacted service teams, especially during busy bug fixing time. Therefore, many impacted services received failure reports and diagnosed their services independently. Particularly, an IT Management Software (ITMS) service attributed the failures to DNS service due to the direct dependency. However, the DNS service was managed by the CRM service (the true root cause), which took ITMS team some time to figure out.

Figure 4.2: Incident Example 2

Table 4.4: Distribution of TTx from Postmortem Reports

	DCM	Network	Storage	Compute	Database	WS
TTD	15.17x	5.24x	1x	13.63x	**18.47x**	5.72x
TTE	15.91x	11.23x	9.82x	7.27x	13.35x	1.57x
TTM	10.95x	12.13x	**14.71x**	**16.21x**	15.51x	**14.84x**
TTB	**17.91x**	**13.59x**	11.06x	12.59x	13.09x	8.69x

To understand the situation of redundant efforts at Microsoft, we calculate the number of incidents that are redundantly handled by more than one service team and the average number of teams involved in such incidents. Particularly, we make use of the links between incidents to identify the incidents of interest. These links are marked by OCEs during the incident investigation, and the caused-by relations can be deduced from them. Specifically, for each incident, we first find its responsible team and the incident that triggers it (if any), called the

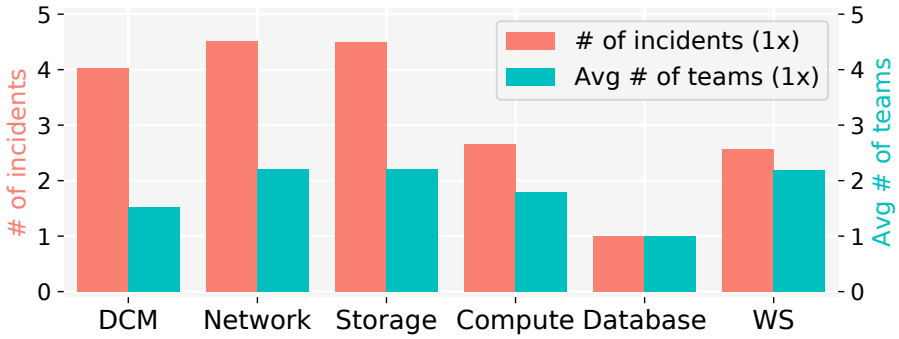


Figure 4.3: The No. of Incidents Incurring Redundant Effort and the Average No. of Involved Service Teams (with different bases)

parent incident. Then, for incidents with the same parent, the associated teams will be considered as addressing the same root cause, i.e., the parent incident. Since the links are incomplete, this selection criteria is quite conservative, yet we still notice a serious situation of redundant effort across different services. In Figure 4.3, we can see Database has both the least number of redundant effort-inducing incidents and the average number of teams. However, Network and Storage generate nearly five times more such incidents and involve more service teams.

> Lesson Learned 2. *Merging separate maintenance work from dependent service teams is vital for quick incident mitigation and effort saving.*

To facilitate failure impact estimation and repetitive effort saving, we propose an alert aggregation framework Girdle in Chapter 7. Girdle leverages multi-source information (i.e., alerts, metrics, and topologies) to characterize the behavioral similarity among services during failures. A high similarity indicates that services are suffering from the same failure. In this way, we can measure the failure impact scope and determine whether different teams are fixing a common fault.

2) *Resource Health Assessment.* In cloud systems, it is an art to design monitoring mechanisms that are able to cover different

Incident ID Resolved Critical	***High Storage resource utilization detected***	
	Service: Storage	# of impacted requests: ~100,000
	Datacenter: DC #3	# of impacted accounts: ~10,000

Summary

Monitor has detected an anomalous high storage resource utilization, impacting multiple properties in [Region].

Diagnosis

The root cause found was the misconfiguration on the request throttling threshold for a specific group of storage nodes. The storage nodes were therefore overloaded. In addition, an OS upgrade by management service was underway at the same time, which further increased the number of transactions and pushed the CPU utilization to an overwhelming level. As many services rely on virtual storage service, 12 other services got impacted, triggering a large number of alerts although they were already aggregated. For example, a web app service reported high failure rate, long response time of some APIs, etc.; a business app service reported SLA drop for different job types, high ratio of unhealthy nodes, etc. Besides, the cloud storage service reported even more types of incidents, such as customer low availability, high impacted counts on VM, different API requests in low availability, high error rate of different container types, etc.

Figure 4.4: Incident Example 3

types of failures for numerous resources and APIs. Particularly, it includes a signal (time-series telemetry data from resources) capturing and anomaly detection. On the one hand, too-sensitive alerts would cause flooding alarms; on the other hand, too-tolerant alerts would cause the missing of potentially impactful failures. The following highlights the pain points that we have observed.

Flooding Alarms. Normally, the alerting threshold of monitors is set to be static and conservative, which will inevitably produce a large number of non-critical alarms. Moreover, due to cloud applications' multi-tiered structure, services in each tier will generate alerts for their failed components. Such a chain effect will trigger a flood of homologous incidents. To alleviate this situation, cloud systems adopt aggregation policies to merge duplicated alerts in appropriate resource levels. Each individ-

Incident ID Resolved Critical	***Disk firmware update disabled disk cache***	
	Service: Storage	# of impacted requests: ~100,000
	Datacenter: DC #4	# of impacted accounts: ~10,000

Summary

Writing to a big data storage platform experienced high failure counts.

Diagnosis

Firmware upgrade to a game drive service inadvertently disabled write cache. At the beginning, there was no direct impact on the service because the number of machines getting into bad state was small and the system was built to tolerate such instances. However, as more and more machines were getting upgraded, the overall latency of the service stack was slowly accumulating and at some point got tipped. It took quite some time to detect the incident which unfortunately deteriorated into a critical issue.

Figure 4.5: Incident Example 4

ual level may contain failures that happened to a finer level of resources. For example, let us assume that a data center has the following toy resource hierarchy: data center > row > rack > node (there are more logical layers in real-world systems). When a failure happens, instead of generating incidents for each separate node, incidents should be created at the data center level to merge the failures of impacted rows, at the row level to merge failed racks, and so forth. Such fine-grained aggregation rules should be carefully kept, as merging all failures only at the highest possible level would mislead the diagnosis. We have seen cases where similar failures of hardware devices in a data center were coincidentally caused by distinct reasons (power failure and firmware bug). In general, to profile service failures more comprehensively, cloud systems apply aggregation to the following monitoring aspects:

- *Resource/API*: Cloud entities associated with the failures.
- *Failure type*: Error type, error code, etc.
- *Impact type*: Availability, performance, task error rate, etc.
- *Customer*: Users experiencing the failures.

- *Location*: Regions where service failures happen.

The goal of alert aggregation is to provide engineers and operators with a clear and integrated view of service health status. However, there are still entities from identical aspects that cannot be merged (e.g., multiple APIs impacted and multiple failure types). Thus, the number of incidents is still overwhelming the incident management system. Figure 4.4 presents a case showing that a failure could trigger a large number of incidents even if aggregations have been applied.

> Lesson Learned 3. *Monitoring cloud systems in different resource levels is proven to be effective for improving the coverage of failure detection in the early stage. However, this may induce flooding alarms even if signal aggregations have been applied. More sophisticated signal aggregation strategies are in high demand.*

Gray Failures. Different from fail-stop failures, the manifestations of gray failures are fairly subtle and thus defy quick and definitive detection, which is quite common for large-scale services. Huang et al. [74] conducted a systematic study on gray failures. For example, if a system's request-handling module is stuck but its heartbeat module is not, an error-handling module dependent on heartbeats will perceive the system as healthy, while a client seeking service will regard it as failed. We have also observed such gray failure incidents in cloud systems (Figures 4.5 and 4.6). In both cases, the error rate reported by monitors is in a reasonable level, so the issues are mistakenly tolerated by the monitoring systems.

We design the following experiments to study the problem of incident false detection. For flooding alarms, although falsely detected incidents will be marked in our system, we also consider incidents that never get handled and mitigate automatically as

Incident ID Resolved Critical	***High latency and timeouts on blob writes***	
	Service: Network	# of impacted requests: ~100,000
	Datacenter: DC #5	# of impacted accounts: ~100

Summary

A Storage blob write API was suffering from long latency (95 quantile latency exceeds threshold). Auto log analysis showed the bottleneck of most requests were on a storage frontend service API.

Diagnosis

Storage team found there were no failures in the storage API, but some network errors in related APIs. The first glance of Network team did not find any bad links. The incident was transferred back to Storage team. However, further checking uncovered a rise on CRC (Cyclic Redundancy Check) error counter on some core regional router devices. One link from router to the regional hub was flapping, which was caused by an unstable cable. Replacing the cable fixed the problem. The issue was not reported immediately, because: (1) the CRC error counter of the router was too small to hit the alerting threshold, and (2) Network service had enough redundant devices and alternative routers to tolerate the errors; therefore, such a small proportion of bad links (<0.1%) was hidden in network UI tool by default. However, the impacted service happened to be very sensitive to jitters, so problem got triggered.

Figure 4.6: Incident Example 5

flooding cases. Another type of flooding alarm is the incidents generated due to the chain effect of cloud failures. We adopt redundant effort-inducing incidents (Figure 4.3) as such cases, as they stem from identical issues. Particularly, duplicated incidents found by different criteria are removed. Regarding gray failure, our system does not explicitly mark them because: (1) trivial mistakes can be safely ignored as they have no impact on services; (2) serious failures will eventually be found when they manifest themselves, but there is no need to mark them as technically they are all failed detection. To tackle this problem more reasonably, we make use of the incident's severity. Specifically, during the lifetime of an incident, if its severity level ever gets upgraded, it will be considered a gray failure because it is not correctly identified regarding how serious it is at the beginning. Again,

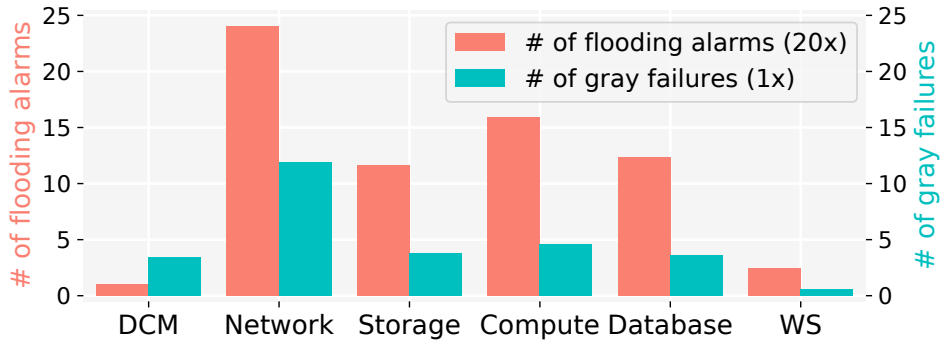


Figure 4.7: The No. of Flooding Alarms and Gray Failures

the designed rules for discovering false incident detection are pretty conservative, but we can still see it is a ubiquitous problem in cloud services. As shown in Figure 4.7, compared to DCM and WS, other services have a much more serious situation of flooding alarms. Regarding gray failure, other services encounter much more cases than WS does, especially Network. This aligns with the results in Table 4.3, demonstrating the complexity of Network-related failures.

> LESSON LEARNED 4. *Large-scale cloud systems are prone to gray failures. Setting insensitive alarms to avoid flooding alerts may cause missing critical issues.*

To suppress flooding alarms and gray failures, we delve into service anomaly detection based on logs and metrics. Specifically, in Chapter 5, we conduct a systematic review and evaluation of deep learning techniques for log anomaly detection. By knowing the characteristics of the representative methods, the industry is able to select appropriate ones for different problems. In Chapter 6, we propose ADSketch, an interpretable and adaptive performance anomaly detection algorithm. ADSketch gains engineers' trust in its outputs by explaining anomalies' type, and updates its knowledge of the anomalies online to combat system dynamicity.

4.3.2 Understand the Key Challenges

This section details the fundamental reasons behind the aforementioned key challenges. Specifically, we believe the dependency issues are essentially brought by system modular design and the virtualization of physical infrastructure; the difficulty of resource health assessment is twofold: (1) the system's fault-tolerant property and (2) a series of monitor design and distribution problems.

Software System Modularity. In cloud systems, applications follow a microservice-based architecture that decomposes the application logic into several interacting component services. These components are often independently developed in a cloud hosting environment, making cloud systems much more complicated than conventional ones. The complexity of the dependency graph would grow exponentially with the number of services. One important reason is that there is no general way to identify from which source an API is called. The capacity planner and load balancer used in the architecture further complicate the API calling, as shown in Figure 4.8. Therefore, we need to start from the source side to collect respective dependency sub-graphs and integrate them into a global one. In this manner, it might seem that the graph can be easily constructed if the service designer can generate rules to specify its dependencies. However, typical challenges include the diversity of different services, fast system evolution, and rule unavailability of legacy systems, which are also mentioned by Bahl et al. [7].

Physical Infrastructure Virtualization. Virtualization allows the abstraction of physical infrastructure, which however makes it difficult to identify the dependency graph from an incident to the problematic physical component(s). There are mainly two reasons. First, the dependencies are dynamically constructed due to system reconfiguration and/or resource migration. For example, if one VM is temporarily stopped on its host machine, it

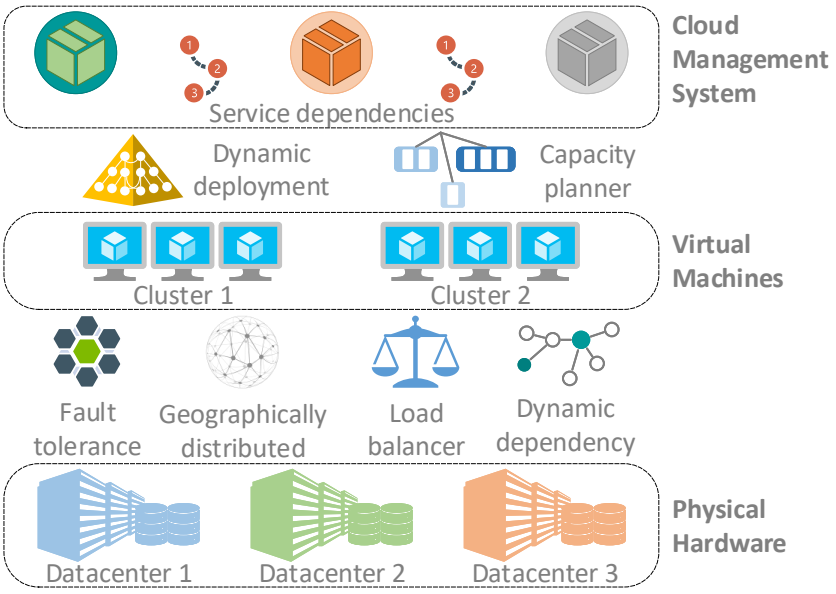


Figure 4.8: A Typical Cloud Computing Architecture

can be moved to a new host without disrupting its users. Therefore, services can be dynamically deployed in different VMs, and the dependencies between VMs and physical machines are also dynamic, as shown in Figure 4.8. Second, logically related resources can be highly geographically-distributed. For example, many modern cloud applications use clusters of virtual machines to implement load-balancing and ensure resilience for critical tasks. The physical nodes that host the VMs in the same cluster could be in different data centers or regions, which increase the difficulty of problem localization, as illustrated in Figure 4.8.

Fault Tolerance. Fault tolerance is critical for cloud platforms to provide highly stable availability and business continuity of mission-critical systems or applications. In cloud computing, availability zone is one of the best practices of fault tolerance, which protects service availability from data center failures by replicating applications and data. The resiliency is ensured by its physical separation in terms of power, data, networking, etc. However, in some cases, fault tolerance hinders the assessment

of resource health by hiding problems in the early stage. These small issues have the potential to incur fatal consequences if not handled seriously and timely. Figure 4.5 presents such a potential threat, demonstrating the need for more sophisticated fault-tolerant mechanisms.

Monitor Design and Distribution. Monitor design and monitor distribution are two important factors affecting the performance of assessing resource health. Specifically, monitor design refers to what signals should be monitored and the corresponding alerting logic; monitor distribution describes what resources should be monitored to pursue accurate and timely incident detection.

When designing monitors, we need to identify what metrics and events that are most representative of resource health status. More often than not, a set of metrics collectively can constitute a stronger performance predictor as they provide more complete and comprehensive information. This is a typical feature engineering problem that relies on IT practitioners' domain knowledge. Another issue of monitor design is the alerting rules determining when to raise incidents. One widely-adopted rule is setting thresholds on the time-series signals of resources and checking whether any of them is violated. However, this kind of rule is too simple, which causes a large number of flooding alarms. To address this problem, some monitors incorporate dynamic thresholds, multi-dimension-metrics-based diagnostics, and others. However, despite the advances in monitor design, there are still far too many flooding alarms.

In cloud systems, how monitors should be distributed still remains an unexplored problem. Typical challenges include: (1) resources can be either physical or virtual; (2) granularity problem, i.e., and sometimes a single computer should be monitored, sometimes a single process is appropriate [7]. For the case in Figure 4.6, as there is no monitor monitoring per-service errors, it

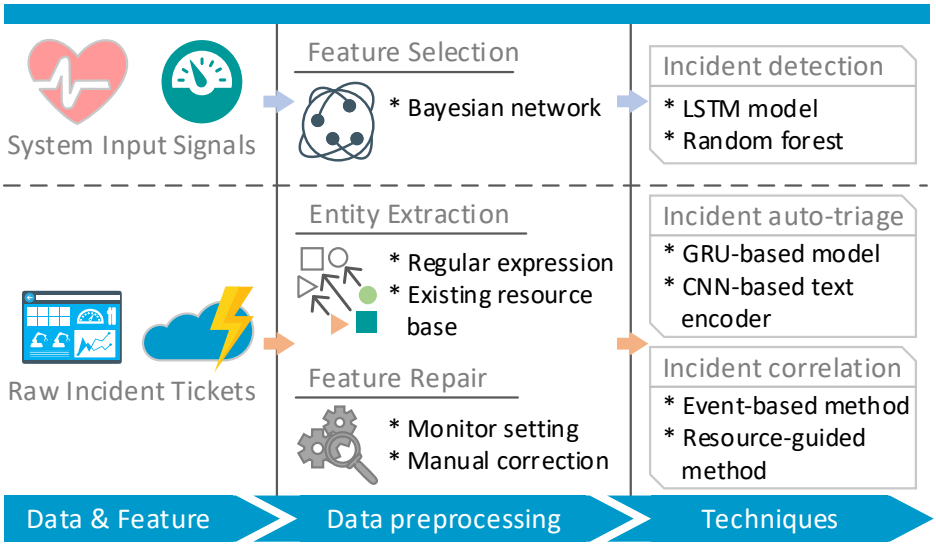


Figure 4.9: The Framework of BRAIN

is hard to provide tailored troubleshooting for individual services. However, it is not saying that we should deploy as many monitors as possible, which is practically infeasible and will incur system performance degradation.

4.4 BRAIN: An AIOps Framework

As shown by our previous study, critical cloud incidents often occur in an unexpected manner, and thus, dedicated approaches could fail. Nevertheless, we notice the root causes of critical incidents share many similar features. This is where we see AI/ML techniques can help by extracting patterns from recurrent incident symptoms and providing actionable recommendations.

We present BRAIN, an AIOps framework aiming at improving the entire pipeline of incident management at Microsoft. As shown in Figure 4.9, BRAIN consists of three modules: Data and Features, Data Preprocessing, and Techniques.

4.4.1 Design Principles

Based on our experience and empirical analysis, we first describe the design principles of BRAIN, i.e., how BRAIN addresses the identified key challenges (Section 4.3.1).

Regarding dependency discovery, attempts have been made to track the run-time dependencies of applications by standardizing the middleware infrastructure [14, 8, 27]. However, as applications come from a wide variety of vendors, it is impractical that all vendors will instrument their services in a common fashion [7]. Log analysis [181, 106, 9] would be a non-intrusive way to construct dependencies across different servers, processes, and third-party services. However, this solution cannot meet the real-time needs of extremely large-scale distributed systems due to data explosion, log heterogeneity, dynamic changes of dependencies, etc. On the other hand, we notice that before the occurrence of a critical cloud issue, many related incidents would have happened in a short period of time. In this process, individual service teams are alerting and mitigating incidents separately. Being able to provide OCEs with related incidents can dramatically save redundant engineering effort and facilitate root cause localization. Therefore, instead of tracking fine-grained service dependencies, BRAIN resorts to incident correlation to pursue reliable cloud services.

The accuracy of resource health assessment is crucial to cloud systems. However, it cannot be achieved by pursuing the perfection and completeness of a purely rule-based monitoring system. As in Figure 4.6, the fundamental reason for such failure is the absence of per-service monitors. Given the system's dynamicity and the intransparency between different application tiers, it is extremely hard to formulate the problem of monitor design and distribution mathematically. In contrast, BRAIN develops a series of incident detection algorithms on top of various system signals, e.g., service health data and infrastructure signals. More-

over, the resource hierarchy relationship is used to understand topologies, resiliency models, and dependencies among the entire cloud system.

4.4.2 Data and Features

Two sources of data are utilized in BRAIN, namely, raw incident tickets and various system input signals.

1) *Raw incident tickets.* In BRAIN, we utilize all incident tickets that have been reported to the incident management system at Microsoft. These incidents come from different service teams and therefore can provide us with a global view of the service health state across the cloud system.

2) *System input signals.* BRAIN runs 24×7 non-stop, analyzing the signals and patterns to detect anomalies in the systems. Particularly, the input to BRAIN includes the following categories:

- *Near Real Time (NRT) health signals.* The health signals are collected from each cloud resource, individual services' monitors, and system-deployed active monitors.
- *NRT metrics.* Service availability, performance metrics, request volume, etc.
- *System topology.* The hierarchy information of different resources across the entire cloud system.
- *Infrastructure signals.* Low-level infrastructure sensors sensing data center traffic volume, temperature, power consumption, local weather, etc.
- *Customer input.* Customer Service & Support reports which consist of many categorical attributes such as product version, the problematic product feature, product configuration, client OS, service package, etc. [196, 100].
- *Historical data.* Change history, metric history, etc.

4.4.3 Data Preprocessing

The incident management system at Microsoft is a hub system that involves thousands of service teams. Particularly, different teams may have their own platforms for monitoring and processing service failures. The tools for incident diagnosis may also vary. Consequently, incident tickets in the system are rendered by different monitoring platforms with different diagnostic tools and thus contain various data types. Poor signal-to-noise ratio and data inconsistency are therefore inevitable. Moreover, the incident management system is essentially a ticketing system that only records relevant information throughout the lifecycle of incidents. Such a system is not dedicated to facilitating data analysis in the postmortem phase regarding its design. Therefore, we value the procedure of data preprocessing and propose the following three methods to improve data quality.

1) *Entity Extraction.* For large-scale cloud enterprises, different service teams and monitors usually have distinct standards for rendering incident tickets, such as different abbreviations for locations, different incident title templates, etc. Consequently, it is challenging to design an incident ticket parser generally applicable for raw feature extraction. Therefore, to profile incidents in a unified manner, we maintain global dictionaries for different entities in incidents, e.g., resource, device name, etc. Particularly, entities are extracted through regular expressions combined with the existing resource base at Microsoft. Such dictionaries can assist us in recognizing special terms with low occurrence frequency [161].

2) *Feature Repair.* Incorrect and empty features are two common data quality issues in incident management. To tackle them, we propose to conduct feature repair for incidents before consuming them. Specifically, we first search empty fields in an incident ticket and check whether each non-empty field has a valid value. This is done by querying the global dictionaries

built in entity extraction stage. Then, problematic fields will be auto-filled or -corrected by borrowing the setting of the alerting monitors or mining the right features from its textual descriptions (i.e., title, summary, and discussions) with predefined regular expressions. Meanwhile, for impactful incidents, due to their significance and minority, we perform manual correction for their features.

3) *Signal Selection.* When diagnosing failures for cloud services, engineers usually start by hunting for a small subset of system signals that are symptoms incurred by the causes of the incidents, called service-incident beacons [107]. However, the manual signal selection is too inefficient and relies heavily on domain expertise. To tackle this problem, we develop a Bayesian network inference method [26] to model the relationship between system signals and impactful incidents. The most relevant signals will be selected as the features for model training.

4.4.4 Techniques of BRAIN

1) *Incident Detection.* Incident detection is to identify service issues based on various system signals. It pursues an early detection of gray failures and recognizes important issues from trivial ones. In cloud systems, time series and event sequence are two major types of telemetry data, where anomalies often manifest as having a large magnitude of upward/downward changes. Besides traditional martingale methodologies [72, 45], BRAIN also exploits sophisticated characteristics of the signals. Particularly, signals are classified as temporal or spatial, tackled by an LSTM model and a Random Forest model, respectively [99]. To enhance the interaction among different signals, BRAIN calculates a series of statistical features for a set of data points in a rolling window, e.g., mean and variance [188]. In BRAIN, significant progress (e.g., a $\sim$0.7 F1 score [26]) has been made when detecting certain

types of cloud failures, e.g., unplanned VM reboot, node failure, and API throttling.

2) *Incident Auto-triage.* In the incident triage phase, OCEs continuously hold discussions until the correct service team is found. During this process, knowledge is accumulated with the number of discussions. BRAIN tries to automate the triage of incidents with fewer discussions such that problem investigation can be triggered earlier. Specifically, we design a GRU-based model [21] to effectively utilize incremental discussions by considering their temporal relationship. Three types of data are fed into our model: 1) incident title and summary, 2) incident raw discussions, and 3) environment information, e.g., incident type, monitor and device reporting the incident, etc. The global entity dictionaries can be used to ensure the correctness and consistency of this information. However, since discussions are conducted by engineers, it tends to introduce noise. We propose an attention-based mask strategy [21] to bypass the noise. In this way, different weights can be automatically assigned to different discussion information, and noise can be masked out by assigning trivial weights. Due to low frequency, special terms (e.g., API and component names) cannot be adequately encoded by traditional text encoding methods. We adopt a CNN-based neural-language model [84, 79] to perform domain-specific text encoding. Our model [21] has achieved a remarkable accuracy of 0.64-0.73, which outperforms the state-of-the-art bug triage approach [89] by a significant margin of 12.2%-35.5%.

3) *Incident Correlation.* Incident correlation tries to alleviate the situation of redundant efforts and assist the impact estimation of failures. We propose two algorithms: event-based and resource-guided methods. In the event-based method, due to the high resemblance between the incident title and log, we use an automatic log parsing method [50] to extract templates from the repaired incident titles. Based on word-level similarity, tem-

plates are grouped to form incident events, representing different types of service issues. The relationship among incident events is deduced from incidents' historical links, which are marked by OCEs during incident investigation. Such links are used for model training and evaluation. During the evaluation, incidents will be connected if their representing events are ever linked before. Although this is an effective way of using OCEs' domain knowledge, in some cases, log parsing methods cannot meet our needs. It is because in titles, special terms with small frequency are often erased, and the extracted events are indistinguishable in terms of identifying which resource is unhealthy. Thus, we develop a resource-guided method to perform incident correlation in a fine-grained manner. Specifically, two incidents are considered correlated if they are tagged with identical or related resources (with appropriate location and time constraints). In particular, there are two ways for identifying related resources: (1) leveraging existing hierarchy information of resources at Microsoft, and (2) mining their spatial and temporal co-occurrences in incidents. Combining these two methods, we are able to achieve ¿0.89 precision, recall, and F1 score for incident correlation.

4.4.5 Evaluation

BRAIN features have been continuously deployed in the incident management system at Microsoft. To evaluate its effectiveness so far, we collected impactful incidents captured in the past one year and split them into two groups. The first group, referred to as "No BRAIN," contains 55.2% of the total incidents that were not engaged with BRAIN. The second group, referred to as "BRAIN," contains the rest 44.8% incidents engaged with BRAIN. Particularly, we compare the time spent in different phases of incident management. The bar chart in Figure 4.10 shows the 75th percentile TTx of the two groups, and it clearly shows

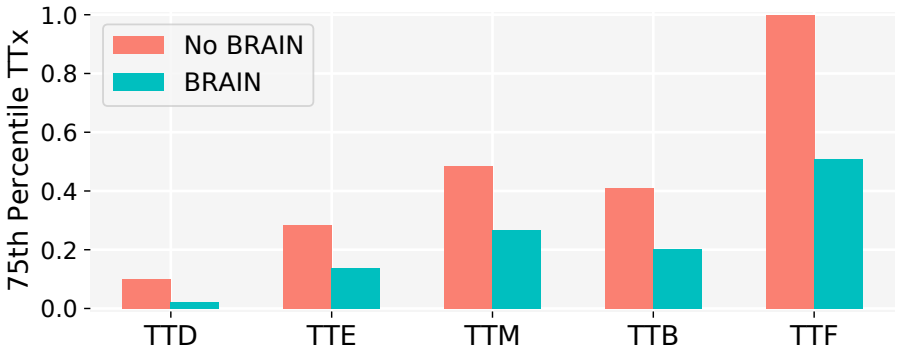


Figure 4.10: BRAIN's Effect on TTx (normalized)

Table 4.5: Non-parametric Hypothesis Test on TTx reduction

Null Hypothesis	p-value	Decision
BRAIN has no effect on the shorter TTD	3.38E-08	Reject
BRAIN has no effect on the shorter TTE	5.44E-08	Reject
BRAIN has no effect on the shorter TTM	5.90E-03	Reject
BRAIN has no effect on the shorter TTB	4.81E-16	Reject
BRAIN has no effect on the shorter TTF	8.93E-15	Reject

impactful incidents engaged with BRAIN have shorter TTD, TTE, TTM, TTB (Section 4.3.1), and TTF (Section 4.2.2).

To account for the sample-to-sample variation, we performed a non-parametric hypothesis test (Mann-Whitney-Wilcoxon test). The null hypothesis is that the reduction TTx seen here is due to the randomness in the data. The p-value measures how probable the null hypothesis is, given the observed trend in the sample. If the p-value is low (i.e., ¡0.05, at the 95% significance level), we would claim that the null hypothesis is improbable and reject it in favor of the alternative hypothesis - *the observed reduction of TTx is indeed related to BRAIN*. The key results are summarized in Table 4.5. Supported by this testing, we conclude that BRAIN's associations with shorter TTx are statistically significant. Therefore, BRAIN manifests itself as an effective facilitator for TTx reduction.

4.5 Summary

In this chapter, we summarize two main challenges of incident management at large-scale service providers: (1) the lack of a fine-grained service/resource dependency graph; and (2) the imprecision of health assessment for cloud resources. Particularly, the dependency graph is dramatically complicated by system modularity and visualization technology. While the fault-tolerant mechanism could sometimes impede the detection of unhealthy resources, the imperfection of monitor design and distribution further compound the problem. We conduct a quantitative analysis of incidents from six core services at Microsoft and provide five real-world incident examples as well as four lessons learned. We also present BRAIN, our AIOps framework, which is able to reduce the time cost in different incident management phases effectively.

We believe our work could shed light on future research and engineering effort towards failure-resilient cloud systems, for example, high-performance algorithms for accelerating different incident management phases, design of efficient incident management workflow, and more advanced cloud architecture.

□ End of chapter.

Chapter 5

Deep Log Anomaly Detection

Logs faithfully reflect the runtime status of a software system, which are of great importance for the monitoring, administering, and troubleshooting of a system. Therefore, log-based anomaly detection, which aims to uncover abnormal system behaviors, has become an essential means to ensure system reliability and service quality. To leverage this technique for assessing resource health condition (Chapter 4), in this chapter, we provide an evaluation of existing DL-based log anomaly detection methods. We compare their performance regarding accuracy, robustness, and efficiency on two representative log datasets. A toolkit is released for public reuse. This chapter is organized as follows. Section 5.1 introduces the problem background and our contributions. Section 5.2 summarizes the problem formulation of log anomaly detection and reviews six representative methods leveraging neural networks. Section 5.3 presents the experiments and experimental results. Section 5.4 shares our industrial practices. Finally, Section 5.5 summarizes this chapter.

5.1 Problem and Contributions

For traditional on-premise software systems, engineers usually perform simple keyword searches (such as "failed," "exception,"

and "error") or rule matching [64, 134] to locate suspicious logs that might be associated with system problems. Due to the ever-increasing volume, variety, and velocity of logs produced by modern software systems, such manual approaches fall short for being labor-intensive and error-prone. Thus, many studies resort to statistical and traditional machine learning algorithms to incorporate more automation into this process. Exemplary algorithms are principal component analysis (PCA) [169], log clustering [101], etc. Although these methods have made remarkable performance gains, they still face the following limitations in terms of practical deployments:

- **Insufficient interpretability.** For log anomaly detection, interpretable results are critical for admins and analysts to trust and act on the automated analysis, e.g., which logs are important or which system components are problematic. However, many traditional methods only make a simple prediction for input with no further details. Engineers need to conduct a manual investigation for fault localization, which, in large-scale systems, is like finding a needle in a haystack.
- **Weak adaptability.** During feature extraction, these methods often require the set of distinct log events to be known beforehand [189]. However, as modern systems continuously undergo feature addition and system upgrade, unseen log events could constantly emerge. To embrace the new log events, some models need to be retrained from scratch.
- **Handcrafted features.** As an essential part of traditional ML workflow, many ML-based methods, e.g., [200, 95], require tailored features. Due to the variety of different systems, some of the selected features might not always be applicable, and other critical ones could be missing. Feature engineering is time-consuming and demands human domain knowledge.

Due to the exceptional ability to model complex relationships, deep learning has produced results comparable to and in some areas surpassing human expert performance. It often adopts a multiple-layer architecture called neural networks to progressively extract features from inputs with different layers dealing with different levels of feature abstraction. Typical architectures include recurrent neural networks (RNNs), convolutional neural networks (CNNs), autoencoders, etc. They have been widely applied to various fields, including computer vision, neural language processing, etc. In recent years, there has been an explosion of interest in applying neural networks to log-based anomaly detection. For example, Du et al. [40] employed long short-term memory (LSTM) networks for this purpose. On top of their work, Zhang et al. [189] and Meng et al. [114] further considered the semantic information of logs to improve the model's adaptability to unprecedented logs.

Given such fruitful achievements in the literature, we, however, observe a gap between academic research and industrial practices. One important reason is that site reliability engineers may not have fully realized the advances of DL techniques in log-based anomaly detection [37]. Thus, they are not aware of the existence of some state-of-the-art anomaly detection methods. This issue is further compounded by the fact that engineers may not have enough ML/data science background and skills. As a result, it would be cumbersome for them to search through the literature and select the most appropriate method(s) for the problems at hand. Another important reason is that, to the best of our knowledge, there is currently no open-source toolkit available that applies DL techniques for log-based anomaly detection. Thus, if the code of the original paper is not open-source (which is not uncommon), engineers need to re-implement the model from scratch. In this process, bias and errors can be easily introduced because 1) the papers may not provide enough implementation

details (e.g., parameter settings), and 2) engineers may lack experience developing DL models with relevant frameworks such as PyTorch [135] and TensorFlow [157].

He et al. [70] have conducted a systematic comparative study in this area, covering only traditional ML-based methods. Compared to them, DL-based methods possess the following merits: 1) more interpretable results, which are vital for engineers and analysts to take remediation actions, 2) better generalization ability to unseen logs which constantly appear in modern software systems, and 3) automated feature engineering which requires little human intervention. These merits render the necessity of a complementary study of DL-based solutions. In this chapter, we conduct a comprehensive review and evaluation of five representative neural networks used by six log anomaly detection methods. To facilitate reuse, we also release an open-source toolkit[1] containing the studied models. We believe researchers and practitioners can benefit from our work in the following two aspects: 1) they can quickly understand the characteristics of popular DL-based anomaly detectors and the differences with their traditional ML-based counterparts, and 2) they can save enormous efforts on re-implementations and focus on further customization or improvement.

The log anomaly detectors selected in this work include four unsupervised methods (i.e., two LSTMs [40, 114], the Transformer [122], and Autoencoder [44]) and two supervised methods (i.e., CNN [108] and attentional BiLSTM [189]). As labels are often unobtainable in real-world scenarios [29], unsupervised methods are more favored. When a system runs in a healthy state, the generated logs often exhibit stable and normal patterns. An abnormal instance usually manifests as an outlier that significantly deviates from such patterns. Based on this observation, unsupervised methods try to model logs' normal patterns

[1] https://github.com/logpai/deep-loglizer

and measure the deviation for each data instance. On the other hand, supervised methods directly learn the features that can best discriminate between normal and abnormal cases based on the labels. All selected methods are evaluated on two widely-used log datasets that are publicly available, i.e., HDFS and BGL, containing nearly 16 million log messages and 0.4 million anomaly instances in total. The evaluation results are reported in *accuracy*, *robustness*, and *efficiency*. We believe our work can prompt industrial applications of more recent log-based anomaly detection studies and provide guidelines for future research.

5.2 Log Anomaly Detection

To leverage neural networks for log anomaly detection, the network architecture as well as its loss function should be properly decided. Particularly, the loss function guides how the model learns the log patterns. In this section, we first elaborate on how existing work formulates the model loss. Then, we introduce six state-of-the-art methods, including four unsupervised methods (i.e., DeepLog [40], LogAnomaly [114], Logsy [122], and Autoencoder [44]) and two supervised methods (i.e., LogRobust [189] and CNN [108]).

5.2.1 Loss Formulation

The task of log anomaly detection is to uncover anomalous samples in a large volume of log data. A loss should be set for a model with respect to the characteristics of the log data, which serves as the goal to optimize. Generally, each neural network has its typical loss function(s). However, we can set a different goal for it with proper modification in its architecture (e.g., [189]). We have summarized the following three representative types of model losses.

Forecasting Loss

Forecasting loss guides the model to predict the next appearing log event based on previous observations. A fundamental assumption behind an unsupervised method is that the logs produced by a system's normal executions often exhibit certain stable patterns. When failures happen, such normal log patterns may be violated. For example, erroneous logs appear, the order of log events shifts incorrectly, log sequences become incomplete due to early termination, etc. Therefore, by learning log patterns from normal executions, the method can automatically detect anomalies when the log pattern deviates from normal cases. Specifically, for a log event e_i which shows up at time step t, an input window $\mathcal{W}$ is first composed, which contains m log events preceding e_i, i.e., $\mathcal{W} = [e_{t-m}, \ldots, e_{t-2}, e_{t-1}]$. This is done by dividing log sequences (generated by some log partition strategy) into smaller subsequences. The division process is controlled by two parameters called window size and step size, which are similar to the partition size and stride of the sliding partitioning (Section 2.4.3). A model is then trained to learn a conditional probability distribution $P(e_t = e_i|\mathcal{W})$ for all e_i in the set of distinct log events $E = \{e_1, e_2, \ldots, e_n\}$ [40]. In the detection stage, the trained model makes a prediction for a new input window, which will be compared against the actual log event. An anomaly is alerted if the ground truth is not one of the most k probable log events predicted by the model. A smaller k imposes more demanding requirements on the model's performance.

Reconstruction Loss

Reconstruction loss is mainly used in autoencoders, which trains a model to copy its input to its output. Specifically, given an input window $\mathcal{W}$ and the model's output $\mathcal{W}'$, the reconstruction

loss can be calculated as $sim(\mathcal{W}, \mathcal{W}')$, where sim is a similarity function such as the Euclidean norm. By allowing the model to see normal log sequences, it will learn how to reconstruct them properly. However, when faced with abnormal samples, the reconstruction may not go well, leading to a large loss.

Supervised Loss

Supervised loss requires anomaly labels to be available beforehand. It drives the model to automatically learn the features that can help distinguish abnormal samples from normal ones. Specifically, given an input window $\mathcal{W}$ and its label y_w, a model is trained to maximize a conditional probability distribution $P(y = y_w|\mathcal{W})$. Commonly-used supervised losses include cross-entropy and mean squared error.

5.2.2 Existing Methods

In this section, we introduce six existing methods which utilize popular neural networks to conduct log-based anomaly detection. They have a particular choice of model loss and whether to employ the semantic information of logs. We would like to emphasize different combinations (with respect to the model's characteristics and the problem at hand) would yield different methods. For example, by incorporating different loss functions, LSTM models can be either unsupervised [40, 114] or supervised [189]; one method uses the index of log events purely may also accept their semantics; model combinations are also possible as demonstrated by Yen et al [180], i.e., a combination of CNN and LSTM.

Unsupervised Log-based Anomaly Detection

The selected four unsupervised methods are introduced as follows:

DeepLog. Du et al. [40] proposed DeepLog, which is the first work to employ LSTM for log anomaly detection. Particularly,

the log patterns are learned from the sequential relations of log events, where each log message is represented by the index of its log event. It is also the first work to detect anomalies in a forecasting-based fashion, which is widely used in many follow-up studies.

LogAnomaly. To consider the semantic information of logs, Ma et al. [114] proposed LogAnomaly. Specifically, they proposed *template2Vec* to distributedly represent the words in log templates by considering the synonyms and antonyms therein. For example, the representation vector of the word "down" and "up" should be distinctly different as they own opposite meanings. To this end, *template2Vec* first searches synonyms and antonyms in log templates and then applies an embedding model, dLCE [123], to generate word vectors. Finally, the template vector is calculated as the weighted average of the word vectors of the words in the template. Similarly, LogAnomaly adopts forecasting-based anomaly detection with an LSTM model. In this chapter, we follow this work to evaluate whether log semantics can bring performance gain to DeepLog.

Logsy. Logsy [122] is the first work utilizing the Transformer [159] to detect anomalies in log data. It is a classification-based method that learns log representations in a way to better distinguish between normal data from the system of interest and abnormal samples from auxiliary log datasets. The auxiliary datasets help learn a better representation of the normal data while regularizing against overfitting. Similarly, in this work, we employ the Transformer with the multi-head self-attention mechanism. The procedure of anomaly detection follows that of DeepLog [40], i.e., forecasting-based. Particularly, we use two types of log event sequences: one only contains the indices of log events as in DeepLog [40], while the other is encoded with log semantic information as in LogAnomaly [114].

Autoencoder. Farzad et al. [44] were the first to employ

the autoencoder [144] combined with isolation forest [102] for log-based anomaly detection. The autoencoder is used for feature extraction, while the isolation forest is used for anomaly detection based on the produced features. In this chapter, we employ an autoencoder to learn representation for normal log event sequences. The trained model is able to encode normal log patterns properly. When dealing with anomalous instances, the reconstruction loss becomes relatively large, which serves as an important signal for anomalies. We also evaluate whether the model performs better with logs' semantics.

Supervised Log-based Anomaly Detection

The selected two supervised methods are introduced as follows:

LogRobust. Zhang et al. [189] observed that many existing studies of log anomaly detection fail to achieve the promised performance in practice. Particularly, most of them carry a closed-world assumption, which assumes: 1) the log data is stable over time; 2) the training and testing data share an identical set of distinct log events. However, log data often contain previously unseen instances due to the evolution of logging statements and log processing noise. To tackle such a log instability issue, they proposed LogRobust to extract the semantic information of log events by leveraging off-the-shelf word vectors, which is one of the earliest studies to consider logs' semantics, as done by Meng et al. [114].

More often than not, different log events have distinct impacts on the prediction result. Thus, LogRobust incorporates the attention mechanism [6] into a Bi-LSTM model to assign different weights to log events, called attentional BiLSTM. Specifically, LogRobust adds a fully-connected layer as the attention layer to the concatenated hidden state h_t. It calculates an attention weight (denoted as a_t), indicating the importance of the log event at time step t as $a_t = tanh(W_t^a \cdot h_t)$, where W_t^a is the weight of

the attention layer. Finally, LogRobust sums all hidden states at different time steps with respect to the attention weights and employs a softmax layer to generate the classification result (i.e., anomaly or not) as $prediction = softmax(W \cdot (\sum_{t=1}^{T} a_t \cdot h_t))$, where W is the weight of the softmax layer, and T is the length of the log sequence.

CNN. Lu et al. [108] conducted the first work to explore the feasibility of CNN [88] for log-based anomaly detection. The authors first constructed log event sequences by applying identifier-based partitioning (Section 2.4.3), where padding or truncation is applied to obtain consistent sequence lengths. Then, to perform convolution calculation which requires a two-dimensional feature input, the authors proposed an embedding method called *logkey2vec*. Specifically, they first created a trainable matrix whose shape equals $\#distinct\ log\ events \times embedding\ size$ (a tuneable hyperparameter). Then, different convolutional layers (with different shape settings) are applied, and their outputs are concatenated and fed to a fully-connected layer to produce the prediction result.

5.2.3 Tool Implementation

In the literature, tremendous efforts have been devoted to the development of DL-based log anomaly detection. While they have achieved remarkable performance, they have not yet been fully integrated into industrial practices. This gap largely comes from the lack of publicly available tools that are ready for industrial usage. For site reliability engineers who have limited expertise and experience in ML techniques, re-implementation requires non-trivial efforts. Moreover, they are often busy with emerging issue mitigation and resolution. Yet, the implementation of DL models is usually time-consuming and involves the process of parameter tuning. This motivates us to develop a unified toolkit

that provides out-of-the-box DL-based log anomaly detectors.

We implemented the studied six anomaly detection methods in Python with around 3,000 lines of code and packaged them as a toolkit with standard and unified input/output interfaces. Moreover, our toolkit aims to provide users with the flexibility for model configuration, e.g., different loss functions and whether to use logs' semantic information. For DL model implementation, we utilize a popular machine-learning library, namely PyTorch [135]. PyTorch provides basic building blocks (e.g., recurrent layers, convolution layers, Transformer layers) for the construction of a variety of neural networks such as LSTM, CNN, the Transformer, etc. For each model, we experiment with different architecture and parameter settings. We employ the setting that constantly yields a good performance across different log datasets.

5.3 Evaluation

In this section, we evaluate six DL-based log anomaly detectors on two widely-used datasets [71] and report the benchmarking results in terms of accuracy, robustness, and efficiency. They represent the key quality of interest to consider during industrial deployment.

- *Accuracy* measures the ability of a method to distinguish anomalous logs from normal ones. This is the main focus of this field. A large false-negative rate would miss critical system failures, while a large false-positive rate would incur a waste of engineering effort.

- *Robustness* measures the ability of a method to detect anomalies with the presence of unknown log events. As modern software systems involve rapidly, this issue starts to gain more attention from both academia and industry. One

common solution is leveraging logs' semantic information by assembling word-level features.

- *Efficiency* gauges the speed of a model to conduct anomaly detection. We evaluate the efficiency by recording the time an anomaly detector takes in its training and testing phases. Nowadays, terabytes and even petabytes of data are being generated on a daily basis, imposing stringent requirements on the model's efficiency.

5.3.1 Experiment Design

Log Dataset

He et al. [71] released Loghub, a large collection of system log datasets. We report results evaluated on two popular datasets, namely, HDFS [169] and BGL [125]. Our toolkit can be easily extended to other datasets. Table 5.1 summarizes the dataset statistics.

HDFS. HDFS dataset contains 11,175,629 log messages, which are generated by running map-reduce tasks on more than 200 Amazon's EC2 nodes [40]. Particularly, each log message contains a unique *block_id* for each block operation such as allocation, writing, replication, and deletion. Thus, identifier-based partitioning can be naturally applied to generate log event sequences. After preprocessing, we end up with 575,061 log sequences, among which 16,838 samples are anomalous. A log sequence will be predicted as anomalous if any of its log windows, $\mathcal{W}$, is identified as an anomaly.

BGL. BGL dataset contains 4,747,963 log messages, which are collected from a BlueGene/L supercomputer at Lawrence Livermore National Labs. Unlike HDFS, logs in this dataset have no identifier to distinguish different job executions. Thus, timestamp-based partitioning is applied to slice logs into log sequences. The number of the resulting sequences depends on

the partition size (and stride). In the BGL dataset, 348,460 log messages are labeled as failures. A log sequence is marked as an anomaly if it contains any failure logs.

Evaluation Metrics

Since log anomaly detection is a binary classification problem, we employ *precision*, *recall*, and *F1 score* for accuracy evaluation. Specifically, precision measures the percentage of anomalous log sequences that are successfully identified as anomalies over all the log sequences that are predicted as anomalies: $precision = \frac{TP}{TP+FP}$; recall calculates the portion of anomalies that are successfully identified by a model over all the actual anomalies: $recall = \frac{TP}{TP+FN}$; F1 score is the harmonic mean of precision and recall: $F1\ score = 2 \times \frac{precision \times recall}{precision+recall}$. TP is the number of anomalies that are correctly disclosed by the model, FP is the number of normal log sequences that are wrongly predicted as anomalies by the model, FN is the number of anomalies that the model misses.

Experiment Setup

For a fair comparison, all experiments are conducted on a machine with 4 NVIDIA Titan V Pascal GPUs (12GB of RAM), 20 Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz, and 256GB of RAM. The parameters of all methods are fine-tuned to achieve the best results. To avoid bias from randomness, we run each method five times and report the best result.

For all datasets, we first sort logs in chronological order and apply log partition to generate log sequences, which will then be shuffled. Note we do not shuffle the input windows, $\mathcal{W}$, generated from log sequences. Next, we utilize the first 80% of the data for model training and the remaining 20% for testing. Particularly, for unsupervised methods that require no anomalies for training,

Table 5.1: Dataset Statistics

Dataset	Time span	#Logs	#Anomalies
HDFS	38.7 hrs	11,175,629	16,838
BGL	7 mos	4,747,963	348,460

we remove them from the training data. This is because many unsupervised methods try to learn the normal log patterns and alert anomalies when such patterns are violated. Thus, they require anomaly-free log data to yield the best performance. Nevertheless, we will evaluate the impact of anomaly samples in training data. For log partition, we apply identifier-based partitioning to HDFS and fixed partitioning with six hours of partition size to BGL. The default values of window size and step size are ten and one, which are set empirically based on our experiments. For HDFS and BGL, we set k as ten and 50, respectively. In particular, a log event sequence will be regarded as an anomaly if any one of its log windows, $\mathcal{W}$, is predicted as anomalous.

5.3.2 Accuracy of Log Anomaly Detection

In this section, we explore the models' accuracy. We first show the results when log event sequences are composed of log events' indices. Then, we evaluate the effectiveness of logs' semantics by incorporating it into the log sequences. Finally, we control the ratio of anomalies in the training data to see its influence.

Accuracy without Log Semantics

The performance of different methods is shown in Table 5.2 (the left-side figures). It is not surprising that supervised methods generally achieve better performance than their unsupervised counterparts do. For HDFS and BGL, the best F1 scores (here-

Table 5.2: Accuracy of DL-based Log Anomaly Detection Methods

Models	HDFS (w/o and w/ semantics)			BGL (w/o and w/ semantics)		
	Precision	**Recall**	**F1 score**	**Precision**	**Recall**	**F1 score**
LSTM [40, 114]	**0.96**/**0.965**	**0.928**/0.904	**0.944**/**0.945**	**0.935**/**0.946**	**0.989**/0.989	**0.961**/**0.967**
Transformer [122]	0.946/0.86	0.867/**1.0**	0.905/0.925	**0.935**/0.917	0.977/**1.0**	0.956/0.957
Autoencoder [44]	0.881/0.892	0.878/0.869	0.88/0.881	0.791/0.942	0.773/0.92	0.782/0.931
Attn. BiLSTM [189]	0.933/0.934	0.989/**0.995**	0.96/0.964	**0.989**/0.989	**0.977**/**0.977**	**0.983**/0.983
CNN [108]	**0.946**/**0.943**	**0.995**/**0.995**	**0.97**/**0.969**	0.966/**1.0**	**0.977**/**0.977**	0.972/**0.989**

after, we mainly talk about this metric unless otherwise stated) that unsupervised methods can attain are 0.944 and 0.961, respectively, both of which come from the LSTM model [40]. On the other hand, supervised methods have pushed them to 0.97 (by CNN [108]) and 0.983 (by attentional BiLSTM [189]), achieving noticeable improvements. Among all unsupervised methods, Autoencoder, the only construction-based model, performs relatively poorly, i.e., 0.88 in HDFS and 0.782 in BGL. Nevertheless, it possesses the merit of great resistance against anomalies in training data, as we will show later. LSTM shows outstanding overall performance, demonstrating its exceptional ability to capture normal log patterns. On the supervised side, CNN and attentional BiLSTM achieve comparable results in both datasets, outperforming unsupervised methods by around 2%.

We also present the results of traditional ML-based methods in Table 5.3 using the toolkit released by He et al. [70], which contains three unsupervised methods, i.e., Log Clustering (LC), Principal Component Analysis (PCA), Invariant Mining (IM), and three supervised methods, i.e., Logistic Regression (LR), Decision Tree (DT), and Support Vector Machine (SVM). For HDFS, Decision Tree achieves a remarkable performance, i.e., 0.998, ranking the best among all. Other traditional ML-based methods are generally defeated by their DL-based counterparts. This is also the case for BGL. Moreover, traditional unsupervised methods seem to be inapplicable for BGL, e.g., the F1 score of

Table 5.3: Accuracy of Traditional ML-based Methods

	HDFS			BGL		
Meth.	**Prec.**	**Rec.**	**F1**	**Prec.**	**Rec.**	**F1**
LC	**1.0**	0.728	0.843	**0.975**	0.443	0.609
PCA	0.971	0.628	0.763	0.52	**0.619**	0.56
IM	0.895	**1.0**	**0.944**	0.86	0.489	**0.623**
LR	0.95	0.921	0.935	0.791	0.818	0.804
DT	**0.997**	**0.998**	**0.998**	0.964	**0.92**	0.942
SVM	0.956	0.913	0.934	**0.988**	0.909	**0.947**

PCA is only 0.56, while unsupervised DL-based methods yield much better results. Particularly, compared with the experiments conducted by He et al. [70], we achieve better results on BGL when running both DL-based and traditional ML-based methods. This attributes to the fact that we apply shuffling to the dataset, which alleviates the issue of unseen logs in BGL's testing data. Note that this is done at the level of log sequences. The order of log events in each input window is preserved.

Accuracy with Log Semantics

To leverage logs' semantics, some work, e.g., [189], adopts off-the-shelf word vectors, e.g., pre-trained on Common Crawl Corpus dataset using the FastText algorithm [80]. Different from them, in our experiments, we randomly initialize the embedding vector for each word as we did not observe much improvement when following their configurations. An important reason is that many words in logs are not covered in the pre-trained vocabulary. Table 5.2 (the second figures) presents the performance when models have access to logs' semantic information for anomaly detection. We can see almost all methods benefit from logs' semantics, e.g., Autoencoder obtains nearly 15% of performance gain. Particularly, the best F1 scores achieved by unsupervised

and supervised methods on the BGL dataset become 0.967 (by LSTM [40]) and 0.989 (by CNN [108]), respectively, while the best F1 scores on the HDFS dataset remain almost unchanged. Nevertheless, the Decision Tree is still undefeated on the HDFS dataset. Logs' semantics not only promotes the accuracy of anomaly detection but also brings other kinds of benefits to the models, as we will show in the next sections.

FINDING 1. *Supervised methods generally achieve superior performance than unsupervised methods do. Logs' semantics indeed contributes to the detection of anomalies, especially for unsupervised methods.*

Accuracy with Varying Anomaly Ratio

In this experiment, we evaluate how the anomalies in training data will impact the performance of unsupervised DL-based methods. The motivation is that some works claim that a small amount of noise (i.e., anomalous instances) in training data only has a trivial impact on the results. This is because normal data are dominant, and the model will forget the anomalous patterns. In our previous experiments, we remove all anomalies from the training data such that the normal patterns could be best learned. However, in reality, anomalies are inevitable. We simulate this situation by randomly putting a specific portion of anomalies (from 1% to 10%) back into the training data. The results on HDFS are shown in Figure 5.1, where we experiment without and with logs' semantics. Clearly, even with just 1% of anomalies, the F1 score of both LSTM and the Transformer drops significantly to 0.634 and 0.763, respectively. Logs' semantics safeguards around 10% of performance against anomalies. When the percentage of anomalies reaches 10%, the F1 score of LSTM even degrades to less than 0.4. Interestingly, Autoencoder exhibits remarkable

resilience against noisy training data, which demonstrates that compared with forecasting-based methods, construction-based methods are indeed able to forget anomalous log patterns.

FINDING 2. *For forecasting-based methods, anomalies in training data can quickly deteriorate performance. Different from them, reconstruction-based methods are more resistant to training data containing anomalies.*

5.3.3 Robustness of Log Anomaly Detection

In this section, we study the robustness of the selected anomaly detectors, i.e., the accuracy with the presence of unseen logs. We also compare them against traditional ML-based methods. To simulate the log instability issue, we follow Zhang et al. [189] to synthesize new log data. Given a randomly sampled log event sequence in the testing data, we apply one of the following four noise injection strategies: randomly injecting a few pseudo log events (generated by trivial word addition/removal or synonym replacement) or deleting/shuffling/duplicating a few existing log events in the sequence. We inject the synthetic log sequences into the original log data according to a specific ratio (from 5% to 20%). With the injected noises, DL-based methods that leverage logs' semantics can continue performing anomaly prediction without retraining. However, their traditional ML-based counterparts need to be retrained because the number of distinct log events is fixed. We follow Zhang et al. [189] to append an extra dimension to the log count vector (for both training and testing data) to host all pseudo log events.

The results of DL-based methods on HDFS are presented in Figure 5.2. Clearly, the performance of all models is harmed by the injected noises. In particular, unsupervised methods are much more vulnerable than supervised methods. For LSTM and

the Transformer, 5% of noisy logs suffice to degrade their F1 score by more than 20%. Logs' semantics offers little help in this case. Autoencoder again demonstrates good robustness against noise and benefits more from logs' semantics. The situations of supervised models are much better. With the access to logs' semantics, they successfully maintain an F1 score of around 0.9 even with 20% noises injected, while that of LSTM and the Transformer are both lower than 0.5. This proves that logs' semantic information indeed helps DL-based models adapt to unprecedented log events. On the side of traditional ML-based methods in Figure 5.3, unsupervised methods are also more sensitive than their supervised counterparts. In particular, SVM and Logistic Regression achieve the best performance, i.e., around 0.8 of the F1 score is retained when the testing data contains 20% noise. Under the same setting, PCA and Invariant Mining have the worst results, i.e., around 0.4 of the F1 score.

FINDING 3. *Unprecedented logs have a significant impact on anomaly detection. Supervised methods exhibit better robustness against such logs than unsupervised methods. Moreover, logs' semantics can further promote robustness.*

5.3.4 Efficiency of Log Anomaly Detection

In this section, we evaluate the efficiency of different models by recording the time spent on the training and testing phases for both datasets. The results are given in Figure 5.4, where we do not consider logs' semantics. We can see that each model generally requires tens of seconds for model training and around five seconds for testing. BGL consumes less time due to its smaller volume. For HDFS, LSTM and Autoencoder are the most time-consuming models for training, while for BGL, supervised models demand more time. On the other hand, some traditional ML-

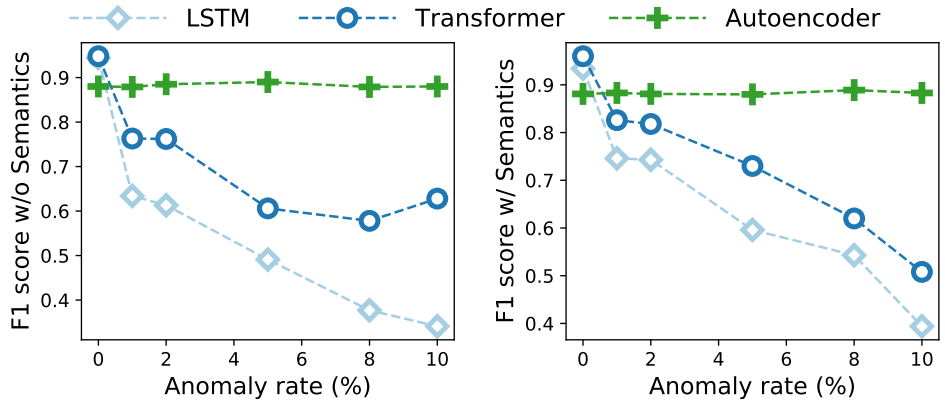


Figure 5.1: Accuracy w/ Varying Anomaly Ratio in Training Data

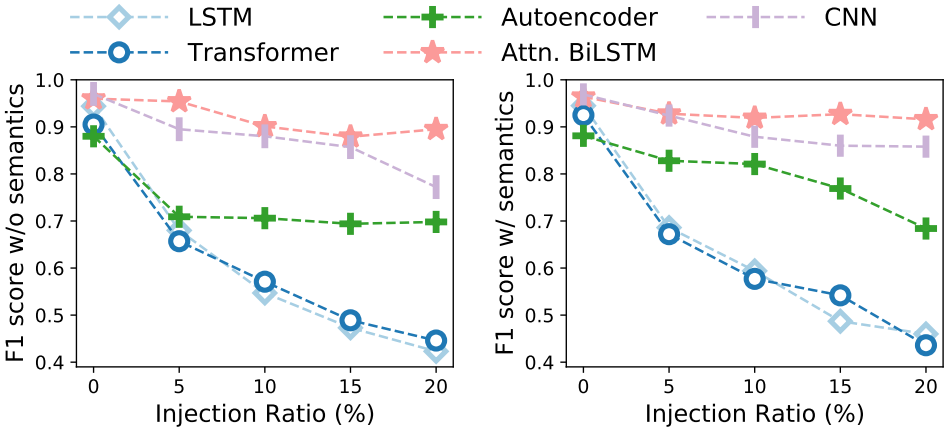


Figure 5.2: Robustness of DL-based Methods on HDFS

based methods, i.e., Logistic Regression, Decision Tree, SVM, and PCA show superior performance over DL-based models, which only take seconds for model training. SVM and PCA can even produce results in a real-time manner. However, Invariant Mining consumes thousands of seconds for pattern mining on HDFS. Regarding model testing, except for Log Clustering, other methods only require tens of milliseconds.

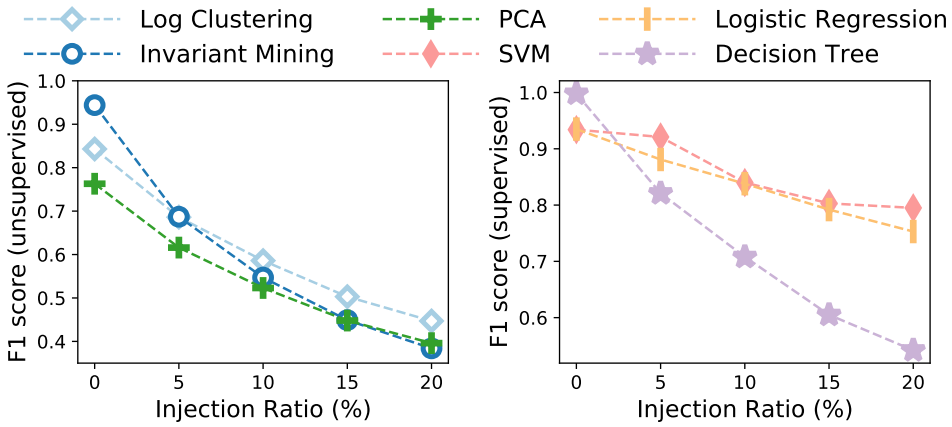


Figure 5.3: Robustness of ML-based Methods on HDFS

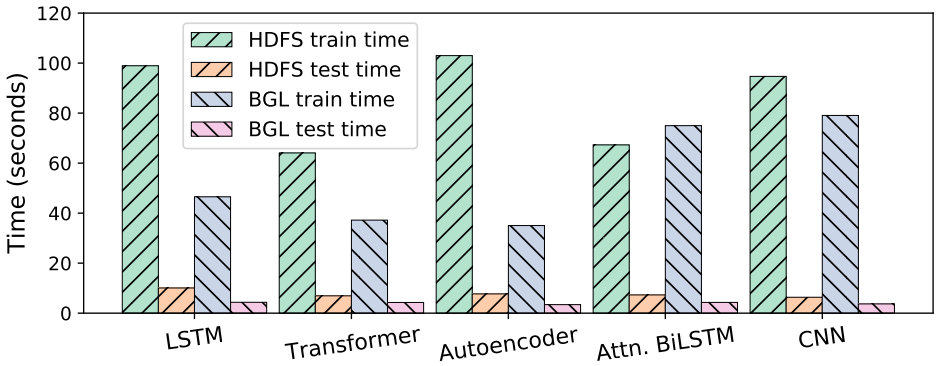


Figure 5.4: Efficiency on Both HDFS and BGL Datasets

FINDING 4. *Compared to traditional ML-based methods, DL-based methods often require more time for training and testing. Some ML-based methods demonstrate outstanding efficiency.*

5.4 Industrial Practices

In this section, we present an industrial case study of deploying automated log-based anomaly detection in production at Huawei Cloud. The model is an optimized version based on DeepLog [40]. DeepLog was selected for its simplicity and superior performance

(see Table 5.2). It is one of the most highly-cited papers in the field and serves as the prototype for many follow-up models. Services in Huawei Cloud serve hundreds of millions of users worldwide and produce terabytes of log data on a daily basis. Such a large volume of data is impractical for engineers to detect anomalies manually. Thus, automated log anomaly detection is in high demand.

To share our industrial practices, we first introduce the deployment architecture of our model. Then, we summarize the real-world challenges that we met during its usage in production. Finally, we discuss some promising future improvements that can push this field forward beyond current practices.

5.4.1 Industrial Deployment

Based on our experience of model deployment and operations, we have summarized the following challenges in production systems, which are ubiquitous in the entire industry [37]. We conclude that although some successful adoptions have been made, the current research on log-based anomaly detection is inadequate to combat the challenges of pursuing highly intelligent and reliable log-based anomaly detection.

Data-Specific Challenges

Fundamental support for the human-surpassing capabilities of deep learning is adequate and high-quality log data, which however is not always available, especially in cloud environments.

High Complexity. The production logs present great complexity. For example, logs can be interleaving [120, 68] due to the concurrent execution of multiple tasks. There can be multiple types of logs across a software system, e.g., testing logs, trace logs, and business logs. They profile the system's performance and health status from different perspectives and thus may need

tailored model design. The possibility of combining them also remains an unexplored problem.

Large-Volume and Low-Quality Data. Although a massive amount of log data has been generated, a significant portion of logs only records plain system runtime information. Moreover, there is currently a lack of rigorous guides and specifications on developer logging behaviors. Thus, logs exhibit different styles across different service teams and often contain meaningless tokens, making it hard to design an appropriate log parser for feature extraction. Production logs may possess limited semantics, i.e., containing only the service resources in a REST style [163]. As a result, only superficial features (e.g., count) of log events are used [194, 70].

Concept Drift. Modern software systems continuously undergo feature upgrades, which may incur concept drift [52]. For example, new log events may emerge, log patterns may change. As a result, the threshold for anomaly assertion may need adjustment from time to time after the model has been deployed. The optimal setting obtained in offline training may not always be applicable for online deployment. Moreover, without proper online learning capability, frequent model retraining is required.

Data Management Issues. Large-scale software systems can produce terabytes of log data on a daily basis [116]. How to efficiently manage such a large amount of data become very challenging. Common data management tasks include data storing, sharing, querying, etc. In the literature, efforts have been devoted to fast logging infrastructure [111, 110], log compression [103, 178], and scalable search on compressed logs [141]. However, the fundamental question - of which logs are important and should be kept, is still hard to answer. Ding et al. [39] made a good attempt to identify useful logs by considering the execution time of code blocks.

Model-Specific Challenges

While DL techniques are transforming the whole industry, we recognize some challenges when driving their adoption in the company.

Insufficient Interpretability. Although DL-based methods possess better interpretability, e.g., specifically locating the problematic logs, they are still unable to explain the occurrence of anomalies from the system perspective. Learning logs' semantics can be a promising direction to enhance interpretability further. However, many logs themselves are meaningless. Tokens like abbreviations and self-defined words are hardly understandable to other engineers.

Difficult Model Selection. There are mainly two types of anomalies in log data [18, 164], i.e., point anomalies and contextual anomalies. The first type talks about the occurrence of particular error logs, while the second can be the unexpected order of log events. We found some engineers have a strong AI-solves-everything mindset and lean towards AI solutions regardless of the specific problem scenarios. However, for point anomalies, simply maintaining a set of important log events would be sufficient. Even if AI solutions indeed suit better the problem at hand, the selection of an appropriate model architecture is painful. This is what motivates our work.

Operational Challenges

We also present some operational issues regarding the promotion of DL-based log anomaly detection.

Immature Engineering Support. AI-oriented engineering best practices and principles are not established. Building AI solutions requires substantial engineering support, e.g., model selection, important feature identification, and data labeling. On the other hand, to facilitate the manual labeling effort, some tools

should be developed to infer the label automatically based on engineers' responses to different system statuses. For example, IBM Cloud has designed a help-desk software solution [124] to record and track how engineers react to the problems that occur in the monitored system. The labels can then be obtained programmatically based on the temporal and spatial correlation of this information source.

Privacy Issues. Due to privacy policies, it is prohibitive to access some customers' logs. Without some detailed log information, prompt anomaly detection becomes a more significant challenge. We encounter log privacy issues when trying to access customers' logs, which is required for proactive failure detection. Without proper access rights, we cannot conduct failure diagnoses for customers.

5.4.2 Future Directions

Closer Engineering Collaboration

Companies need to establish and align on a clear objective at the executive level. The infrastructure development, service architecture design, and engineers' mindsets should serve this clear objective collaboratively. In current practices, logs are collected from different services in an ad-hoc manner and used for independent model development. A pipeline of log data generation, collection, labeling, and usage should be built. In this process, the engineering principles should include data/label sanity check and continuous model-quality validation.

Better Logging Practices

Good-quality log data play an essential role in many downstream log analytics tasks, including anomaly detection, failure diagnosis, performance optimization, user profiling, etc. Thus, better logging practices should be established to guide the writing of

logging statements. Some examples are: 1) a logging statement should include a timestamp, a proper verbosity level, etc.; 2) the context should be made clear by including critical variables, components, process IDs, etc.; 3) a log message should be meaningful and understandable to its audience; 5) apply template-based logging [35] that is human-friendly and machine-readable; 6) the number of logging statements should be kept in a proper level. Other studies include rule-based logging [90], diagnosability-oriented logging [184, 183], and log statement auto-completion [113].

Log Quality Assessment

Being able to assess the quality of logs [172, 83] could be a promising solution to boost the performance of log data management and downstream tasks. We have summarized the following properties that good-quality logs should possess: 1) less performance overhead (e.g., small CPU resources, little energy consumption) [186]; 2) long survival unless feature upgrades (i.e., the logs will not be modified or removed across different software versions) [19, 65, 148]; 3) a good system failure indicator (i.e., effectively tell the occurrence and root cause of failures). The last property can be evaluated via fault injection [34].

5.5 Summary

In this chapter, we conduct a detailed review of five popular neural networks for log-based anomaly detection and evaluate six state-of-the-art methods in terms of accuracy, robustness, and efficiency. Particularly, we explore whether logs' semantics can bring performance gain and whether they can help alleviate the issue of log instability. We also compare DL-based methods against their traditional ML-based counterparts. The results demonstrate

that logs' semantics indeed improves models' robustness against noises in both training and testing data. Furthermore, we release an open-source toolkit of the studied methods to pave the way for model customization and improvement for both academia and industry.

□ **End of chapter.**

Chapter 6

Adaptive Performance Anomaly Detection

Online service systems are closely monitored with various metrics (e.g., service response delay) on a 24×7 basis. This is because they often serve as the most direct and fine-grained signals that flag the occurrence of service performance issues. In addition, they provide informative clues for engineers to pinpoint the root causes. However, due to the large scale and complexity of online service systems, the number of metrics is overwhelming the existing troubleshooting systems [29]. Automated anomaly detection over the metrics, which aims to discover the unexpected or rare behaviors of the metric time series, is therefore important for reliability assurance. In this chapter, we present ADSketch, a solution to address the main shortcomings of previous approaches, i.e., the lack of interpretability and adaptability. ADSketch aims to tackle the pain points of flooding alarms and gray failures introduced in Chapter 4. The remainder of this chapter is organized as follows. Section 6.1 introduces the problem background and the contributions we made. Section 6.2 elaborates on the algorithms for adaptive performance anomaly detection. Section 6.3 presents the experiments and results. Section 6.4 shares our success stories and some case studies. Finally, Section 6.5 summarizes this chapter.

6.1 Problem and Contributions

Although many efforts, e.g., [190, 153, 69], have been devoted to performance anomaly detection, most of the existing work does not possess the merit of interpretability. Specifically, at each timestamp, they calculate a probability indicating the likelihood of anomalies. A threshold is then chosen to convert the probability into a binary label – normal vs. anomaly. However, in reality, a simple recommendation of the suspicious anomalies might not be of much interest to engineers. This is because they need to manually investigate the problematic metrics (recommended by the model) for fault localization. For large-scale online services, this process is like finding a needle in a haystack. The problem is compounded by the fact that false alerts are not rare. Moreover, many state-of-the-art methods train models with historical metric data in an offline setting. As online services continuously undergo feature upgrades and system renewal, the patterns of metrics may evolve accordingly [52, 62]. Without adaptability, these models are unable to accommodate the ever-changing services and user behaviors.

In this Chapter, we present ADSketch[1], a performance anomaly detection approach for online service systems based on *pattern sketching*, which is interpretable and adaptive. The main idea is to identify discriminative subsequences from metric time series that can represent classes of service performance issues. This is similar to the problem of shapelet discovery in time series data [138, 179]. Particularly, for multiple subsequences that describe the same type of performance issue, we take the average of them and regard the result as a *metric pattern* for the issue. For example, services may be experiencing performance degradation when we observe a level shift down in Service Throughput or a level shift up in CPU Utilization. The advantages of such metric

[1] `https://github.com/OpsPAI/ADSketch`

patterns are twofold. First, the normality of the incoming metric subsequences can be quickly determined through a comparison with the metric patterns. Second, by associating the patterns with typical anomaly symptoms, we can immediately understand the ongoing performance issues when the metric subsequences exhibit known patterns. This is similar to failure/issue profiling [133, 101, 109]. In this way, ADSketch provides a novel mechanism to characterize service performance issues with metric time series. Previous work on failure/issue profiling often requires handcrafted features, which suffers from limited generalization. For example, Brandon et al. [13] manually defined a set of features collected from metrics, logs, and anomalies to characterize failures. Pattern sketching with metrics enjoys the advantages of automation and accuracy. Moreover, ADSketch is able to adaptively embrace new anomalous patterns when detecting anomalies on the fly. Experimental results demonstrate the superiority of our design over the existing state-of-the-art time series anomaly detectors on both public and industrial data. In particular, we have achieved an average F1 score of over 0.8 in production systems.

The goal of this work is to detect performance anomalies for modern software systems, especially online service systems, based on monitoring metrics. To facilitate issue understanding and problem mitigation, we intend to improve the interpretability of the detection results. To this end, we propose to sketch performance issues with metrics based on our observation that similar issues often exhibit alike patterns. By extracting such anomalous metric patterns, we can conduct performance anomaly detection by examining whether the incoming metric subsequences match the known patterns. Moreover, by associating the extracted metric patterns with specific performance issues, we can obtain a quick understanding of the ongoing issues in online scenarios. Additionally, as online services are continuously evolving, un-

precedented metric patterns may emerge. Thus, our algorithm should be adaptive to the new patterns. The problem can be formally defined as follows.

The input of a metric time series can be represented as $\mathcal{T} \in \mathbb{R}^l = [t_1, t_2, ..., t_l]$, where l is the number of observations. $t_i^m = [t_i, ..., t_{i+m-1}]$ is a consecutive subsequence of $\mathcal{T}$ starting from t_i with length m, where $i \in [0, l-m]$. The objective of performance anomaly detection is to determine whether or not a given t_i^m is anomalous, i.e., whether there are performance issues happening from timestamp i to $i+m-1$. Particularly, we also try to explain the type of performance issues associated with t_i^m. The anomalous subsequences will be used to construct abnormal metric patterns, while the benign ones will be regarded as normal patterns. Both the normal and abnormal metric patterns will be updated as anomaly detection proceeds.

6.2 Methodology

6.2.1 Overview

In online service systems, performance anomalies often serve as the (early) signals for critical failures, which should be detected effectively. However, accuracy alone is far from satisfactory, as it will be labor-intensive to manually investigate the problematic metrics for issue understanding. ADSketch facilitates this process by providing prompt anomaly alerts with explanations.

The overall framework of ADSketch is shown in Figure 6.1, which consists of two phases, namely, *offline anomaly detection* and *online anomaly detection*. In the offline phase, ADSketch takes as input a pair of metric time series. One metric time series is anomaly-free, which serves as the basis for detecting anomalies in the other metric (if any). In this process, a set of metric patterns will be automatically learned. A metric pattern

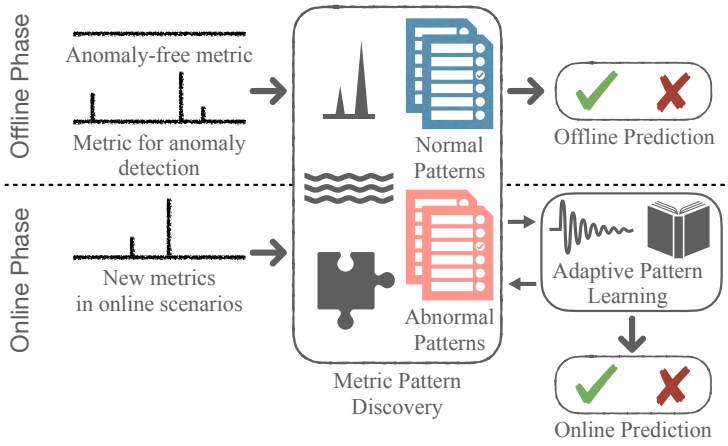


Figure 6.1: The Overall Framework of ADSketch

is essentially the mean of a set of similar metric subsequences representing similar service behaviors. The identified metric patterns are divided into two types, i.e., normal and abnormal. The abnormal patterns often characterize some particular types of performance issues, as discussed in Section 2.5.1. Thus, by investing manual efforts to link them to the corresponding issues, a clearer picture of the underlying problems can be easily obtained if similar patterns are encountered again. In the online phase, we leverage the metric patterns built in the offline phase to conduct anomaly detection in online scenarios, where metrics arrive in streams. Particularly, in production environments, unprecedented patterns could appear. Thus, we design an adaptive learning algorithm to capture the new patterns continuously.

Before formally introducing our algorithms, we have summarized the variables involved in Table 6.1.

6.2.2 Offline Anomaly Detection

Metric Pattern Discovery

The idea for discovering abnormal patterns follows the basic definition of an anomaly: if a metric subsequence deviates significantly from those collected during a service's normal executions,

Table 6.1: Summary of Variables

Variable	Meaning
$\mathcal{T}_n$	An anomaly-free metric time series
$\mathcal{T}_a$	An input metric time series for anomaly detection
t	A subsequence of metric time series
m	The length of the metric subsequence t
p	The percentile threshold to find deviated subseqs
$\mathcal{P}_n$	The index set of normal metric patterns
$\mathcal{P}_a$	The index set of anomalous metric patterns
μ_C	The vector of cluster mean vectors
$\mathcal{S}_C$	The vector of cluster sizes
$\mathcal{R}_C$	The vector of cluster radii

it is likely that the subsequence captures some misbehaving moments of the service. To measure how deviated a metric subsequence is, we calculate its distance to other subsequences and search for the smallest distance score. Intuitively, metric subsequences which have large scores to others tend to be anomalous. The function for distance measure is customizable, and we adopt Euclidean distance, which is a popular choice.

Given a metric time series with l observations, the number of all possible subsequences is $l - m + 1$, where m is the length of its subsequences. A naïve solution for calculating the smallest pair-wise distance (which we refer to as *SPW distance* hereafter) would be brute force searching. However, this algorithm owns a quadratic time complexity, which is practically infeasible for large time series. Fortunately, some novel scalable algorithms [177, 179, 205] have been proposed in the literature to attack such all-pairs-similarity-search problems for time series subsequences. Particularly, Yeh et al. [179] proposed STAMP, which has achieved orders of magnitude faster compared to state-of-the-art methods. For exceptionally large datasets, an ultra-fast approximate solution is also provided. An illustrating example

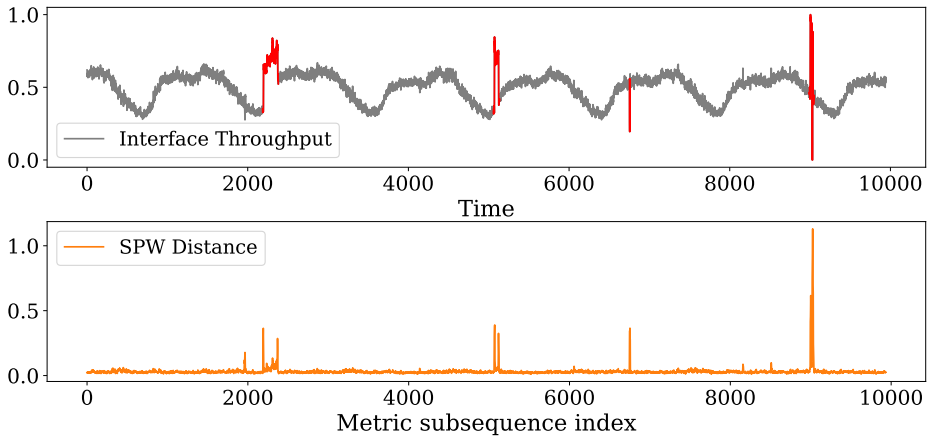


Figure 6.2: The SPW Distance of Different Metric Subsequences

is provided in Figure 6.2, where we can see the misbehaving metric subsequences have larger SPW distances. In particular, the original STAMP algorithm adopts z-normalization for data preprocessing. However, we found that min-max normalization yields more meaningful results in our scenario. For a subsequence t_i^m in a metric time series $\mathcal{T}$, we record the index and distance score of another subsequence having the SPW distance to it. Such index and score of all subsequences, i.e., t_i^m $(i \in [0, l-m])$, constitute two vectors $\mathcal{I}$ and $\mathcal{S}$. In particular, for t_i^m, its closest subsequence can either come from the same time series (i.e., self-union) or another time series (i.e., cross-union). In the first case, a trivial match region around t_i^m will be excluded to avoid self-matches [179].

The proposed algorithm for metric pattern discovery is presented in Algorithm 1, which is illustrated in Figure 6.3. Algorithm 1 takes as input two metric time series, i.e., $\mathcal{T}_n$ and $\mathcal{T}_a$ ($\mathcal{T}_n$ is anomaly-free and $\mathcal{T}_a$ may contain anomalies to be detected), and two hyper-parameters, i.e., m and p (m is the length of subsequences and p is the percentile threshold to find the deviated subsequences). As production service systems are mostly running in normal status [29], the anomaly-free input is easily obtainable (we discuss how we address the violating cases in

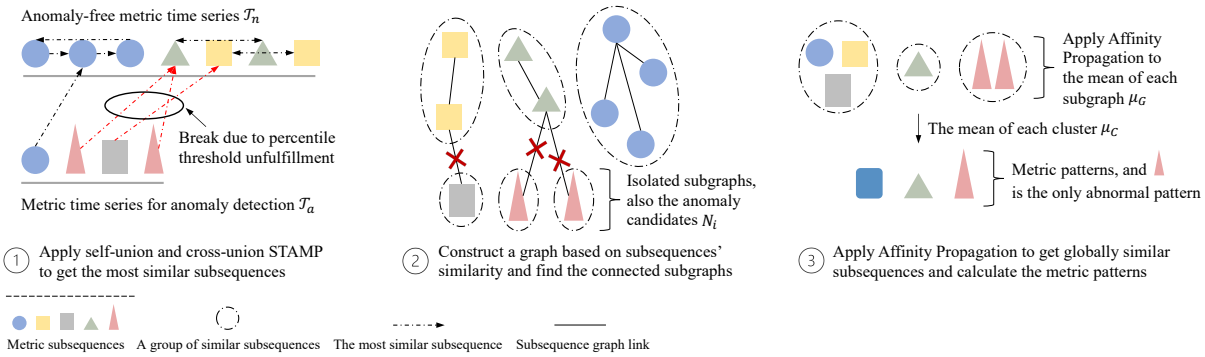


Figure 6.3: The Algorithm of Performance Anomaly Pattern Discovery

Section 6.4.3). In line 1 of Algorithm 1, we apply STAMP to $\mathcal{T}_n$ with *self-union* (i.e., similar subsequences come from $\mathcal{T}_n$), and obtain the index and score vectors $\mathcal{I}_{nn}$ and $\mathcal{S}_{nn}$. In line 2, we search similar subsequences for $\mathcal{T}_a$ from $\mathcal{T}_n$, i.e., *cross-union*, and get $\mathcal{I}_{na}$ and $\mathcal{S}_{na}$. Intuitively, given the fact that $\mathcal{T}_n$ is anomaly-free, subsequences in $\mathcal{T}_a$ having large SPW distances to their closest peers in $\mathcal{T}_n$ are suspected to be anomalous. Interestingly, we later learn that Mercer et al. [115] proposed a similar idea concurrently. We introduce a percentile threshold (i.e., p) on $\mathcal{S}_{na}$ to find such deviated subsequences. In particular, p is loosely set to avoid missing anomalies, i.e., false negatives. Such a setting will inevitably produce false positives. We next discuss how we alleviate this issue.

A metric pattern is defined as the mean of a group of similar subsequences, which represents some typical behaviors of the metric time series. To mine similar subsequences, we propose to leverage their similarity connections. Specifically, in line 3, we construct a graph G whose nodes correspond to the subsequences. Two nodes will be linked if any one of them is deemed as the most similar subsequence to the other, as indicated by $\mathcal{I}_{nn}$ and $\mathcal{I}_{na}$. Note such a relationship is not mutual, i.e., t_i^m is the most similar to t_j^m does not necessarily imply the opposite case. We break the edges whose distance score fails to meet the threshold

Algorithm 1 Performance Anomaly Pattern Discovery

Input: $\mathcal{T}_n$, $\mathcal{T}_a$, m, and p
Output: Two disjoint sets of $\mathcal{P}_n$ and $\mathcal{P}_a$

1 $\mathcal{I}_{nn}, \mathcal{S}_{nn} \leftarrow \text{STAMP}(\mathcal{T}_n, \mathcal{T}_n, m)$
2 $\mathcal{I}_{na}, \mathcal{S}_{na} \leftarrow \text{STAMP}(\mathcal{T}_n, \mathcal{T}_a, m)$
3 $G \leftarrow \text{ConnectedSubgraphs}(\mathcal{I}_{nn} + \mathcal{I}_{na}, \mathcal{S}_{na}, p)$
4 $N_i \leftarrow \text{IsolatedNodes}(G)$
5 $\mu_G \leftarrow \text{GraphWiseMean}(G)$
6 $C \leftarrow \text{AffinityPropagation}(\mu_G)$
7 $\mu_C \leftarrow \text{ClusterWiseMean}(C)$
8 $\mathcal{P}_n \leftarrow \text{EmptyArray}, \mathcal{P}_a \leftarrow \text{EmptyArray}$
9 **for** *each* idx *in* $1 : \text{Size}(C)$ **do**
    // $C[idx]$: `all subsequences in the cluster`
10 **if** $C[idx] \subset N_i$ **then**
11 $\mathcal{P}_a \leftarrow$ Append $\mathcal{P}_a$ with idx
12 **else**
13 $\mathcal{P}_n \leftarrow$ Append $\mathcal{P}_n$ with idx
14 **end**
15 **end**

requirement p. The above operations are depicted in the first part of Figure 6.3. Next, we find the connected subgraphs of G, each of which is composed of subsequences resembling each other. Particularly, there will be some isolated nodes, i.e., subgraphs with a single node, which are collected at line 4. Such deviated subsequences constitute a set of anomaly candidates, i.e., N_i. The second part of Figure 6.3 illustrates this process.

Up to this point, we have divided the subsequences of $\mathcal{T}_n$ and $\mathcal{T}_a$ into different parts, each of which is represented as a subgraph. However, each subgraph cannot be directly regarded as a metric pattern because: 1) the graph construction criteria can be too strict (i.e., only the most similar pairs are connected), so some subgraphs might still be similar; 2) the loosely set percentile threshold p may flag some normal subsequences as abnormal (i.e., false positives). To further combine the similar subsequences, we apply the Affinity Propagation algorithm [49]

Algorithm 2 Performance Anomaly Detection

Input: t, $\mathcal{P}_a$, and μ_C
Output: Anomaly detection result for t

```
16 D_t ← PairWiseDistance(t, μ_C)
17 idx ← MinIndex(D_t)
18 if idx ∈ P_a then
19 |   return True
20 else
21 |   return False
22 end
```

to cluster the mean vector of each subgroup (line 5-6). We choose this algorithm because of its superior performance and efficiency, and it requires no pre-defined cluster number. As a result, similar normal subgraphs can be merged together, and abnormal subgraphs have a chance to embrace their normal communities. Thus, each cluster will contain all similar subsequences across the two time-series inputs, and different clusters represent distinct patterns. The mean of clusters (i.e., μ_C) will form the set of metric patterns (line 7). For each cluster, we check whether or not all its members come from the set of anomaly candidates N_i (line 9-15). If yes, the mean of the cluster will be regarded as an abnormal metric pattern and otherwise normal, indexed by $\mathcal{P}_a$ and $\mathcal{P}_n$, respectively. The third part of Figure 6.3 presents the above operations. Finally, all subsequences in the anomalous clusters will be predicted as an anomaly to be the output of this phase.

Metric Pattern Interpretability

In this section, we expound on how to label the performance issues that each metric pattern represents. By allowing metric patterns to have semantics, the understanding and mitigation of service problems can be greatly accelerated. Given the fact that the duration of different performance issues may vary, our

fixed-length metric patterns may over-represent (i.e., the metric pattern is much larger than the issue's duration) or under-represent (i.e., the metric pattern is only an excerpt of the issue) the corresponding issues. To alleviate the first problem, we select a relatively small m, which turns out to be aligned with the goal of better performance. For the second problem, we adopt the following strategy to group clusters which are actually describing a common issue. For each pair of clusters, we check whether they have some subsequences that share some parts in common. All clusters sharing such overlaps together can recover the complete picture of the issue. Thus, we regard them as describing an identical issue. Finally, for each metric pattern, domain engineers will label the type of performance issue that triggers it. Particularly, one pattern can have multiple labels simultaneously. The metric patterns with overlaps will share the same set of performance issue labels.

6.2.3 Online Anomaly Detection

Anomaly Detection on the Fly

Based on the metric patterns identified in Algorithm 1, we now describe our algorithm (Algorithm 2) for anomaly detection in online scenarios. The idea is straightforward: given a new metric subsequence t with length m, we search for its most similar metric pattern (line 1-2) and check which pattern pool it comes from. If t is more similar to an abnormal pattern, it will be predicted as anomalous; otherwise, normal (line 3-7). In real-world systems where monitoring metrics are generated in a stream manner, this process is continuously running for all coming subsequences. When an anomaly is identified, we would like to provide more interpretation about it, e.g., what kinds of performance issues have happened. This is done by simply recommending the issues associated with the most similar metric pattern for all involved

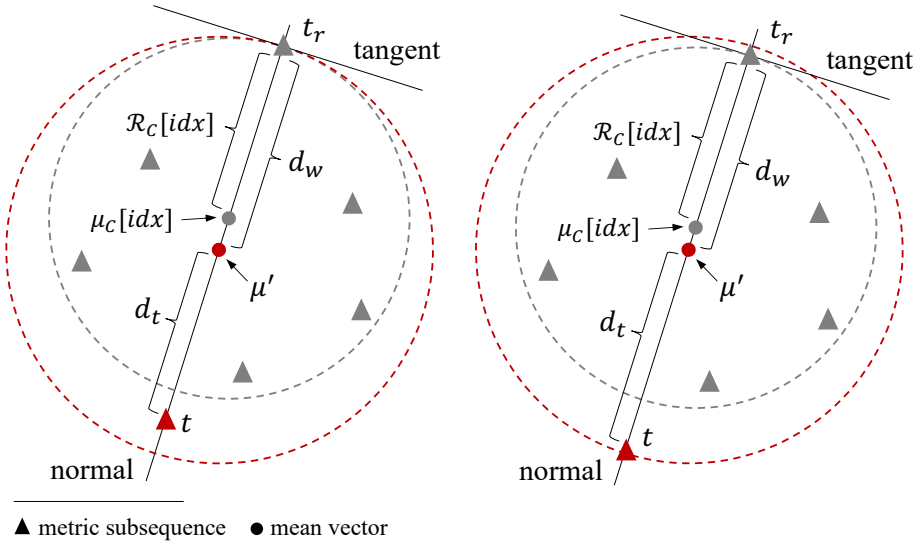


Figure 6.4: The Update of the Radius of a Cluster

metrics. Particularly, in Algorithm 1, each cluster (i.e., C at line 6) contains all subsequences that are deemed as similar. The design of our online anomaly detection only requires the mean vector of each cluster, i.e., μ_C. Thus, instead of keeping all its members (which is storage-intensive), the clusters can be simply represented by their mean vectors.

Note that offline and online anomaly detection can work collaboratively as a performance anomaly detector without the interpretability component, which requires human intervention.

So far the metric patterns for anomaly detection are discovered based on historical data. However, due to the dynamics of online service systems (e.g., software upgrade, customer behavior change), the metrics may experience concept drift [62, 52], which produces brand-new patterns. Thus, an adaptive learning mechanism is desirable to help adapt to such unprecedented patterns and update the metric patterns accordingly. In the next section, we will introduce the algorithm to this end called adaptive pattern learning.

Algorithm 3 Adaptive Pattern Learning

Input: t, $\mathcal{P}_n$, $\mathcal{P}_a$, μ_C, $\mathcal{S}_C$, and $\mathcal{R}_C$

Output: Updated variables: $\mathcal{P}_n$, $\mathcal{P}_a$, μ_C, $\mathcal{S}_C$, and $\mathcal{R}_C$

$\mathcal{D}_t \leftarrow \text{PairWiseDistance}(t, \mu_C)$

$idx \leftarrow \text{MinIndex}(\mathcal{D}_t)$

$\mu^{'} \leftarrow (\mu_G[idx] \times \mathcal{S}_C[idx] + t)/(\mathcal{S}_C[idx] + 1)$

$d_w \leftarrow \text{Distance}(\mu_C[idx], \mu^{'}) + \mathcal{R}_C[idx]$

$d_t \leftarrow \text{Distance}(t, \mu^{'})$

$d^{'} \leftarrow \text{Max}(d_t, d_w)$

$d_n, d_a \leftarrow \text{Max}(\mathcal{R}_C[\mathcal{P}_n]), \text{Max}(\mathcal{R}_C[\mathcal{P}_a])$

if $idx \in \mathcal{P}_a$ **then** $d \leftarrow d_a$ **else** $d \leftarrow d_n$ **end**

if $\mathcal{D}_t[idx] < d$ **then**

    `// add` t `to the most similar cluster`

    $\mu_C[idx], \mathcal{S}_C[idx], \mathcal{R}_C[idx] \leftarrow \mu^{'}, \mathcal{S}_C[idx] + 1, d^{'}$

    if $\mathcal{S}_C[idx] > \text{Max}(\mathcal{S}_C[\mathcal{P}_a])$ and *idx is a new cluster* **then**

        $\mathcal{P}_n \leftarrow$ Append $\mathcal{P}_n$ with idx

        $\mathcal{P}_a \leftarrow$ Remove idx from $\mathcal{P}_a$

    else

        $d \leftarrow \text{Max}(d, d^{'})$

        `//` d `will be assigned to` d_n `or` d_a `accordingly`

    end

else

    `// create a new anomalous cluster for` t

    $\mathcal{P}_a \leftarrow$ Append $\mathcal{P}_a$ with $\text{Length}(\mu_C) + 1$

    $\mu_G \leftarrow$ Append μ_G with t

    $\mathcal{R}_C \leftarrow$ Append $\mathcal{R}_C$ with 0

    $\mathcal{S}_C \leftarrow$ Append $\mathcal{S}_C$ with 1

end

Adaptive Pattern Learning

The algorithm of adaptive pattern learning is presented in Algorithm 3, which automatically updates metric patterns during streaming anomaly detection. To start with, for each cluster, we calculate its size and the maximum distance between its mean vector and all members (which we refer to as *radius*), denoted as $\mathcal{S}_C$ and $\mathcal{R}_C$, respectively. In particular, the size and radius of clusters with only a single member are one and zero. For adaptive pattern learning, all clusters can be sufficiently represented with the following properties: μ_C, $\mathcal{S}_C$, and $\mathcal{R}_C$. All subsequences can be discarded.

The main idea is that given a new subsequence t, we determine whether it possesses a known metric pattern carried by an existing cluster. If yes, the cluster will absorb t as a new member and update its properties; otherwise, a brand-new anomalous cluster with only t itself will be created, representing an unseen metric pattern. Specifically, we first search for the closest pattern of t (line 1-2). Then, we determine whether t should become a new member to the corresponding cluster by checking if the distance $\mathcal{D}_t[idx]$ is smaller than the largest radius recorded in all clusters, i.e., $\mathcal{D}_t[idx] \leq \text{Max}(\mathcal{R}_C)$. If it is the case, t should be considered as an old pattern; otherwise, it should be expressing a new pattern.

When a cluster accepts a new member (line 9-16), we need to update its mean vector $\mu_C[idx]$ (i.e., the metric pattern), size $\mathcal{S}_C[idx]$, and radius $\mathcal{R}_C[idx]$. For $\mu_C[idx]$, it can be precisely updated by the equation at line 3 (i.e., $\mu^{'}$). $\mathcal{S}_C[idx]$ can be trivially updated by increasing itself by one. The update of the radius $\mathcal{R}_C[idx]$ is a bit problematic. We cannot directly calculate the new radius as the original subsequences are not available. To address this problem, we employ the worst-case distance for approximation. As shown in Figure 6.4, the new radius reaches its maximum value when t lies in the (inward-pointing) normal

of the tangent space at the member yielding the radius (denoted as t_r) [11], which can be calculated by the equation at line 4. We omit the proof, which is standard. Two cases are possible. The first (the left subfigure) is that t_r continues to be the farthest member from the new mean $\mu^{'}$. The second (the right subfigure) is that t takes the place of t_r and becomes the farthest one. Therefore, besides d_w, we also compute the distance between t and $\mu^{'}$, i.e., d_t, and compare them (line 4-6). The bigger one will be the new radius (line 10). Recall we need to check if $\mathcal{D}_t[idx] \leq \text{Max}(\mathcal{R}_C)$ to decide whether or not t should be taken as a new member. Considering the high imbalance between normal and abnormal clusters, we maintain two maximum radii for them, denoted as d_n and d_a, respectively (line 7). Once a cluster alters its radius, we reset the maximum radius of its kind (d_n or d_a as determined by line 8) if it is exceeded by $d^{'}$ (line 15). On the other hand, if the cluster rejects t, we form a new anomalous cluster containing only t by properly setting its properties (line 18-21).

An issue with this strategy is that false positives will accumulate in $\mathcal{P}_a$ as the unseen patterns can also be normal. We alleviate it by setting a threshold to the size of the newly-formed anomalous clusters (line 11). The role of the cluster will be switched from abnormal to normal if its size exceeds the threshold (line 12-13). The rationale is that performance anomalies are generally rare events. A large anomalous cluster would mean the particular type of issue it represents occurs too often. However, a pattern with a large frequency tends to be the metric's normal behavior. We simply set the default threshold as the largest size of the anomalous clusters identified in the offline stage, i.e., $\text{Max}(\mathcal{S}_C[\mathcal{P}_a])$. Nevertheless, more sophisticated strategies can be applied by, for example, considering the distribution of clusters' sizes.

6.2.4 Time and Space Complexity

Time Complexity

For Algorithm 1, the theoretical time complexity of operation STAMP is $\mathcal{O}(n^2)$. Thus, line 1-2 require $\mathcal{O}(l_n^2)$ and $\mathcal{O}(l_a^2)$, respectively, where l_n and l_a are the length of $\mathcal{T}_n$ and $\mathcal{T}_a$. Another operation with an interesting time complexity is the affinity propagation algorithm (line 7), whose complexity is quadratic in the number of clusters (which is often small), i.e., $\mathcal{O}(|C|^2)$. Other operations are of trivial linear time complexity, which is also the case for Algorithm 2 and Algorithm 3. Overall, ADSketch owns a time complexity of $\mathcal{O}(n^2)$ ($\mathcal{O}(l_n^2 + l_a^2 + |C|^2)$). Fortunately, unlike other models such as deep neural networks, STAMP can be embarrassingly parallelized by distributing its unit operation (SPW distance calculation) to multi-core processors [179]. Moreover, STAMP has an ultra-fast approximation to generate results in an anytime fashion.

Space Complexity

As described in Section 44, pattern clusters have a lightweight representation, i.e., μ_C, $\mathcal{S}_C$, and $\mathcal{R}_C$. We also need $\mathcal{P}_n$ and $\mathcal{P}_a$ to distinguish anomalous patterns from the normal ones. Besides μ_C whose space complexity is $\mathcal{O}(m \times |C|)$, other vectors are of $\mathcal{O}(|C|)$. Therefore, the dominant term of space complexity is $\mathcal{O}(m \times |C|)$. Since both m and $|C|$ are usually small, the space overhead of ADSketch can be considered trivial.

6.3 Experiments

In this section, we evaluate ADSketch using both public data and real-world metric data collected from the industry. Particularly, we aim to answer the following research questions.

RQ1: How effective is ADSketch's offline anomaly detection?

Table 6.2: Dataset Statistics

Dataset	#Curves	#Points	Anomaly Ratio
Yahoo	67	94,866	1.8%
AIOps18	58	5,922,913	2.26%
Industry	436	4,394,880	1.07%

RQ2: How effective is ADSketch's online anomaly detection?

RQ3: How effective is ADSketch's adaptive pattern learning?

The evaluation process of much existing work, e.g., [153, 139], essentially corresponds to the process adopted in RQ1 (i.e., the offline anomaly detection phase), because the threshold they select for anomaly alerting is determined by iterating the full range of its possible values. The best results achieved during the iteration process are reported. To fully examine the performance of different methods in online scenarios, we fix models' data and parameters (including the threshold learned in offline mode) as if they are deployed in production systems, i.e., RQ2. The online adaptability of ADSketch will be evaluated in RQ3.

6.3.1 Experiment Setting

Dataset

To evaluate the effectiveness of ADSketch in performance anomaly detection, we conduct experiments on two publicly available datasets. Moreover, to confirm its practical significance, we collect a production dataset from a large-scale online service of Huawei Cloud. Table 6.2 summarizes the statistics of the datasets.

Public dataset. The public datasets for experiments are Yahoo [140] and AIOps18 [2, 139]. Particularly, we do not conduct online anomaly detection on Yahoo due to its limited number of anomalies.

- **Yahoo.** Yahoo released by Yahoo! Research [140] is a benchmark dataset for time series anomaly detection. Part of the dataset is synthetic (which is simulated by algorithmically injecting anomalies), and part of the dataset is collected from the real traffic of Yahoo services. The anomalies in the real dataset are manually labeled. All time series are sampled every hour. In particular, as our goal is detecting performance anomalies for online services, we only use the real dataset, which reflects the real-world service performance issues. For each time series, we select the first 300 data points as the anomaly-free input (any anomalies in it are ignored), and the remaining part as the input for offline anomaly detection. Therefore, for offline anomaly detection, the data are split into training and testing sets in a ratio of 0.2:0.8.
- **AIOps18.** AIOps18 dataset was released by an international AIOps competition held in 2018 [1]. The dataset is composed of multiple metric time series collected from the web services of large-scale IT companies. Particularly, the dataset contains two types of metrics, i.e., service metrics and machine metrics. The service metrics record the scale and performance of the web services, including response time, traffic, and connection errors, while the machine metrics reflect the health status of physical machines, including CPU usage and network throughput. Some metric time series has a sampling interval of one minute, while that of others is five minutes. Each metric has a training and a testing time series, which follows a roughly 1:1 split ratio. Thanks to its large quantity, we follow the following procedure to separate the data for ADSketch offline and online anomaly detection. First, we extract a small part of the training time series that is anomaly-free, which often contains thousands of data points. Then, we use the remain-

der of the training time series for offline anomaly detection. Finally, the whole testing time series will be employed for online anomaly detection. We also compare the performance of online anomaly detection with and without the adaptive learning component.

Industrial dataset. To evaluate ADSketch in production scenarios, we collect various metrics (e.g., Application CPU Usage, Interface Throughput, Request Timeout Number, and Round-trip Delay) from a large-scale online service (we conceal the name for privacy concerns) of Huawei Cloud. The system under study produces millions of metric time series, which contain an abundance of different metric patterns. The number of metric curves collected is 436, which come from multiple instances of virtual machines, containers, and applications of the selected service system. For each metric, we collect one week of data with a sampling interval of one minute, resulting in more than four million data points in total. The anomalies representing the performance issues of the service are labeled by experienced domain engineers. From Table 6.2, we can see that the anomaly ratio is very low. Specifically, we use the first day as the anomaly-free input, whose anomalies (if any) are simply ignored. The next three days are used for offline anomaly detection. Finally, we conduct online anomaly detection on the remaining three days, where we also evaluate the adaptability of different approaches to unseen anomaly patterns. Therefore, for offline anomaly detection, the split ratio for training and testing sets is 0.25:0.75 (1:4), while for online and adaptive anomaly detection, the split ratio is 0.57:0.43 (4:3).

Evaluation Metrics

As anomaly detection is essentially a binary classification problem, i.e., normal and abnormal, we employ *precision*, *recall*,

and *F1 score* for evaluation. They can gauge the performance of an anomaly detection algorithm at a fine-grained level. A satisfactory algorithm should be able to quickly and precisely detect both the occurrence and duration of performance anomalies. Specifically, precision measures the percentage of anomalous metric points that are successfully identified as anomalies over all the metric points that are predicted as anomalous: $precision = \frac{TP}{TP+FP}$. Recall calculates the portion of anomalous metric points that are successfully identified by ADSketch over all the actual anomalous points: $recall = \frac{TP}{TP+FN}$. Finally, the F1 score is the harmonic mean of precision and recall: $F1\ Score = \frac{2 \times precision \times recall}{precision+recall}$. TP is the number of anomalous metric points that are correctly discovered by ADSketch; FP is the number of normal metric points that are wrongly predicted as an anomaly by ADSketch; FN is the number of anomalous metric points that ADSketch fails to notice. Since there are multiple metrics in each dataset, we report their average weighted by the size of each metric time series.

Comparative Methods

We select the following state-of-the-art unsupervised approaches for the comparative evaluation of ADSketch. As all baselines have open-sourced their code, we directly borrow the implementations and follow the procedure of model training and parameter tuning introduced in each method.

- *LSTM* [75, 190]. This method employs Long Short-Term Memory (LSTM) network to capture the normal behaviors of metrics in a forecasting-based manner. Specifically, it predicts the next values of a metric based on its past observations. The predicted values are then compared with the actual values. Anomaly warnings will be raised if the differences exceed the pre-defined thresholds.

- *Donut* [167]. Donut adopts the Variational Autoencoder (VAE) framework to properly reconstruct the normal metric subsequences. The trained model will have a large reconstruction loss when it meets anomalous instances, which serves as the signal to alert anomalies.
- *LSTM-VAE* [127]. Similar to Donut, this work detects anomalies based on metric subsequence reconstruction. It combines LSTM and VAE in the model design.
- *DAGMM* [206]. DAGMM utilizes a deep autoencoder to generate a low-dimensional representation for each input data point, which is further fed into a Gaussian Mixture Model to estimate the anomaly score.
- *SR-CNN* [139]. SR-CNN first applies Spectral Residual to highlight the most important regions for seasonal metric data where anomalies often reside. It then trains a Convolutional Neural Network (CNN) through synthetic anomalies to detect the real anomalies.
- *iForest* [102]. An Isolation Forest (iForest) is composed of a collection of isolation trees, which isolates anomalies based on random subsets of the input features. The height of an input sample, averaged over the trees, is a measure of its normality. Samples with noticeably shorter heights are likely to be anomalies. We use metric subsequences as the input samples.
- *LODA* [132]. LODA is an online anomaly detector based on the ensemble of a series of one-dimensional histograms. Each histogram approximates the probability density of input data projected onto a single projection vector. LODA calculates the likelihood of an anomaly based on the joint probability of the projections.
- *Extreme Value Theory* [150]. Extreme Value Theory (EVT) is an anomaly detection approach in streaming data, which requires no hand-set thresholds and makes no assumption on

Table 6.3: Experimental Results of Offline Anomaly Detection

Method	Yahoo			AIOps18			Industry		
	precision	**recall**	**F1 score**	**precision**	**recall**	**F1 score**	**precision**	**recall**	**F1 score**
LSTM	0.598	**0.706**	0.530	0.499	0.531	0.518	0.704	0.656	0.632
LSTM-VAE	0.622	0.634	0.484	0.510	0.625	0.537	0.717	0.639	0.622
Donut	0.530	0.658	0.524	0.405	0.527	0.382	0.693	0.628	0.604
LODA	**0.754**	0.583	0.428	0.553	0.429	0.401	0.583	0.498	0.529
iForest	0.713	0.597	0.437	0.555	0.439	0.413	0.616	0.567	0.538
DAGMM	0.643	0.517	0.401	0.590	0.477	0.461	0.597	0.542	0.530
SR-CNN	0.433	0.618	0.307	0.424	0.387	0.363	0.519	0.471	0.434
ADSketch	0.511	0.673	**0.541**	**0.744**	**0.670**	**0.677**	**0.811**	**0.813**	**0.740**

the data distribution. It bases on the theorem stating that under a weak condition, extreme events have the same kind of distribution (Extreme Value Distributions), regardless of the original one [46, 55]. Therefore, by inferring the distribution of the extreme values, we can compute them as anomalies.

6.3.2 Experimental Results

In this section, we conduct experiments to answer the research questions.

RQ1 The Effectiveness of ADSketch's Offline Anomaly Detection

To answer this research question, we compare ADSketch with the baselines in the offline setting. The results are shown in Table 6.3, where we can see the average F1 score of ADSketch outperforms all baseline methods in all datasets. In AIOps18 and Industry, the improvement achieved by ADSketch is more significant. In particular, the patterns of anomalies in Yahoo are relatively simple. By iterating over all possible values of the anomaly threshold, the baselines can find the best setting for the dataset under study. Among them, LSTM [75, 190] and Donut [167] achieve comparable performance compared to that of ADSketch (i.e., 0.541), whose average F1 scores are 0.53

and 0.524, respectively. Moreover, LSTM [75, 190] has the best recall (i.e., 0.706), while the best precision (i.e., 0.754) goes to LODA [132]. DAGMM and SR-CNN turn out to be the worst methods in this dataset. In terms of AIOps18 and Industry datasets, we can see ADSketch surpasses the baselines by a larger margin. Specifically, the average F1 score of ADSketch in AIOps18 is 0.677, while that of the second-best method (i.e., LSTM-VAE) is 0.537. ADSketch also attains the best precision and recall. In AIOps18, the anomaly patterns are much more complicated. Baselines tend to predict more data points as anomalous, leading to a lower precision. Different from them, ADSketch is able to precisely capture them and outperforms other methods. The situation is similar in Industry. Particularly, this dataset is collected from online services, and many of its metric curves possess more perceivable and regular patterns. Thus, all methods perform better in this dataset than in the other two. The average F1 scores of ADSketch and the second-best method (i.e., LSTM) are 0.740 and 0.632, respectively.

In Table 6.3, we can see among all comparative methods, LSTM and LSTM-VAE have better overall performance, which are forecasting-based and reconstruction-based methods, respectively. They both try to model the normal patterns of a metric time series and alert anomalies once the metric significantly deviates from the learned patterns. The difference is that a forecasting-based method aims to predict the following metric values and a reconstruction-based method tries to encode and regenerate metric subsequences. We can see, except for LSTM-VAE in Yahoo, these two methods attain the best results compared to other baseline counterparts in the other two datasets. However, LSTM lacks the ability to explicitly detect anomalies in the level of subsequence. Many anomalies are composed of a collection of anomalous points corresponding to the period of performance issues. LSTM-VAE does not take into account the

Table 6.4: Experimental Results of Online Anomaly Detection

Method	AIOps18			Industry		
	prec.	**rec.**	**F1**	**prec.**	**rec.**	**F1**
LSTM	0.425	0.462	0.408	0.612	0.606	0.592
LSTM-VAE	0.336	0.521	0.389	0.624	0.598	0.601
Donut	0.431	0.326	0.376	0.662	0.581	0.590
LODA	0.407	0.397	0.355	0.653	0.526	0.503
iForest	0.397	0.334	0.322	0.576	0.507	0.487
DAGMM	0.392	0.367	0.378	0.557	0.538	0.502
SR-CNN	0.329	0.288	0.307	0.438	0.422	0.410
ADSketch	**0.543**	**0.575**	**0.507**	**0.705**	**0.603**	**0.606**

relationship among subsequences. Many suspicious subsequences are not necessarily anomalies if they often occur in the history of the service systems. Compared to them, ADSketch is able to simultaneously learn the subsequence-level features and consider the context of metric time series.

RQ2 The Effectiveness of ADSketch's Online Anomaly Detection

We also compare ADSketch against the selected methods for online anomaly detection. Table 6.4 presents the experimental results. Except for Donut in AIOps18, all models and algorithms encounter an obvious performance degradation in both datasets. Nevertheless, ADSketch manages to maintain the best ranking (0.507 in AIOps18 and 0.606 in Industry), which is followed by LSTM (0.408 in AIOps18) and LSTM-VAE (0.601 in Industry). Particularly, in AIOps18, the average F1 score of different methods drops by 11%-27%. This observation demonstrates the existence of unprecedented metric patterns in online scenarios. By relying on the "outdated" data and parameters (e.g., ADSketch's metric patterns and baselines' anomaly thresholds) learned from the offline stage, the methods cannot accommodate them. In addition, by plotting the metric time series, we observe

Table 6.5: Experimental Results of Adaptive Pattern Learning

Method	AIOps18			Industry		
	prec.	**rec.**	**F1**	**prec.**	**rec.**	**F1**
LODA	0.424	0.405	0.387	0.623	0.512	0.548
EVT	0.455	0.528	0.406	0.710	0.612	0.458
ADSketch	**0.594**	**0.557**	**0.548**	**0.882**	**0.856**	**0.832**

the emergence of new patterns in metrics. This can be caused by software upgrades or the integration of new service components (e.g., virtual machines, containers). In the industrial dataset, the evaluation results of the baselines are more promising (i.e., the average F1 score drops by less than 10%). This is because the anomalies are triggered by real-world performance issues. The issues have a more natural distribution, and the collected metrics exhibit relatively stable patterns. ADSketch presents a significant performance degradation. We found it is because in some cases, the two metric time series fed to the offline stage are often both anomaly-free. Consequently, no abnormal patterns will be learned, disabling ADSketch to detect anomalies in the online stage. Therefore, when designing an anomaly detection algorithm, adaptability is indispensable.

RQ3 The Effectiveness of ADSketch's Adaptive Pattern Learning

This research question looks into the issue of online adaptability to identify new metric patterns that have not been encountered in the offline stage. We compare ADSketch with two baseline methods with the design of online learning, i.e., LODA and EVT. Similar to RQ2, we only conduct experiments with AIOps18 and Industry datasets. Table 6.5 shows the experimental results, where we can see performance gains in all methods. With more anomalous patterns identified, ADSketch is able to detect anomalies more accurately, i.e., a better precision (0.594 in AIOps18

and 0.882 in Industry). The average F1 score also enjoys some improvements, i.e., 0.548 in AIOps18 and 0.832 in Industry. Particularly, in the industrial case, adaptive ADSketch achieves a performance of over 0.8 in all evaluation metrics (even in some cases without any abnormal patterns learned from the offline stage). Such an achievement indicates its potential to meet the industrial requirements of performance anomaly detection.

On the other hand, the online version of LODA shows little performance improvement (i.e., an average F1 score of 0.387 in AIOps18 and 0.548 in Industry), which even falls behind some methods without the capability of online learning. EVT achieves good performance. By fitting the Extreme Value Distributions with the extreme values, it can capture the distribution of anomalies. However, in many cases, anomalies manifest as a group of consecutive points, especially in the AIOps18 dataset. For example, a long-lasting spike deserves attention even if it does not contain any extreme values. ADSketch considers such cases.

Parameter Sensitivity

In ADSketch, there are only two parameters to tune (both in Algorithm 1), i.e., the pattern length m and the percentile threshold p for identifying deviated metric subsequences. We evaluate the sensitivity of ADSketch to these two parameters by conducting experiments with different settings. Due to space limitations, we only show the results of the Industry dataset. The default value of m and p for the dataset is 15 and 99.5th, respectively. We fix one parameter and employ a different setting for the other one. Specifically, m ranges from 9 to 21, and p varies from 97th to 99.8th. Figure 6.5 presents the results. Performance degradation is observed in both offline and online stages when the two parameters deviate from their default setting. The offline stage exhibits a greater sensitivity, and thus, less anomalous metric patterns are captured. Nevertheless, both the online anomaly detection and

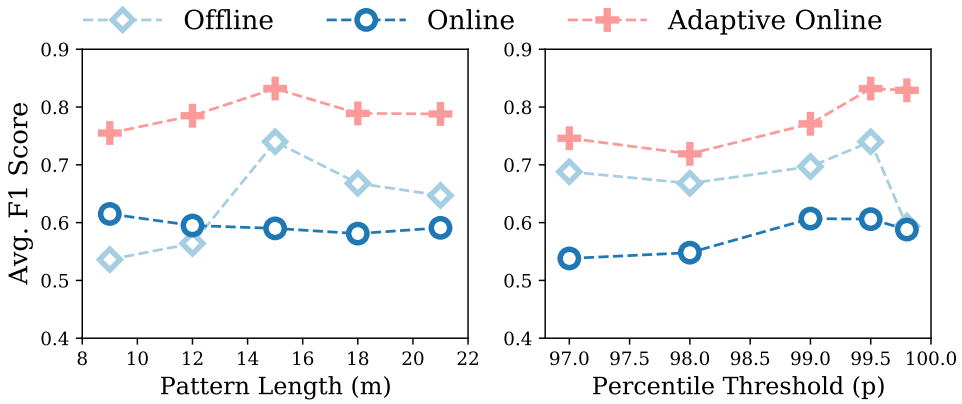


Figure 6.5: Parameter Sensitivity

adaptive pattern learning algorithms achieve stable performance with a smaller set of abnormal patterns. This further confirms ADSketch's capability of new pattern discovery.

6.4 Industrial Practices

6.4.1 Online Deployment

Since October 2020, ADSketch has been successfully incorporated into the performance anomaly detection system of a large-scale online service system in Huawei Cloud. The deployment process can be easily done by leveraging the existing data analytics pipeline, for example, data consumption by Apache Kafka [48], and online parallel execution by Apache Flink [47]. After months of usage, ADSketch has demonstrated its effectiveness in metric-based system troubleshooting. A lot of positive feedback has been received from on-site engineers. Particularly, engineers confirmed its superiority in anomaly detection over the current algorithms (e.g., fixed thresholding, moving average) in operation. One typical case is multiple benign spikes arriving suddenly and consecutively. ADSketch is able to quickly figure out that such recurrent spikes have happened before, which reduces the number of false alerts. In terms of issue understanding, engineers

benefited from ADSketch by having readily-available descriptions of the anomaly symptoms. Therefore, we have initialized a project of metric pattern database construction. ADSketch is continuously accumulating anomalous patterns in the database. Moreover, engineers also expressed the need for metric pattern auto-correlation across different metrics. This is because multiple anomalies collectively could constitute a stronger performance issue indicator. We leave the identification of such correlations to our future work.

6.4.2 Case Study

We provide some case studies of ADSketch collected from production systems in Figure 6.6, where anomalies are indicated by the red lines. Due to space limitations, we only showcase three metric time series. Clearly, all anomalous metric patterns have been successfully located regardless of shape, scale, and length. Each metric time series possesses at least two types of anomalous patterns, e.g., level shifts and spikes. Interestingly, we found the depression in the second metric can help catch a similar pattern in the third metric, demonstrating the feasibility of cross-metric pattern sharing. Moreover, engineers confirmed that these patterns are typical, based on which they can make a good guess about the ongoing issues. For example, the spikes often come from user request surges or network attacks; the depressions in the second and third metrics often indicate service restart or link flap. To quantify the interpretability of ADSketch, we label the recurrent performance issues and employ the learned metric patterns to identify them. As performance issues may contain uncertainty [158], we allow one pattern to be associated with multiple labels simultaneously (Section 22). During the evaluation, an anomaly interpretation is considered correct if the predicted performance issue appears in the label set. In our

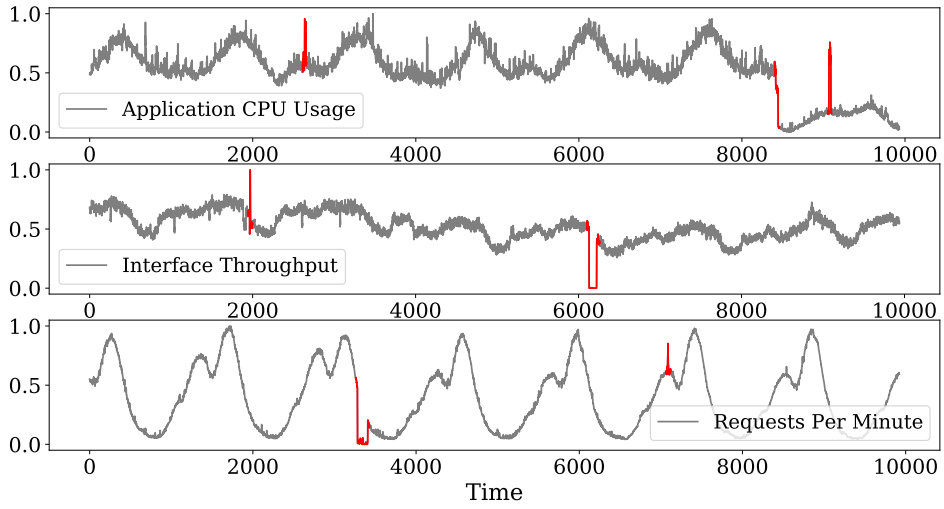


Figure 6.6: Case Study of ADSketch

experiments, ADSketch attains a promising F1 score of 0.825. This demonstrates the potential of ADSketch in providing interpretable results to engineers, which can greatly accelerate the investigation of service performance issues.

6.4.3 Threats to Validity

We have identified the following major threats to validity.

Internal threats. The implementation and parameter selection are two critical internal threats to the validity. To reduce the implementation threat, we directly borrow the codes released by the baseline approaches. For the proposed approach, we employ peer code review, i.e., the authors are invited to carefully check the implementation for mistakes. In terms of parameter selection, we conduct multiple comparative experiments with different parameters for all methods. We choose the parameter settings empirically based on the best results.

External threats. The selection of the service system and the baselines are two main external threats to validity. We choose a large-scale online service of Huawei Cloud, which produces millions of metrics with diverse patterns. Moreover, we detect

anomalies by following the basic definition of an anomaly, i.e., the data point that deviates from the majority in a dataset. Thus, ADSketch is generalizable to other systems. For baselines, we select the representative ones in the literature, covering a wide spectrum of techniques.

Construct threats. The main construct threat to validity is that the anomaly-free input (i.e., $\mathcal{T}_n$) to Algorithm 1 actually contains anomalies. Although anomaly-free data are easily obtainable in reality, false negatives could happen if the data are contaminated. We alleviate this issue by applying percentile thresholding to $\mathcal{T}_n$. Specifically, after obtaining the closest subsequence pairs in $\mathcal{T}_n$, we break the connection between those having a distance above the percentile threshold. Thus, the set of anomaly candidates, i.e., N_i, becomes larger. If $\mathcal{T}_n$ is indeed clean, this operation is harmless as the (isolated) normal metric subsequences can be grouped with other similar ones again; if not, they will stay isolated and eventually be recognized as anomalies. We have also conducted experiments on some cases where $\mathcal{T}_n$ contains anomalies, and the results show its effectiveness.

6.5 Summary

In this chapter, we introduce ADSketch, a performance anomaly detection approach based on pattern sketching. By extracting normal and abnormal patterns from metric time series, anomalies can be quickly detected through a comparison with the identified patterns. By associating metric patterns with typical performance issues, ADSketch can provide interpretable results when any known patterns appear again. Moreover, we design an adaptive learning algorithm to help ADSketch embrace unprecedented metric patterns during online anomaly detection. We have conducted experiments on two public datasets and one production dataset collected from a representative online service

system of Huawei Cloud. For offline anomaly detection where models' parameters are still being tuned, ADSketch has achieved the highest F1 score, outperforming the existing methods by a significant margin. For online anomaly detection where models are fixed, ADSketch safeguards its best rankings. Finally, the adaptive pattern learning brings noticeable performance gains, especially in the industrial dataset. From our industrial practice, we have witnessed it shedding light on accurate and interpretable performance anomaly detection, which confirms its practical benefits conveyed to Huawei Cloud. We believe ADSketch is able to assist engineers in service failure understanding and diagnosis.

For future work, we will extend our algorithms to multivariate metric time series. We will also try to provide more detailed information about failures by exploring the correlations among the metric patterns.

□ **End of chapter.**

Chapter 7

Graph-based Alert Aggregation

Alerts capture service and system anomalies that deserve engineers' immediate attention. A failure could result in a large number of alerts across the entire cloud system. In this chapter, we introduce GIRDLE, an alert aggregation framework based on graph representation learning. By gathering related alerts, GIRDLE helps engineers understand the failure, estimate its impact scope, and save duplicate engineering efforts as mentioned in Chapter 4. The remainder of this chapter is organized as follows. Section 7.1 introduces the problem background and our contributions. Section 7.2 describes the proposed framework. Section 7.3 shows the experiments and experimental results. Section 7.4 presents our success story and lessons learned from practice. Finally, Section 7.5 summarizes this chapter.

7.1 Problem and Contributions

When a failure happens, system monitors will render a large number of alerts to capture different failure symptoms [29, 25, 191], which can help engineers quickly obtain a big picture of the failure and pinpoint the root cause. For example, "Special instance cannot be migrated" is a critical network failure in Virtual Private Cloud (VPC) service. Alert "Tunnel bearing

network pack loss" is a signal for this network failure, which is caused by the breakdown of a physical network card on the tunnel path. Due to the large scale and complexity of online service systems, the number of alerts is overwhelming the existing alert management systems [28, 29, 191]. When a service failure occurs, aggregating related alerts can greatly reduce the number of alerts that need to be investigated. For example, linking alerts caused by a hardware issue can provide engineers with a clear picture of the failure, e.g., the type of the hardware error or the specific malfunctioning components. Without automated alert aggregation, engineers may need to go through each alert to discover the existence of such a problem and collect all related alerts to understand it. Moreover, alert aggregation can also facilitate failure diagnosis. In cloud systems, some trivial alerts are being generated continuously, and multiple (independent) failures can happen simultaneously. Identifying correlated alerts can therefore accelerate the process of root cause localization.

To aggregate related alerts, one straightforward way is to measure their textual similarity [191, 25]. For example, alerts that share similar titles are likely to be related. Besides textual similarity, system topology (e.g., service dependency, network IP routing) is also an important feature to resort to. Due to the dependencies among online services, failures often have a cascading effect on other interdependent services. A service dependency graph can help track related alerts caused by such an effect. However, as cloud systems often possess a certain fault-tolerant capability, some services may not report alerts, impeding the tracking of failures' impact (as introduced in Section 2.6). This issue is ubiquitous in production systems, which has not yet been properly addressed in existing work. Moreover, the patterns of alerts are collectively influenced by different factors, such as their topological and temporal locality. Existing work [191, 25] combine them by a simple weighted sum, which may not be able

to reveal the latent correlations among alerts.

In this chapter, we present GIRDLE (which stands for Graph representatIon leaRning-baseD aLert aggrEgation), which is an alert aggregation framework to assist engineers in failure understanding and diagnosis. Different from the existing work of alert storm handling [191] and linked alert identification [25], we do not rely on alerts' textual similarity. Moreover, we learn alerts' topological and temporal correlations in a unified manner (instead of by a weighted combination). Traditional applications of graph representation learning often learn the semantics of a fixed graph. Unlike them, we propose to learn a feature representation for each unique type of alert, which can appear in multiple places on the graph. The representation encodes the historical co-occurrence of alerts and their topological structure. Thus, they can be naturally used for alert aggregation in online scenarios. To track the impact graph of a failure (i.e., the alerts triggered by the failure), we exploit more fine-grained system signals, i.e., metrics, as auxiliary information to discover the scope of its cascading effect. Metrics profile the impact of failures in a more sophisticated way. Therefore, if two services exhibit similar abnormal behaviors (characterized by alerts and metrics), they should be suffering from the same problems even if no alerts have been reported. Finally, we apply community detection algorithms to find the scope of different failures.

This work aims to assist engineers in failure understanding and diagnosis with online alert aggregation, which is to aggregate alerts caused by the same failure. When services encounter failures, alerts that capture different failure symptoms constitute an essential source for engineers to conduct a diagnosis. However, it is time-consuming and tedious for engineers to manually examine each alert for failure investigation when faced with such an overwhelming number of alerts. Online alert aggregation is to cluster relevant alerts when they come in a streaming man-

Table 7.1: Examples of Alert Aggregation

No.		Alert Title	Time	Pod	Severity
1	Alert group 1	Virtual machine is in abnormal state	2020/10/09 19:40	pod01	Low
2		Virtual network interface receive lost ratio over 20%	2020/10/09 19:40	pod02	High
3		Traffic burst seen in Nginx node	2020/10/09 19:40	pod02	Low
4		Traffic burst seen in LVS (Linux Virtual Server) node	2020/10/09 19:41	pod09	Medium
5		OSPF (Open Shortest Path First) protocol state change	2020/10/09 19:41	pod04	Medium
6	Alert group 2	Excessive I/O delay of storage disk	2020/10/12 14:34	pod09	Medium
7		Component failure	2020/10/12 14:34	pod05	High
8		Hard disk failure	2020/10/12 14:34	pod09	Medium
9		Database account login error	2020/10/12 14:34	pod18	Medium
10		Monitor detected customer-impacting alerts for Storage in [AZ1]	2020/10/12 14:35	pod10	Medium

ner (i.e., continuously reported by the system). Examples are presented in Table 7.1, where items in blue and gray belong to two groups of aggregated alerts. The first group shows a virtual network failure. Note that only the No.3 and No.4 alerts share some words in common, while others do not. Meanwhile, the second group describes a hardware failure and, more specifically, a storage disk error. Engineers can benefit from such alert aggregation as the problem scope is narrowed down to each alert group. However, accurately aggregating alerts for online service systems is challenging. We have identified three main reasons:

- **Background noise**. Although related alerts are indeed generated around the same time, many other cloud components are also constantly rendering alerts. These alerts are primarily trivial issues and therefore become background noise. Alert aggregation based on temporal similarity would suffer from a high rate of false positives.
- **Dissimilar textual description**. Text (e.g., alert title and summary) similarity is an essential metric for alert correlation, which has been widely used in existing work [191, 25]. However, in reality, related alerts, especially critical ones, do not necessarily have similar titles. Failing to correlate

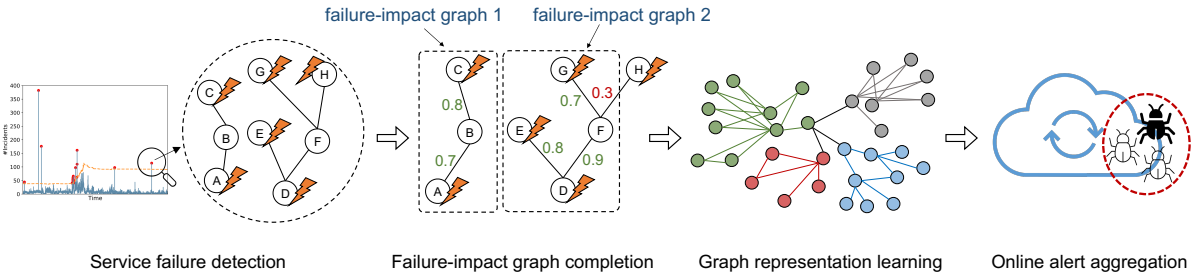


Figure 7.1: The Overall Framework of GIRDLE

such critical alerts greatly hinders root cause diagnosis.

- **Unclear failure-impact graph**. To correlate alerts accurately, we need to estimate the impact graph of service failures. As discussed in Section 2.6.2, this task is challenging. alerts alone are insufficient to completely reflect the impact of failures on the entire system. Therefore, we need to utilize more fine-grained information about the failures.

7.2 Methodology

7.2.1 Overview

In cloud systems, a large number of monitors are configured to continuously monitor the state of its services from different aspects. Many alerts rendered by the monitors tend to co-occur due to their underlying dependencies. For example, some failure symptoms often appear together, and some alerts may develop a causal relationship. Our main idea is to capture the co-occurrences among alerts by learning from historical failures. In online scenarios, such correlations can be leveraged to distinguish correlated alerts that are generated in streams.

The overall framework of GIRDLE is illustrated in Figure 7.1, which consists of four phases, i.e., *service failure detection*, *failure-impact graph completion*, *graph representation learning*, and *online alert aggregation*. The first phase tries to identify the

occurrence of service failures and retrieves different types of monitoring data, including alerts, metric time series, and service system topology. In the second phase, we try to identify the alerts that are triggered by each individual failure detected above. More often than not, it is hard to precisely identify the impact scope of failures (as discussed in Section 2.6.2), which hinders the learning of alerts' correlations. Therefore, we utilize the trends observed in metric curves to auto-complete the failure-impact graphs. After obtaining the set of alerts associated with each failure, in the third phase, an embedding vector is learned for different types of alerts by leveraging existing graph representation learning models [60, 198]. Such representation encodes not only the temporal locality of alerts, but also their topological relationship. In the final phase, the learned alert representation will be employed for online alert aggregation by considering their cosine similarity and topological distance. In particular, we do not explicitly consider the dynamic change of a system topology because the changes often happen to a small area of the topology, e.g., container creation or kill. GIRDLE essentially learns the correlations among alerts, which are also applicable to the changed portion of the topology. Nevertheless, when the system topology goes through a significant alteration, our framework is efficient enough to support quick model retraining.

7.2.2 Service Failure Detection

Due to the cascading effect, when service failures occur, a large number of alerts are often reported in a short period of time. Thus, setting a fixed threshold for the average number of reported alerts (e.g., #alerts/min>50) could be a reasonable criterion to detect failures. However, such a design suffers from a trade-off between false positives and false negatives due to online service systems' complex and ever-changing nature [191]. For example,

different services have distinct sensitivity to the number of alerts, and continuous system evolution/feature upgrades could change the threshold. Thus, a self-adaptive algorithm is more desirable.

For time-series data, anomalies often manifest themselves as having a large magnitude of upward/downward changes. Extreme Value Theory (EVT) [150] is a popular statistical tool to identify data points with extreme deviations from the median of a probability distribution. It has been applied to predicting unusual events, e.g., severe floods and tornado outbreaks [38], by finding the law of extreme values that usually reside at the tail of a distribution. Moreover, it requires no hand-set thresholds and makes no assumptions about data distribution. In this work, we follow [150, 191] to detect bursts in time series of the number of alerts per minute. As a typical time series anomaly detection problem, other approaches (e.g., [99, 75]) in this field are also applicable. The bursts are regarded as the occurrence of service failures. This algorithm can automatically learn the normality of the data in a dynamic environment and adapt the detection method accordingly. Figure 7.1 (phase one) presents an example of service failure detection, where all abnormal spikes are successfully found by the decision boundary (the orange dashed line). For consecutive bins that are marked as anomalies, we regard them as one failure because failures may last for more than one minute. The next phase will distinguish multiple (independent) failures that happen simultaneously. Particularly, the detection algorithm is only required to have a high recall, and the precision is of less importance. It is because the goal of the follow-up two phases is to find the correlations between alerts. Such correlation rules will not be violated even if alerts are not appearing together during actual cloud failures.

7.2.3 Failure-Impact Graph Identification

In the first phase, the number of alerts per minute is calculated, and alert bursts are regarded as the occurrence of service failures. For each failure, the alerts collected from the entire system are not necessarily related to it. This is because: 1) while some services are suffering from the failure, others may continuously report alerts (could be trivial and unrelated issues); and 2) multiple service failures could happen simultaneously. Therefore, we need to identify the set of alerts for each individual failure that is generated due to the cascading effect.

To this end, the concept of community detection is exploited. Community detection algorithms aim to group the vertices of a graph into distinct sets, or communities, such that there exist dense connections within a community and sparse connections between communities. Each community represents a collection of alerts rendered by the common service failure, in which the correlations among alerts can be explored. A comparative review of different community detection algorithms is available in [176]. In this work, we employ the well-known *Louvain* algorithm [10], which is based on modularity maximization. The modularity of a graph partition measures the density of links inside communities compared to links between communities. For weighted graphs, the modularity can be calculated as follows [10]:

$$M = \frac{1}{2m} \sum_{i,j} [W_{i,j} - \frac{k_i k_j}{2m}] \delta(c_i, c_j) \tag{7.1}$$

where W_{ij} is the weight of the link between node i and j, $k_i = \sum_j W_{ij}$ sums the weights of the links associated with node i, c_i is the community to which node i is assigned to, $m = \frac{1}{2} \sum_{ij} W_{ij}$, and the $\delta(u, v) = 1$ if $u = v$ and 0 otherwise.

To better understand the identification of failure-impact graphs using community detection, an illustrating example is depicted

in Figure 7.1 (phase two). In this case, except for nodes B and F, other nodes all report alerts. By conducting community detection, we obtain two communities: $\{A, B, C\}$ and $\{C, E, F, G\}$, which are regarded as the complete impact graph of their respective failure. The weight between nodes is provided with their link. We can see that intra-community links all have a relatively large weight. Such a partition can achieve the best modularity score for this example. Particularly, node H is excluded from the second community due to the small weight of its connection to node F.

To apply community detection, the weight between two nodes should be defined. Inspired by [104], we combine metrics with alerts to calculate the behavioral similarity between two nodes and use the similarity value as the weight. Specifically, the weight is composed of two parts, i.e., alert similarity and metric trend similarity.

Alert similarity

Th alert similarity is to compare the alerts reported by two nodes. Typically, if two nodes encounter similar errors, they will render similar types of alerts. Jaccard index is employed to quantify such similarity, which is defined as the size of the intersection divided by the size of the union of two alert sets:

$$Jaccard(i, j) = \frac{|inc(i) \cap inc(j)|}{|inc(i) \cup inc(j)|} \tag{7.2}$$

where $inc(i)$ is the alerts reported by node i. In particular, we allow duplicate types of alerts in each set by assigning them a unique number. This is because the distribution of alert types also characterizes the failure symptoms.

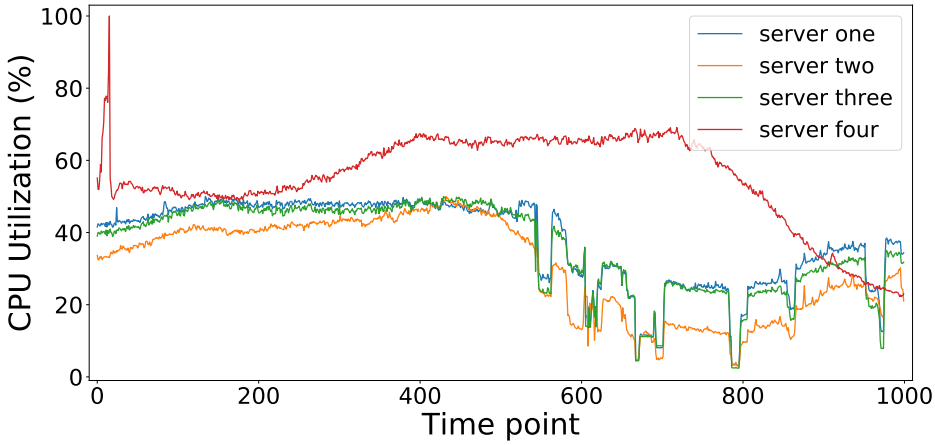


Figure 7.2: CPU Usage Curve of Four Servers

Metric trend similarity

As discussed in Section 2.6, some services may remain silent when failures happen, hindering the tracking of related alerts. To bridge this gap, we resort to metrics, which are more sophisticated monitoring signals. Intuitively, the metric trend similarity measures the underlying consistency of cloud components' abnormal behaviors, which cannot be captured by alerts alone. An example is shown in Figure 7.2, which records the CPU utilization of four servers. Clearly, the curve of the first three servers exhibits a highly similar trend, while such a trend cannot be observed in server four. The implication is that the first three servers are likely to be suffering from the same issue and thus should belong to the same community. We adopt dynamic time warping (DTW) [81] to measure the similarity between two temporal sequences with varying speeds. We observe the issue of temporal drift between two time series, which is common as different cloud components may not be affected by a failure simultaneously during its propagation. Also, the clock of two nodes may not always synchronized, so their metric may not aligned. Therefore, DTW fits our scenario.

The remaining problem is which metrics should be utilized for

similarity evaluation. Normal metrics which record the system's normal status should be excluded as they provide trivial and noisy information. Therefore, EVT introduced in phase one is utilized again to detect anomalies for each metric. Only the abnormal metrics shared by two connected cloud components will be compared. In particular, we can incorporate ADSketch proposed in Chapter 6 to make the metric similarity more interpertable. Specifically, with the help of ADSketch, we are able to figure out what anomalies two services are experiencing to determine whether they should be correlated. However, this requires building a knowledge pool of the correlations between anomalies. As metric similarity can serve as a good indicator of whether two services are suffering from the same problem, the exploration of interpretability is left for future work. Finally, when there exists more than one type of abnormal metric, we use the average similarity score calculated as follows:

$$DTW(i,j) = \frac{1}{K}\sum_{k=1}^{K} dtw(t_k^i, t_k^j) \tag{7.3}$$

where K is the number of metrics to compare for node i and j, t_k^i is the k^{th} metric of node i, and $dtw(u,v)$ measures the DTW similarity between two metric time series u and v, which is normalized for path length. The weight W_{ij} between node i and j is computed by taking the weighted sum of the two types of similarities as follows:

$$W_{ij} = \alpha \times Jaccard(i,j) + (1-\alpha) \times DTW(i,j) \tag{7.4}$$

where the balance weight α is a hyper-parameter. In our experiments, if two nodes both report alerts, we set it as 0.5; otherwise, it is set to be 0, i.e., only the metric trend similarity is considered.

Finally, for each discovered community, the alerts inside it form the complete impact graph of the service failure. Note

that in online scenarios, we cannot directly adopt the techniques introduced in this phase for alert aggregation. This is because they involve a comparison between different metrics, which are not complete until the failures fully manifest themselves. Thus, the comparison is often delayed and inefficient. Moreover, they can be error-prone without fully considering the historical cases.

7.2.4 Graph-based Alert Representation Learning

After obtaining the impact graph for each service failure (i.e., the actual alerts triggered by it), we can learn the correlations among alerts. Such correlations describe the sets of alerts that tend to appear together. FP-Growth proposed by Han et al. [61] is a standard algorithm to mine such frequent item sets. However, our analysis reveals the following drawbacks it possesses for our problem:

- It is vulnerable to background noise. In production environments, some simple alerts are constantly being reported, e.g., "High CPU utilization rate". These alerts will appear in many transactions (a collection of items that appear together) for FP-Growth. As a result, unrelated alerts might be put into the same frequent item set due to sharing such alerts. These simple alerts cannot be trivially removed as they provide necessary information about a system, and a burst of such alerts can also indicate serious problems.
- It cannot handle alerts with a low frequency. FP-Growth has a parameter called *support*, which describes how frequently an item set is in the dataset. Alert sets with a low support value will be excluded to guarantee the statistical significance of the results. However, more often than not, such alert sets are more important, as they report some critical failures that do not happen frequently.

In online service systems, different resources (e.g., microser-

vices and devices) are naturally structured in graphical forms, such as service dependency and network IP routing. Therefore, graph representation learning [60] can be an ideal solution to deal with the above issues. Graph representation learning is an essential and ubiquitous task with applications ranging from drug design to friendship recommendation in social networks. It aims to find a representation for graph structure that preserves the semantics of the graph. A typical graph representation learning algorithm learns an embedding vector for all nodes of a graph. For example, Chen et al. [25] employed *node2vec* [56] to learn a feature representation for cloud components. Different from them, we propose to learn a representation for each unique type of alert, which can appear in multiple places on the graph. In our framework, we employ *DeepWalk* [130] because of its simplicity and superior performance. DeepWalk belongs to the class of shallow embedding approaches that learn the node embeddings based on random walk statistics. The basic idea is to learn an embedding ϑ_i for node v_i in graph $\mathcal{G}$ such that:

$$EMB(\vartheta_i, \vartheta_j) \triangleq \frac{e^{\vartheta_i \cdot \vartheta_j}}{\sum_{v_k \in \mathcal{V}} e^{\vartheta_i \cdot \vartheta_k}} \approx p_{\mathcal{G},T}(v_j|v_i) \tag{7.5}$$

where $\mathcal{V}$ is the set of nodes in the graph and $p_{\mathcal{G},T}(v_j|v_i)$ is the probability of visiting v_j within T hops of distance starting at v_i. The loss function to maximize such probability is:

$$\mathcal{L} = \sum_{(v_i,v_j)\in\mathcal{D}} -log(EMB(\vartheta_i, \vartheta_j)) \tag{7.6}$$

where $\mathcal{D}$ is the training data generated by sampling random walks starting from each node. Readers are referred to the original paper [130] for more details.

For each failure-impact graph, alert sequences are generated through random walks starting from every node inside. In reality, each node usually generates more than one alert when failures

happen. Our tailored random walk strategy therefore contains two hierarchical steps. In the first step, a node is chosen by performing random walks on the node level; in the second step, an alert will be randomly selected from those reported by the chosen node. Duplicate types of alerts in a node will be kept because the frequency is also an essential feature of alerts (it impacts the probability of being selected). Following the original setting of [56], we set the walk length as 40, i.e., each alert sequence will contain 40 samples. Finally, the alert sequences will be fed into a Word2Vec model [118] for embedding vector learning. The Word2Vec model has two crucial hyper-parameters: the window size and the dimension of the embedding vector. We set the window size as ten by following [56] and the dimension as 128. In particular, by considering the topological distance between alerts, we can alleviate the problem of background noise. This is because as the distance increases, the impact of noisy alerts gradually weakens, while in FP-Growth, all alerts play an equivalent role in a transaction.

7.2.5 Online Alert Aggregation

With the learned alert representation from the last phase, we can conduct alert aggregation in production environments, where the alerts come in a streaming manner. Each group of aggregated alerts represents a specific type of service issue, such as a hardware issue, network traffic issue, network interface down, etc. The EVT-based method also plays a role in this phase by continuously monitoring the number of alerts per minute. If it alerts a failure, the online alert aggregation will be triggered. When two alerts, say i and j, appear consecutively, GIRDLE measures their similarity. If the similarity score is greater than a predefined threshold, they will be grouped together immediately. In particular, the similarity score consists of two parts, i.e., *his-*

torical closeness (HC) and *topological rescaling* (TR), which are defined as follows:

$$\begin{aligned} HC(i,j) &= \frac{\vartheta_i \cdot \vartheta_j}{\|\vartheta_i\| \times \|\vartheta_j\|} \\ TR(i,j) &= \frac{1}{max(1, d(i,j) - \mathcal{T})} \end{aligned} \tag{7.7}$$

where ϑ_i and ϑ_j are the embedding vectors of alert i and j (as described in Section 7.2.4), respectively; $d(i,j)$ is the topological distance between i and j, which is the number of hops along their shortest path in the system topology; and $\mathcal{T}$ is the threshold for considering the penalty of long distance. That is, the topological rescaling becomes effective (i.e., <1) only if their distance is larger than $\mathcal{T}$. In our experiments, $\mathcal{T}$ is set as four. Incorrect correlations will be learned if $\mathcal{T}$ is too large, while important correlations will be missed if $\mathcal{T}$ is too small. Our experiments show similar results when $\mathcal{T}$ is in $[3, 6]$. Cosine similarity is adopted to calculate the historical closeness, which is related to their co-occurrences in the past. Finally, the similarity between i and j can be obtained by taking the product of $TR(i,j)$ and $HC(i,j)$:

$$\begin{aligned} sim(i,j) &= TR(i,j) \times HC(i,j) \\ &= \frac{1}{max(1, d(i,j) - \mathcal{T})} \times \frac{\vartheta_i \cdot \vartheta_j}{\|\vartheta_i\| \times \|\vartheta_j\|} \end{aligned} \tag{7.8}$$

We set an aggregation threshold λ for $sim(i,j)$ to consider whether or not two alerts are correlated:

$$cor(i,j) = \begin{cases} 1, & if\ sim(i,j) \geq \lambda \\ 0, & otherwise \end{cases} \tag{7.9}$$

In our experiments, λ is empirically set as 0.7. In particular, the

distance of an alert to a group of alerts is defined as the largest value obtained through element-wise comparison.

7.3 Experiments

In this section, we evaluate our framework using real-world alerts collected from industry. Particularly, we aim to answer the following research questions.

RQ1: How effective is the service failure detection module of GIRDLE?

RQ2: How effective is GIRDLE in alert aggregation?

RQ3: Can the failure-impact graph help alert aggregation?

7.3.1 Experiment Setting

Dataset

Alert aggregation is a typical problem across different online service systems. In this experiment, we select a representative, large-scale system, i.e., the Networking service of Huawei Cloud, to evaluate the proposed framework. Besides offering traditional services such as Virtual Network, VPN Gateway, it also features intelligent IP networks and other next-generation network solutions. In particular, the service system comprises a large and complex topological structure. In the layer of infrastructure, platform, and software, it has multiple instances of virtual machines, containers, and applications, respectively. In each layer, their dependencies form a topology graph. The cross-layer topology is mainly constructed by their placement relationships, i.e., the mappings between applications, containers, and virtual machines. Like other cloud enterprises, Huawei Cloud's resources are hosted in multiple regions and endpoints worldwide. Each region is composed of several availability zones (isolated locations within regions from which public online services originate and

operate) for service reliability assurance. The alert management of the Networking service is also conducted in a multi-region way, with each region having relatively isolated issues. We collect alerts generated between May 2020 and November 2020, during which the Networking service reported a large number of alerts. Although we conduct the evaluation of a single online service system, we believe GIRDLE can be easily applied to other online service systems and bring them benefits.

To evaluate the effectiveness of GIRDLE, experienced domain engineers manually labeled related alerts. Thanks to the well-designed alert management system with user-friendly interfaces, the engineers can quickly perform the labeling. Note that the manual labels are only required for evaluating the effectiveness of our framework, which is unsupervised. To calculate the metric trend similarity, we adopt the following metrics, which are suggested by the engineers:

- *CPU utilization* refers to the amount of processing resources used.
- *Round-trip delay* records the amount of time it takes to send a data packet plus the time it takes to receive an acknowledgment of that data packet.
- *Port in-bound/out-bound traffic rate* refers to the average amount of data coming-in to/going-out of a port.
- *In-bound packet error rate* calculates the error rate of the packet that a network interface receives.
- *Out-bound packet loss rate* calculates the loss rate of the packet that a network interface sends.

These metrics are representative that characterize the basic states of the Networking service system. In particular, CPU utilization is monitored for different containers and virtual machines, while the remaining metrics are monitored for the virtual interfaces of each network device. Each metric is calculated or

Table 7.2: Dataset Statistics

Dataset	**Training period**	**Testing period**	**#alerts**	**#failures**
Dataset1	2020 May - July	2020 Aug.	∼18k/∼8k	105/46
Dataset2	2020 May - Aug.	2020 Sept.	∼26k/∼10k	151/52
Dataset3	2020 May - Sept.	2020 Oct.	∼36k/∼8k	203/38

sampled every minute. We collect two hours of data to measure the metric trend similarity. Note that the set of metrics can be tailored for different systems. For example, a database service may also care about the number of failed database connection attempts, the number of SQL queries, etc.

We select the largest ten availability zones for experiments, each of which contains a large system topology. Six months of production alerts are collected from the Networking service of Huawei Cloud. The number of distinct alert types is more than 3,000. Similar to [191, 69, 99], we conduct three groups of experiments using alerts reported in the first four months, the first five months, and all months, respectively. In all periods, alert aggregation is applied to the failures that happened in the last month based on the alert representations learned from previous months. Table 7.2 summarizes the dataset. For column *#alerts* (resp. *#failures*), the first figure calculates the alerts (resp. failures) captured during the training period, while the second figure shows that of the testing period. Particularly, some failures are of small scale and can be quickly mitigated, while others are cross-region and become an expensive drain on the company's revenue. We can see each failure is associated with roughly 200 alerts, demonstrating a strong need for alert aggregation.

Evaluation Metrics

For RQ1, which is a binary classification problem, we employ *precision*, *recall*, and *F1 score* for evaluation. Specifically, precision measures the percentage of alert bursts that are successfully identified as service failures over all the alert bursts that are predicted as failures: $precision = \frac{TP}{TP+FP}$. Recall calculates the portion of service failures that are successfully identified by GIRDLE over all the actual service failures: $recall = \frac{TP}{TP+FN}$. Finally, F1 score is the harmonic mean of precision and recall: $F1\ score = \frac{2 \times precision \times recall}{precision+recall}$. TP is the number of service failures that are correctly discovered by GIRDLE; FP is the number of trivial alert bursts (i.e., no failure is actually happening) that are wrongly predicted as service failures by GIRDLE; FN is the number of service failures that GIRDLE fails to discover.

For RQ2 and RQ3, we choose Normalized Mutual Information (NMI) [152], which is a widely used metric for evaluating the quality of clustering algorithms. The value of NMI ranges from 0 to 1 with 0 indicating the worst result (no mutual information) and 1 the best (perfect correlation): $NMI(\Omega, \mathbb{C}) = \frac{2 \times I(\Omega;\mathbb{C})}{H(\Omega)+H(\mathbb{C})}$, where Ω is the set of clusters, $\mathbb{C}$ is the set of classes, $H(\cdot)$ is the entropy, and $I(\Omega;\mathbb{C})$ calculates and mutual information between Ω and $\mathbb{C}$.

Implementation

Our framework is implemented in Python. We parallelize our experiments by assigning availability zones to different processors. The output of each processor is a list of alert sequences generated through a random walk, which we merge and feed to a Word2Vec model implemented with Gensim [207], an open-source library for topic modeling and natural language processing. We run our experiments on a machine with 20 Intel(R) Xeon(R) Gold 6148 CPU @ 2.60GHz, and 256GB of RAM. The results show

that each phase of our framework takes only a few seconds. The last phase can even produce results in a real-time manner as it only involves simple vector calculation. Thus, our framework can quickly respond in online scenarios. This demonstrates that GIRDLE is of high efficiency.

7.3.2 Comparative Methods

The following methods are selected for comparative evaluation.

- *FP-Growth* [61]. FP-Growth is a widely-used algorithm for association pattern mining. It is utilized as an analytical process that finds a set of items that frequently co-occur in datasets. In our experiments, each impact graph is regarded as a transaction for this algorithm. Given a set of impact graphs, it searches alerts that often appear together, regardless of their distance.
- *UHAS* [191]. This approach is proposed by Zhao et al. aiming at handling alert storms for online service systems. Similar to alert bursts, alert storms also serve as a signal for service failures. Particularly, UHAS employs DBSCAN for alert clustering based on their textual and topological similarity. The textual similarity between the two alerts is measured by Jaccard distance. The topological similarity considers two types of topologies, i.e., software topology (service) and hardware topology (server). The topological distance is computed by the shortest path length between two nodes. Finally, a weighted combination of the two types of similarities yields the final similarity score.
- *LiDAR [25].* LiDAR is a supervised method proposed by Chen et al. to identify linked alerts in large-scale online service systems. Specifically, LiDAR is composed of two modules, i.e., the textual encoding module and the component embedding module. The first module produces similar rep-

resentations for the text description of linked alerts, which are labeled by engineers. In the evaluation stage, the textual similarity between two alerts is measured by the cosine distance of their representations. The second module learns a representation of the system topology (instead of alerts). The final similarity is calculated by taking a weighted sum of both parts. As LiDAR is supervised, it would be unfair to compare it with other unsupervised methods. Considering the success of the Word2Vec model [118, 117] in identifying semantically similar words (in an unsupervised manner), we alter LiDAR to be unsupervised to fit our scenario by representing the text of alerts with off-the-shelf word vectors [80].

7.3.3 Experimental Results

RQ1: The Effectiveness of Girdle's Service Failure Detection

To answer this research question, we compare GIRDLE with the fixed thresholding method on three datasets and report precision, recall, and F1 score. Thresholding remains an effective way for anomaly detection in production systems and serves as a baseline in much existing work. Since both methods require no parameter training, we use them to detect failures for both the training data and evaluation data. Particularly, the threshold of the baseline method is #alerts/min>50, which is recommended by field engineers. Moreover, the ground truth is obtained directly from the historical failure tickets, which are stored in the alert management system.

The results are shown in Table 7.3, where GIRDLE outperforms the fixed thresholding in all datasets and metrics. In particular, GIRDLE achieves an F1 score of more than 0.93 in different datasets, demonstrating its effectiveness in service failure detection. Indeed, we observe that some failures may not always

Table 7.3: Experimental Results of Service Failure Detection

Dataset	Metric	Thresholding	Girdle
Dataset1	Precision	0.711	**0.917**
	Recall	0.913	**0.957**
	F1 Score	0.799	**0.937**
Dataset2	Precision	0.831	**0.944**
	Recall	0.942	**0.981**
	F1 Score	0.883	**0.962**
Dataset3	Precision	0.648	**0.925**
	Recall	0.921	**0.974**
	F1 Score	0.761	**0.949**

Table 7.4: Experimental Results of Alert Aggregation

Method	Dataset1	Dataset2	Dataset3
FP-Growth	0.481	0.523	0.546
UHAS	0.697	0.71	0.707
LiDAR	0.742	0.758	0.826
Girdle	**0.831**	**0.866**	**0.912**

incur a large number of alerts at the beginning. However, if ignored, they could become worse and end up yielding more severe impacts across multiple services. Fixed thresholding does not possess the merit of threshold adaptation based on the context and thus produces many false positives. Girdle outperforms it for being able to adjust the threshold automatically.

RQ2: The Effectiveness of Girdle in Alert Aggregation

We compare the performance of Girdle against a series of baseline methods for alert aggregation. Table 7.4 shows the NMI values of different experiments. From dataset 1 to 3, Girdle achieves an NMI score of 0.831, 0.866, and 0.912, respectively,

while the best results from the baseline methods are 0.742, 0.758, and 0.826, all attained by LiDAR. LiDAR outperforms UHAS by explicitly considering the entire system topology. Except for UHAS, all approaches achieve better performance with more training data available. This is because UHAS directly works on alert storms when failures are detected. Without learning from history, it cannot handle complicated scenarios. Recall that both UHAS and LiDAR rely on the textual similarity between alerts. However, in our system, related alerts do not necessarily possess similar text descriptions. For example, there is a clear correlation between the alert "Traffic drops sharply in vRouter" and "OS network ping abnormal" in VPC service, which tends to be missed by them. Moreover, monitors that render alerts are configured by multiple service teams, which further damages the credit of textual similarity. This is particularly true for some critical alerts because they are often tailored for special system errors, which may not be shared across different services. On the other hand, although GIRDLE does not explicitly leverage the alert's textual features, our experiments show that it is capable of correlating alerts that share some common words, e.g., "VPC service tomcat port does not exist" and "VPC service tomcat status is dead." This is because such a relationship is reflected in their temporal and topological locality, which can be precisely captured by alerts' representation vectors.

Another observation is that FP-Growth does not fit the task of alert aggregation, whose best NMI score is 0.546. As discussed in Section 7.2.4, this method is not robust against background noise. Indeed, in the system, some trivial alerts (e.g., "Virtual machine is in abnormal state") are continuously being reported, which may connect alerts from distinct groups. Furthermore, many essential alerts are excluded by this method due to low frequency, which is undesirable. This problem can be effectively alleviated by leveraging the topological relationship between alerts as done

by other approaches. According to Equation 7.5, the impact of background noise weakens with distance. However, in FP-Growth, each alert co-occurrence will be counted equally towards the final association rules. UHAS considers the topological similarity by simply calculating the distance. LiDAR employs a more expressive machine learning model, i.e., *node2vec* [56], an algorithmic framework for learning a continuous representation of a network's nodes. However, they both ignore the problem of incomplete failure-impact graph, which is a common issue in online service systems according to our study. The necessity of completing the impact graph will be demonstrated in RQ3. Moreover, different from the traditional applications of graph representation learning, we learn a representation for each unique type of alert, compactly encoding its relationship with others.

RQ3: The Necessity of the Failure-Impact Graph for Alert Aggregation

We demonstrate the importance of impact graphs by creating a variant of Girdle without the phase of failure-impact graph completion (i.e., phase two in Figure 7.1), denoted as Girdle′. We follow LiDAR to remove this feature, which considers two alerts as related only when they are directly connected in the system topology. The experimental results are presented in Table 7.5, where we can see a noticeable drop in the NMI score for all datasets. Due to the high complexity and large scale of online service systems, monitors are often configured in an ad-hoc manner. These monitors may not be able to accommodate the ever-changing systems and environments. Thus, some alerts are not successfully captured by them. System engineers may incorrectly perceive the service as healthy, which is a typical situation of gray failures [74]. Without completing the impact graph of failures, the true correlations among alerts cannot be fully recovered.

Table 7.5: Experimental Results of Alert Aggregation using GIRDLE (w/ and w/o failure-impact graph completion)

Method	Dataset1	Dataset2	Dataset3
GIRDLE	**0.831**	**0.866**	**0.912**
GIRDLE′	0.782	0.808	0.846

7.3.4 Threats to Validity

During our study, we have identified the following major threats to the validity.

Labeling noise. Our experiments are conducted based on six months of real-world alerts collected from Huawei Cloud. The evaluation requires engineers to inspect and label the alerts manually. Label noises (false positives/false negatives) may be introduced during the manual labeling process. However, the engineers we invite are cloud system professionals and have years of system troubleshooting experience. Moreover, the labeling work can be done quickly and confidently thanks to the alert management system, which has user-friendly interfaces. Therefore, we believe the amount of noise is small (if it exists).

Selection of study subjects. In our experiments, we only collect alerts from one online service of Huawei Cloud, i.e., the Networking service. This is a large-scale service that supports many upper-layer services, such as web application, virtual machine. Sufficient data can be collected from this service system. Another benefit we can enjoy is that the topology of the Networking service system is readily available and accurate. Although we use the Networking service as the subject, our proposed framework is generalizable, as this service is a typical, representative online service. Thus, we believe GIRDLE can be applied to other services and cloud computing platforms and benefit them.

The second type of subject that could threaten the validity

is the metric. In production systems, there is a large number of metrics available to gauge the similarity between two nodes. Although we only select six representative metrics (as presented in Section 7.3.1), they record the basic and critical states of a service component. Thus, we believe they are able to profile the service system comprehensively.

Implementation and parameter setting. The implementation and parameter setting are two critical internal threats to the validity. To reduce the threat of implementation, we employ peer code review. Specifically, the authors are invited to carefully check others' code for mistakes. In terms of parameter setting, we conduct many groups of comparative experiments with different parameters. We choose the parameters by following the original work or empirically based on the best experimental results. In particular, we found GIRDLE is not very sensitive to the parameter setting.

7.4 Discussion

7.4.1 Success Story

GIRDLE has been successfully incorporated into the alert management system of Huawei Cloud. Based on the positive feedback we have received, on-site engineers (OSEs) highly appreciated the novelty of our approach and benefited from it during their daily system maintenance. Specifically, OSEs confirmed the difficulty of the auto-detection of service failures in the existing monitoring system. This is because simple detection techniques (e.g., fixed thresholding) are widely adopted. GIRDLE introduces more intelligence and automation by leveraging EVT-based alert burst detection. Interestingly, OSEs found problems for some monitors by comparing their configurations with the aggregated alerts, including wrong names, missing information, etc. Mean-

while, during failure diagnosis, alert aggregation assists OSEs in reducing their investigation scope. Before the deployment of GIRDLE, they would have to examine a large number of alerts to locate the failures.

To quantify the practical benefits conveyed to the Networking service system, we further collect failure tickets generated during November 2020. In total, 26 failures are recorded. We calculate the average failure handling time in November and compare it with that in August, September, and October. Results show that the time reduction rate is 24.8%, 21.9%, and 18.6%, respectively, demonstrating the effectiveness of GIRDLE in accelerating the alert management of Huawei Cloud.

7.4.2 Lessons Learned

Optimizing monitor configurations. Today, popular online services are serving tens of millions of customers. During daily operations, they can produce terabytes and even petabytes of telemetry data such as metrics, logs, and alerts. However, the majority of these data do not contain much valuable information for service failure analysis. For example, a significant portion of metrics only record plain system runtime states; most of the alerts are trivial and likely to mitigate automatically with time. The configuration of system monitors should be optimized to report more important yet fewer alerts. In the meantime, monitor configurations show different styles across different service teams, making the monitoring data heterogeneous. Standards should be established for monitor configurations so that high-quality alerts can be created to facilitate the follow-up system analysis, e.g., fault localization.

Building data collection pipeline. In online service systems, IT operations play a critical role in system maintenance. Since it is data-driven by nature, modern cloud service providers

should build a complete and efficient pipeline for monitoring data collection. Common data quality issues include extremely imbalanced data, a small quantity of data, poor signal-to-noise ratio, etc. In general, we are facing the following three challenges: *1) What data should be collected?* We need to identify what metrics and events that are most representative of cloud resource health. Not everything that can be measured needs to be monitored. *2) How to collect and label data?* Labeling alerts (e.g., alert linkages, culprit alerts) requires OSEs to have a decent knowledge of the cloud systems. Since they often devote themselves to emerging issue mitigation and resolution, tools should be developed to facilitate the labeling process, such as label recommendations and friendly interfaces. *3) How to store and query data?* Today's cloud monitoring data are challenging conventional database systems. To save space, domain-specific compression techniques should be developed, for example, log compression [103, 33, 66].

7.5 Summary

In this chapter, we propose GIRDLE, an alert aggregation framework based on graph representation learning. The representation for different types of alerts is learned in an unsupervised and unified fashion, which encodes the interactions among alerts in both temporal and topological dimensions. Online alert aggregation can be efficiently performed by calculating their distance. We have conducted experiments with real-world alerts collected from Huawei Cloud. Compared with fixed thresholding, GIRDLE achieves better performance in failure detection by being able to adjust the threshold automatically. In terms of online alert aggregation, GIRDLE also outperforms existing methods by a noticeable margin, confirming its effectiveness. Furthermore, our framework has been successfully incorporated into the alert

management system of Huawei Cloud. Feedback from on-site engineers confirms its practical usefulness. We believe our proposed alert aggregation framework can assist engineers in failure understanding and diagnosis.

□ **End of chapter.**

Chapter 8

Conclusion and Future Work

8.1 Conclusion

Recent decades have witnessed an increasing prevalence of online services providing a variety of applications in our daily lives, e.g., search engines, social media, and translation applications. Different from traditional on-premises software, online services often serve hundreds of millions of customers worldwide with a goal of 24x7 availability. With such an unprecedented scale and complexity, how service incidents and performance degradation are managed becomes a core competence in the market. This thesis describes our research of intelligent reliability monitoring and engineering for online service systems based on various IT data, i.e., logs, metrics, alerts/events, and topologies.

In Chapter 4, we investigate the current status of incident management at Microsoft based on over two years of production incident tickets. Our study reveals some key challenges of incident handling, which we try to address in this thesis. Specifically, to alleviate flooding alarms and gray failures, we delve into log- and metric-based anomaly detection in Chapter 5 and Chapter 6. To facilitate failure impact estimation and duplicate effort saving, we perform alert aggregation in Chapter 7. We also present the following main findings regarding the key challenges: (1) the modularity design of software systems prevents quick root cause

localization; (2) the virtualization of physical infrastructure further complicates this problem; (3) the system's fault-tolerant capability leads to a trade-off between system availability and service management efficiency; and (4) system monitoring needs a top-down design, which should consider how and what service resources to monitor. We also present Microsoft's incident management framework based on AIOps techniques, emphasizing the essential role played by various monitoring data.

In Chapter 5, we systematically compare six representative log anomaly detectors which draw support from neural networks. Researchers and practitioners can obtain a deep understanding of their characteristics and limitations. Our main findings include: (1) logs' semantics is a useful feature, especially for unsupervised methods; (2) forecasting-based methods are vulnerable to the noises in the training data, while reconstruction-based ones have more resilience; (3) when it comes to unseen logs, supervised methods exhibit better robustness against such logs than unsupervised methods. We also release a toolkit containing the studied methods, which has attracted attention from both academia and industry. Finally, we summarize the key challenges of pursuing more intelligent and reliable log-based anomaly detection, as well as interesting future directions toward industrial deployment.

In Chapter 6, we introduce ADSketch, a metric-based performance anomaly detection method with the merits of interpretability and adaptability. Our key observation is that similar and repetitive unusual metric patterns often indicate similar types of performance anomalies. By capturing such patterns, we can immediately recognize what anomaly has happened if it corresponds to a known pattern. To embrace unseen patterns in online scenarios, we carefully compare the distance between the new metric inputs and the extracted patterns. If a new input is close to a known pattern, it will be fused into the pattern. Otherwise, it will be regarded as a brand-new anomalous

pattern. ADSketch is evaluated on both public and industrial datasets from Huawei Cloud. The results have demonstrated its state-of-the-art performance. Moreover, ADSketch has been incorporated into Huawei Cloud, serving hundreds of millions of service instances and devices.

In Chapter 7, we present GIRDLE, an unsupervised and unified alert aggregation framework based on graph representation learning. GIRDLE automatically gathers alerts stemming from the same failure. This helps engineers understand what problems have happened and which services get impacted, based on which they can plan proper recovery operations. To this end, existing work assumes that related alerts are similar in text. However, experience shows that this may not always be the case. Thus, we resort to measuring services' behavioral similarity during failures by integrating multi-source information. This can help us precisely track the propagation of failures and, thus, related alerts. Graph representation learning further unifies the temporal and topological correlations between alerts. Experiments conducted on the data from Huawei Cloud demonstrate the effectiveness of GIRDLE. Moreover, GIRDLE has been deployed in the network infrastructure of Huawei Cloud. Feedback from on-site engineers confirms its practical benefits.

In summary, we have conducted studies to address some important problems in online service monitoring, including an empirical study on the industrial practices of incident management, a systematic review of deep learning techniques for log anomaly detection, an interpretable and adaptive performance anomaly detection algorithm based on metrics, and an unsupervised and unified alert aggregation framework. We closely collaborate with the industry to pursue practicality. We also release our codes and data whenever possible to benefit the community.

8.2 Future Work

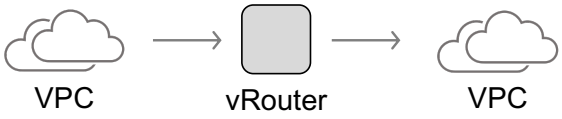
Service reliability monitoring and engineering have been a long-standing research topic. In this thesis, we present our effort toward intelligent online service monitoring. Although the techniques and algorithms that we developed can quickly detect problems that deserve engineers' attention, they do so in a passive manner. That is, they need to collect various monitoring data and analyze them to identify the problems. Moreover, we focus more on the software aspect of the cloud system, i.e., the SaaS and PaaS layers. To further prompt the resilience of cloud systems such that customers can enjoy always-on services, the performance monitoring should be done in a proactive way and extended to the lower stack of the cloud systems, i.e., the IaaS layer, and particularly, the cloud network infrastructure. To this end, we propose to explore two lines of future work, aiming to ultimately realize the full-stack monitoring for cloud systems.

The first is *performance monitoring and diagnosis for cloud overlay networks*. Cloud overlay networks allow tenants to construct sophisticated virtual networks for service deployment. Performance issues, e.g., packet losses and delay spikes, could severely compromise tenants' experiences. Therefore, cloud providers are keen to proactively monitor and quickly determine the root cause of such problems. The second is *cross-layer failure propagation modeling in cloud systems*. Nowadays, various algorithms and systems have been developed to monitor the health state of different parts of a cloud system, regardless of SaaS, PaaS, or IaaS. However, more often than not, they limit the problem scope to a specific service or layer by assuming that others function normally, which we refer to as "single-point" monitoring. In reality, the propagation of failures could be cross-layer and cross-service, which should be taken into consideration.

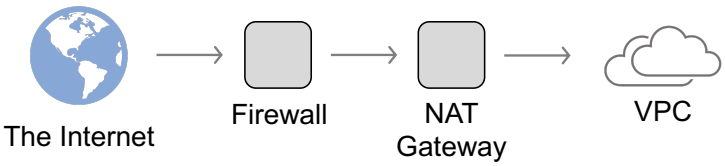
8.2.1 Performance Monitoring and Diagnosis for Cloud Overlay Networks

Cloud overlay networks are created by abstracting the underlying physical infrastructure with network virtualization technologies [77, 85]. They enable running isolated virtual private networks (VPCs) for multi-tenants on a shared physical network. In cloud networks, tenants' traffic is transmitted by both *physical forwarding devices* (PFDs, e.g., switches, routers) and *virtual network gateways* (VGWs). VGWs implement the traffic forwarding rules using software and run as network services in end-hosts. A VGW can act as a switch, a router, a proxy server, a firewall, etc. Compared to physical networks, overlay networks exhibit more complexity in two aspects. First, there could be tens of VGWs designed for various business needs. The production traffic of tenants' VMs is transmitted among different VGWs. Such traffic is called *virtual flows* [42], which could have more than one hundred different types. Second, they experience more frequent updates for a version upgrade or routing table modification. This renders their proneness to faults. However, comparatively little effort has been made regarding the monitoring and diagnosing of cloud overlay networks [142, 42]. To ensure the availability of cloud networking services, it is essential to proactively monitor the performance of virtual flows and automatically diagnose the problems, instead of passively starting analysis tasks upon the arrival of user complaints [42].

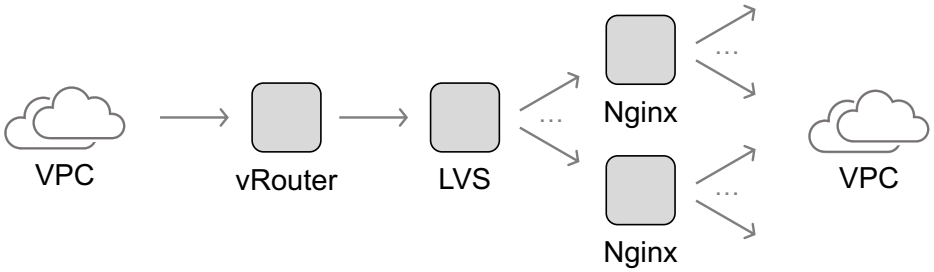
In different application scenarios, virtual flows may cross different types of VGWs in distinct and determined orders. Figure 8.1 briefs some typical types of virtual flows. In physical network monitoring, active probing [156, 128, 195] employs end-hosts to inject and trace probing packets in the network to infer its internal characteristics (e.g., loss rates, delays, bandwidth), which is called network tomography. The performance of a link/device (e.g., packet loss ratio, delay) can be uniquely inferred with



(a) VPC Peering. VPC peering is a networking connection between two VPCs, which enables traffic between them as if they are within the same network. The traffic between them is routed by a virtual router (vRouter).



(b) VPC Accessing the Internet. The traffic between the Internet and VPC traverses through two VGWs, i.e., Firewall for security monitoring and NAT Gateway for public IP and private IP replacement.



(c) Load Balancing. The traffic from the source VPC is routed to the destination VPC by vRouter, which then goes through two VGWs for load balancing. The LVS (Linux Virtual Server) balances Layer 4's traffic, while Nginx does Layer 7's.

Figure 8.1: Virtual Flows in Cloud Overlay Networks

multiple coordinated probing paths, i.e., identifiable link/device. Similarly, we can deploy VMs in the overlay network to send probes along virtual flows to measure the flow state. By leveraging the algebraic relations among virtual flows, we can also make a VGW identifiable by probing appropriate paths. Figure 8.2 demonstrates some examples. Virtual flow f_1 alone suffices to tell the performance of VGW v_3. f_1, f_2, and f_3 can constitute a linear system. Solving it can give us the performance of v_5.

Similar to previous work [59, 128, 195, 24, 156] in physical

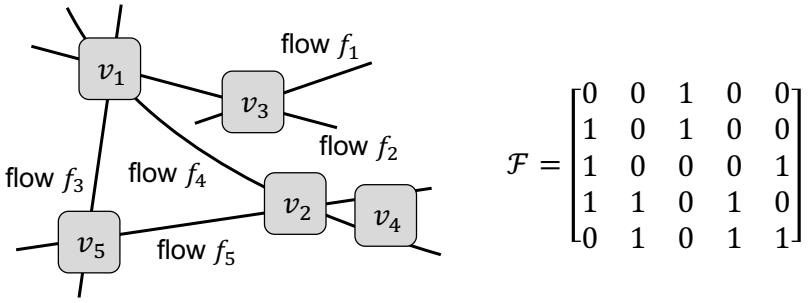


Figure 8.2: Examples of Identifiable VGWs. A virtual flow f_i is represented as a binary row vector $(a_{i1}, \ldots, a_{i5})$. a_{ij}=1 if f_i traverses through VGW v_j.

networks, minimizing the number of probing paths is desirable. This is to reduce network resource (e.g., end-host memory/CPU and network bandwidth) consumption and analysis overhead. However, active probing in the overlay network has its unique challenges. First, the network of virtual flows does not possess the topological regularity as the physical network, e.g., the Clos network [3]. Such a property is required by [59, 156]. Moreover, to balance tenants' traffic and increase fault tolerance, each VGW instance has multiple replicas running in virtual machines (VMs). Tenants' traffic is distributed to these replicas via load balancing. This should be considered as it may result in different probing paths having distinct probing results, even though they pass the same VGW. There are two promising directions to delve into. The first is network telemetry [204, 154], which provides us with enough insights into the virtual flows. The second is passive monitoring [143], which resorts to machine-learning techniques for fault detection and localization.

8.2.2 Cross-layer Failure Propagation Modeling in Cloud Systems

In the area of cloud system reliability, tremendous efforts have been devoted to the monitoring and diagnosis of physical networks [143, 59, 156, 155, 5], virtual/overlay/SDN networks [42,

142, 171], microservices [200, 199, 136, 22], compute nodes [99, 92], disks [62, 63, 170], databases [109], etc. by leveraging different types of runtime data such as network telemetry data [204, 201], logs [30, 68, 66, 43, 70], metrics [31, 188, 109], traces [200, 199], alerts [162, 191], incident tickets [91, 32, 29, 20, 100], etc. These works have achieved remarkable performance. However, when performing root cause localization or failure diagnosis, some of them may hold an assumption that may not be realistic in production systems: the fault scope is restricted to the subject under discussion and other dependent components are assumed to function normally. For example, when inferring the causes of performance problems in microservices systems, it is assumed that the culprit(s) is meant to be one microservice or two. However, the genuine faults could be a network issue, a hardware issue, or a user-side issue. As a result, the problem investigation has to continue even if the "culprit" microservice has been located. Therefore, it is essential to model the propagation of failure across different layers and components in cloud systems.

In this context, some existing work has been carried out. For example, [7, 29, 93] have pointed out that the dependencies between multiple services/microservices/APIs and physical/logical resources play an important role in anomaly detection, fault localization, service migration planning, etc. Qiu et al. [137] organized as a knowledge graph the interactions among routers, switches, physical servers, containers, and microservices in cloud systems to perform causality mining. Roy et al. [142] discovered that different layers of the Microsoft Azure stack could complicate the latency measurement of the logical overlay networks and their interpretation. Such a tangle of different cloud layers is hard to tease apart. The authors identified some patterns in the latency measurements to distinguish virtual-network issues from the ones in the physical network. Lapukhov et al. [131] adopted a similar idea for the troubleshooting of data center networks

at Meta. Rusek et al. [145] made an attempt to estimate the mean delay and jitter of traffic flows by considering the complex relationship between network topology, routing, and input traffic.

To precisely model how failures propagate in a cloud system, we need to explore two orthogonal directions of dependency mining and how different dependencies transfer the failure impact. The first direction is *intra-layer*, which captures the dependencies in each individual layer of the cloud, including SaaS, PaaS, and IaaS. In SaaS and PaaS, failures often propagate along the dependency chains among services. Typical sources for mining service dependencies are codes, traces, and API documentation. In IaaS, the network topology (i.e., the connections between different physical devices such as switches, routers, and servers) is a typical example of such intra-layer dependency. The second direction is *inter-layer*, which reveals the interactions of different cloud system stacks. For example, the placement between servers and services (i.e., which server is hosting the service) can clearly indicate the failure propagation from hardware to software. In the first line of our future work (Section 8.2.1), we can also consider the identifiability of physical devices by tracking how virtual flows traverse through physical switches (or switch clusters) based on flow-level telemetry [201, 204, 165], routing protocols, switch configurations, etc., or directly controlling the path of virtual flows using techniques such as IP-in-IP [156, 16]. By doing so, we can unify the performance monitoring and diagnosis of both physical and overlay networks without manually defining failure patterns for fault attribution as done in [142, 131].

The scale of today's cloud systems has imposed considerable complexity on system reliability assurance. Being able to model the propagation of failures in a full-stack manner plays a vital role in bringing cloud system monitoring to the next generation.

□ **End of chapter.**

Chapter 9

List of Publications

1. He, Pinjia, **Zhuangbin Chen**, Shilin He, and Michael R. Lyu. "Characterizing the natural language descriptions in software logging statements." In 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 178-189. IEEE, 2018.

2. Bai, Haoli, **Zhuangbin Chen**, Michael R. Lyu, Irwin King, and Zenglin Xu. "Neural relational topic models for scientific article analysis." In Proceedings of the 27th ACM International Conference on Information and Knowledge Management, pp. 27-36. 2018.

3. Xu, Hui, **Zhuangbin Chen**, Weibin Wu, Zhi Jin, Sy-yen Kuo, and Michael Lyu. "NV-DNN: towards fault-tolerant DNN systems with N-version programming" In 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), pp. 44-47. IEEE, 2019.

4. **Zhuangbin Chen**, Yu Kang, Feng Gao, Li Yang, Jeffrey Sun, Zhangwei Xu, Pu Zhao et al. "Aiops innovations of incident management for cloud services" (2020).

5. **Zhuangbin Chen**, Yu Kang, Liqun Li, Xu Zhang, Hongyu Zhang, Hui Xu, Yangfan Zhou 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, pp. 1487-1497. 2020.

6. He, Shilin, Pinjia He, **Zhuangbin Chen**, Tianyi Yang, Yuxin Su, and Michael R. Lyu. "A survey on automated log analysis for reliability engineering" ACM Computing Surveys (CSUR) 54, no. 6 (2021): 1-37.

7. **Zhuangbin Chen**, Jinyang Liu, Wenwei Gu, Yuxin Su, and Michael R. Lyu. "Experience report: deep learning-based system log analysis for anomaly detection" arXiv preprint arXiv:2107.05908 (2021).

8. Xu, Hui, **Zhuangbin Chen**, Mingshen Sun, Yangfan Zhou, and Michael R. Lyu. "Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs" ACM Transactions on Software Engineering and Methodology (TOSEM) 31, no. 1 (2021): 1-25.

9. **Zhuangbin Chen**, Jinyang Liu, Yuxin Su, Hongyu Zhang, Xuemin Wen, Xiao Ling, Yongqiang Yang, and Michael R. Lyu. "Graph-based Incident Aggregation for Large-Scale Online Service Systems" In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 430-442. IEEE, 2021.

10. **Zhuangbin Chen**, Jinyang Liu, Yuxin Su, Hongyu Zhang, Xiao Ling, Yongqiang Yang, and Michael R. Lyu. "Adaptive performance anomaly detection for online service systems via pattern sketching" In Proceedings of the 44th International Conference on Software Engineering (ICSE), pp. 61–72. 2022.

11. Li, Yichen, Xu Zhang, Shilin He, **Zhuangbin Chen**, Yu Kang, Jinyang Liu, Liqun Li et al. "An Intelligent Framework for Timely, Accurate, and Comprehensive Cloud Incident Detection" ACM SIGOPS Operating Systems Review, pp. 1-7. 2022.

12. Li, Baitong, Tianyi Yang, **Zhuangbin Chen**, Yuxin Su, Yongqiang Yang, and Michael R. Lyu. "Heterogeneous Anomaly Detection for Software Systems via Attentive Multi-modal Learning" arXiv preprint arXiv:2207.02918 (2022).

Chapter 4 is an adapted reprint of publication "Towards intelligent incident management: why we need it and how we make it," which was published in ACM ESEC/FSE 2020. The thesis author was the primary author of this paper. The paper is co-authored with Yu Kang, Liqun Li, Xu Zhang, Li Yang, Jeffrey Sun, Zhangwei Xu, Yingnong Dang, Feng Gao, Pu Zhao, Bo Qiao, Qingwei Lin, and Dongmei Zhang from Microsoft, Hongyu Zhang from The University of Newcastle, Hui Xu and Yangfan Zhou from Fudan University, and Michael R. Lyu from The Chinese University of Hong Kong.

Chapter 5 is an adapted reprint of arXiv preprint "Experience report: deep learning-based system log analysis for anomaly detection." The thesis author was the primary author of this paper. This paper is collaborated with Jinyang Liu, Wenwei Gu, and Michael R. Lyu from The Chinese University of Hong Kong, Yuxin Su from Sun Yat-sen University, and Jieming Zhu and Yongqiang Yang from Huawei.

Chapter 6 is an adapted reprint of publication "Adaptive performance anomaly detection for online service systems via pattern sketching," which was published in IEEE/ACM ICSE 2022. The thesis author is the primary author of this paper. This is a joint work with Jinyang Liu and Michael R. Lyu from The

Chinese University of Hong Kong, Yuxin Su from Sun Yat-sen University, Hongyu Zhang from The University of Newcastle, and Xiao Ling and Yongqiang Yang from Huawei.

Chapter 7 is an adapted reprint of publication "Graph-based Incident Aggregation for Large-Scale Online Service Systems," which was published in IEEE/ACM ASE 2021. The thesis author was the primary author of this paper. This paper is co-authored with Jinyang Liu, Yuxin Su, and Michael R. Lyu from The Chinese University of Hong Kong, Hongyu Zhang from The University of Newcastle, and Xuemin Wen, Xiao Ling, and Yongqiang Yang from Huawei.

□ **End of chapter.**

Bibliography

[1] Kpi anomaly detection competition, 2018. Available online at: `http://iops.ai/competition_detail/?competition_id=5&flag=1`, last accessed in April, 2021.

[2] Kpi anomaly detection dataset, 2018. Available online at: `http://iops.ai/dataset_detail/?id=10`, last accessed in April, 2021.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In V. Bahl, D. Wetherall, S. Savage, and I. Stoica, editors, *Proceedings of the ACM SIGCOMM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Seattle, WA, USA, August 17-22, 2008*, pages 63–74. ACM, 2008.

[4] A. Arcuri. Restful api automated test case generation with evomaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(1):1–37, 2019.

[5] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. H. Liu, J. Padhye, B. T. Loo, and G. Outhred. 007: Democratically finding the cause of packet drops. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 419–435, 2018.

[6] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[7] V. Bahl, P. Barham, R. Black, R. Chandra, M. Goldszmidt, R. Isaacs, S. Kandula, L. Li, J. MacCormick, D. Maltz, et al. Discovering dependencies for network management. 2006.

[8] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 18–18, 2004.

[9] S. Basu, F. Casati, and F. Daniel. Toward web service dependency discovery for soa management. In *Proceedings of the 2008 IEEE International Conference on Services Computing (SCC)*, pages 422–429. IEEE, 2008.

[10] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[11] S. Boyd, S. P. Boyd, and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[12] M. Braei and S. Wagner. Anomaly detection in univariate time-series: A survey on the state-of-the-art. *arXiv preprint arXiv:2004.00433*, 2020.

[13] Á. Brandón, M. Solé, A. Huélamo, D. Solans, M. S. Pérez, and V. Muntés-Mulero. Graph-based root cause analysis for service-oriented and microservice architectures. *Journal of Systems and Software*, 159:110432, 2020.

[14] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Proceedings of the 2001 IEEE/IFIP International Symposium on Integrated Network Management. Integrated Network Management VII. Integrated Management Strategies for the New Millennium (Cat. No. 01EX470)*, pages 377–390. IEEE, 2001.

[15] J. Candido, M. Aniche, and A. van Deursen. Contemporary software monitoring: A systematic literature review. *arXiv e-prints*, pages arXiv–1912, 2019.

[16] J. Cao, R. Xia, P. Yang, C. Guo, G. Lu, L. Yuan, Y. Zheng, H. Wu, Y. Xiong, and D. Maltz. Per-packet load-balanced, low-latency routing for clos-based data center networks. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 49–60, 2013.

[17] M. J. Chambers and M. A. Thornton. Discrete time representation of continuous time arma processes. *Econometric Theory*, pages 219–238, 2012.

[18] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):1–58, 2009.

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

[20] J. Chen, X. He, Q. Lin, Y. Xu, H. Zhang, D. Hao, F. Gao, Z. Xu, Y. Dang, and D. Zhang. An empirical investigation of incident triage for online service systems. In *Proceedings*

of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pages 111–120. IEEE Press, 2019.

[21] J. Chen, X. He, Q. Lin, H. Zhang, D. Hao, F. Gao, Z. Xu, Y. Dang, and D. Zhang. Continuous incident triage for large-scale online service systems. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 364–375. IEEE, 2019.

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

[23] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *OSDI*, volume 8, pages 117–130, 2008.

[24] Y. Chen, D. Bindel, H. H. Song, and R. H. Katz. An algebraic approach to practical and scalable overlay network monitoring. In R. Yavatkar, E. W. Zegura, and J. Rexford, editors, *Proceedings of the ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 30 - September 3, 2004, Portland, Oregon, USA*, pages 55–66. ACM, 2004.

[25] 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*, pages 304–314, 2020.

[26] 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 2019 International Conference on World Wide Web (WWW)*, pages 2659–2665, 2019.

[27] Y.-Y. M. Chen, A. J. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E. A. Brewer. *Path-based failure and evolution management*. University of California, Berkeley, 2004.

[28] Z. Chen, Y. Kang, F. Gao, L. Yang, J. Sun, Z. Xu, P. Zhao, B. Qiao, L. Li, X. Zhang, et al. Aiops innovations of incident management for cloud services. 2020.

[29] 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*, pages 1487–1497, 2020.

[30] 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.

[31] 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. *arXiv preprint arXiv:2201.02944*, 2022.

[32] Z. Chen, J. Liu, Y. Su, H. Zhang, X. Wen, X. Ling, Y. Yang, and M. R. Lyu. Graph-based incident aggregation for large-scale online service systems. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 430–442. IEEE, 2021.

[33] R. Christensen and F. Li. Adaptive log compression for massive log data. In *SIGMOD Conference*, pages 1283–1284, 2013.

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

[35] M. T. Community. Message templates, 2011. Available online at: `https://messagetemplates.org/`, last accessed in October, 2021.

[36] H. Dai, H. Li, C. S. Chen, W. Shang, and T.-H. Chen. Logram: Efficient log parsing using n-gram dictionaries. *TSE*, 2020.

[37] Y. Dang, Q. Lin, and P. Huang. Aiops: real-world challenges and research innovations. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 4–5. IEEE, 2019.

[38] L. De Haan and A. Ferreira. *Extreme value theory: an introduction*. Springer Science & Business Media, 2007.

[39] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie. Log2: A cost-aware logging mechanism for performance diagnosis. In *2015 USENIX annual technical conference (USENIX ATC 15)*, pages 139–150, 2015.

[40] 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.

[41] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In E. Simoudis, J. Han, and U. M. Fayyad, editors, *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*, pages 226–231. AAAI Press.

[42] C. Fang, H. Liu, M. Miao, J. Ye, L. Wang, W. Zhang, D. Kang, B. Lyv, P. Cheng, and J. Chen. Vtrace: Automatic diagnostic system for persistent packet loss in cloud-scale overlay network. In H. Schulzrinne and V. Misra, editors, *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 31–43. ACM, 2020.

[43] M. Farshchi, J.-G. Schneider, I. Weber, and J. Grundy. Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis. In *2015 IEEE 26th international symposium on software reliability engineering (ISSRE)*, pages 24–34. IEEE, 2015.

[44] A. Farzad and T. A. Gulliver. Unsupervised log message anomaly detection. *ICT Express*, 6(3):229–237, 2020.

[45] V. Fedorova, A. Gammerman, I. Nouretdinov, and V. Vovk. Plug-in martingales for testing exchangeability on-line. In *Proceedings of the 29th International Conference on Machine Learning (ICML)*, pages 923–930, 2012.

[46] R. A. Fisher and L. H. C. Tippett. Limiting forms of the frequency distribution of the largest or smallest member of a sample. In *Mathematical proceedings of the Cambridge philosophical society*, volume 24, pages 180–190. Cambridge University Press, 1928.

[47] A. Flink, 2011. Available online at: `https://flink.apache.org/`, last accessed in October, 2021.

[48] A. S. Foundation. Apache kafka, 2011. Available online at: `https://kafka.apache.org/`, last accessed in October, 2021.

[49] B. J. Frey and D. Dueck. Clustering by passing messages between data points. *science*, 315(5814):972–976, 2007.

[50] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proc. of ICDM'09*, pages 149–158. IEEE, 2009.

[51] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie. Where do developers log? an empirical study on logging practices in industry. In *Proc. of ICSE-C'14*, pages 24–33, 2014.

[52] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia. A survey on concept drift adaptation. *ACM computing surveys (CSUR)*, 46(4):1–37, 2014.

[53] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[54] J. Gao, N. Yaseen, R. MacDavid, F. V. Frujeri, V. Liu, R. Bianchini, R. Aditya, X. Wang, H. Lee, D. Maltz, et al. Scouts: Improving the diagnosis process through domain-customized incident routing. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 253–269, 2020.

[55] B. Gnedenko. Sur la distribution limite du terme maximum d'une serie aleatoire. *Annals of mathematics*, pages 423–453, 1943.

[56] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[57] J. Gu, C. Luo, S. Qin, B. Qiao, Q. Lin, H. Zhang, Z. Li, Y. Dang, S. Cai, W. Wu, et al. Efficient incident identification from multi-dimensional issue reports via meta-heuristic search. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 292–303, 2020.

[58] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, pages 1–16. ACM, 2016.

[59] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. A. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In S. Uhlig,

O. Maennel, B. Karp, and J. Padhye, editors, *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 139–152. ACM, 2015.

[60] W. L. Hamilton, R. Ying, and J. Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017.

[61] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *ACM sigmod record*, 29(2):1–12, 2000.

[62] S. Han, P. P. Lee, Z. Shen, C. He, Y. Liu, and T. Huang. Toward adaptive disk failure prediction via stream mining. In *Proceedings of IEEE ICDCS*, 2020.

[63] S. Han, J. Wu, E. Xu, C. He, P. P. Lee, Y. Qiang, Q. Zheng, T. Huang, Z. Huang, and R. Li. Robust data preprocessing for machine-learning-based disk failure prediction in cloud production environments. *arXiv preprint arXiv:1912.09722*, 2019.

[64] S. E. Hansen and E. T. Atkins. Automated system monitoring and notification with swatch. In *Proc. of LISA'93*, volume 93, pages 145–152, 1993.

[65] M. Hassani, W. Shang, E. Shihab, and N. Tsantalis. Studying and detecting log-related issues. *Empirical Software Engineering*, 23(6):3248–3280, 2018.

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

[67] S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu. A survey on automated log analysis for reliability engineering. *arXiv preprint arXiv:2009.07237*, 2020.

[68] S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu. A survey on automated log analysis for reliability engineering. *CSUR*, 54(6):1–37, 2021.

[69] 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.

[70] S. He, J. Zhu, P. He, and M. R. Lyu. Experience report: System log analysis for anomaly detection. In *Proc. of ISSRE'16*, pages 207–218. IEEE, 2016.

[71] S. He, J. Zhu, P. He, and M. R. Lyu. Loghub: a large collection of system log datasets towards automated log analytics. *arXiv preprint arXiv:2008.06448*, 2020.

[72] S.-S. Ho and H. Wechsler. A martingale framework for detecting changes in data streams by testing exchangeability. *IEEE transactions on pattern analysis and machine intelligence*, 32(12):2113–2127, 2010.

[73] H. Hu, H. Zhang, J. Xuan, and W. Sun. Effective bug triage based on historical bug-fix information. In *Proceedings of the 25th International Symposium on Software Reliability Engineering (ISSRE)*, pages 122–132. IEEE, 2014.

[74] 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*, pages 150–155, 2017.

[75] K. Hundman, V. Constantinou, C. Laporte, I. Colwell, and T. Soderstrom. Detecting spacecraft anomalies using lstms and nonparametric dynamic thresholding. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 387–395, 2018.

[76] M. Inc. Everything you need to know about aiops, 2019. Available online at: `https://www.moogsoft.com/resources/aiops/guide/everything-aiops/`.

[77] R. Jain and S. Paul. Network virtualization and software defined networking for cloud computing: a survey. *IEEE Communications Magazine*, 51(11):24–31, 2013.

[78] G. Jiang, H. Chen, K. Yoshihira, and A. Saxena. Ranking the importance of alerts for problem determination in large computer systems. 14(3):213–227.

[79] R. Johnson and T. Zhang. Deep pyramid convolutional neural networks for text categorization. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 562–570, 2017.

[80] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov. Fasttext. zip: Compressing text classification models. *arXiv preprint arXiv:1612.03651*, 2016.

[81] E. Keogh and C. A. Ratanamahatana. Exact indexing of dynamic time warping. *Knowledge and information systems*, 7(3):358–386, 2005.

[82] S. Khan, A. Gani, A. W. A. Wahab, M. A. Bagiwa, M. Shiraz, S. U. Khan, R. Buyya, and A. Y. Zomaya. Cloud log forensics: foundations, state of the art, and future directions. *CSUR*, 49(1):1–42, 2016.

[83] M. O. Kherbouche, N. Laga, and P.-A. Masse. Towards a better assessment of event logs quality. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8. IEEE, 2016.

[84] Y. Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, page 1746–1751, 2014.

[85] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2014.

[86] M. Landauer, F. Skopik, M. Wurzenberger, and A. Rauber. System log clustering approaches for cyber security applications: A survey. *Computers & Security*, 92:101739, 2020.

[87] 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.

[88] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proc. of the IEEE*, 86(11):2278–2324, 1998.

[89] S.-R. Lee, M.-J. Heo, C.-G. Lee, M. Kim, and G. Jeong. Applying deep learning based automatic bug triager to industrial projects. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, pages 926–931, 2017.

[90] H. Li, W. Shang, and A. E. Hassan. Which log level should developers choose for a new logging statement? *Empirical Software Engineering*, 22(4):1684–1716, 2017.

[91] 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 *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 131–146, 2021.

[92] Y. Li, Z. M. Jiang, H. Li, A. E. Hassan, C. He, R. Huang, Z. Zeng, M. Wang, and P. Chen. Predicting node failures in an ultra-large-scale cloud computing platform: an aiops solution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(2):1–24, 2020.

[93] 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.

[94] Z. Li, Q. Cheng, K. Hsieh, Y. Dang, P. Huang, P. Singh, X. Yang, Q. Lin, Y. Wu, S. Levy, et al. Gandalf: An intelligent,{End-To-End} analytics service for safe deployment in {Large-Scale} cloud infrastructure. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 389–402, 2020.

[95] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo. Failure prediction in ibm bluegene/l event logs. In *Proc. of ICDM'07*, pages 583–588. IEEE, 2007.

[96] M.-H. Lim, J.-G. Lou, H. Zhang, Q. Fu, A. B. J. Teoh, Q. Lin, R. Ding, and D. Zhang. Identifying recurrent

and unknown performance issues. In R. Kumar, H. Toivonen, J. Pei, J. Z. Huang, and X. Wu, editors, *2014 IEEE International Conference on Data Mining, ICDM 2014, Shenzhen, China, December 14-17, 2014*, pages 320–329. IEEE Computer Society.

[97] D. Lin, R. Raghu, V. Ramamurthy, J. Yu, R. Radhakrishnan, and J. Fernandez. Unveiling clusters of events for alert and incident management in large-scale enterprise it. In S. A. Macskassy, C. Perlich, J. Leskovec, W. Wang, and R. Ghani, editors, *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, pages 1630–1639. ACM.

[98] J. Lin, P. Chen, and Z. Zheng. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In *International Conference on Service-Oriented Computing*, pages 3–20. Springer, 2018.

[99] Q. Lin, K. Hsieh, Y. Dang, H. Zhang, K. Sui, Y. Xu, J.-G. Lou, C. Li, Y. Wu, R. Yao, et al. Predicting node failure in cloud service systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 480–490, 2018.

[100] Q. Lin, J.-G. Lou, H. Zhang, and D. Zhang. idice: problem identification for emerging issues. In *Proceedings of the 38th International Conference on Software Engineering*, pages 214–224, 2016.

[101] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen. Log clustering based problem identification for online service systems. In *2016 IEEE/ACM 38th International Confer-*

ence on Software Engineering Companion (ICSE-C), pages 102–111. IEEE, 2016.

[102] 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.

[103] 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 *Proc. of ASE'19*, pages 863–873. IEEE, 2019.

[104] P. Liu, Y. Chen, X. Nie, J. Zhu, S. Zhang, K. Sui, M. Zhang, and D. Pei. Fluxrank: A widely-deployable framework to automatically localizing root cause machines for software service failure mitigation. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 35–46. IEEE, 2019.

[105] C. Lou, P. Huang, and S. Smith. Understanding, detecting and localizing partial failures in large system software. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 559–574, 2020.

[106] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li. Mining invariants from console logs for system problem detection. In *Proc. of USENIXATC'10*, pages 1–14, 2010.

[107] J.-G. Lou, Q. Lin, R. Ding, Q. Fu, D. Zhang, and T. Xie. Software analytics for incident management of online services: An experience report. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 475–485. IEEE, 2013.

[108] S. Lu, X. Wei, Y. Li, and L. Wang. Detecting anomaly in big data system logs using convolutional neural network. In

Proc. of DASC/PiCom/DataCom/CyberSciTech'18, pages 151–158. IEEE, 2018.

[109] M. Ma, Z. Yin, S. Zhang, S. Wang, C. Zheng, X. Jiang, H. Hu, C. Luo, Y. Li, N. Qiu, et al. Diagnosing root causes of intermittent slow queries in cloud databases. *Proceedings of the VLDB Endowment*, 13(10):1176–1189, 2020.

[110] S. Ma, J. Zhai, Y. Kwon, K. H. Lee, X. Zhang, G. Ciocarlie, A. Gehani, V. Yegneswaran, D. Xu, and S. Jha. Kernel-supported cost-effective audit logging for causality tracking. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 241–254, 2018.

[111] S. Ma, X. Zhang, and D. Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS*, volume 2, page 4, 2016.

[112] S.-P. Ma, C.-Y. Fan, Y. Chuang, W.-T. Lee, S.-J. Lee, and N.-L. Hsueh. Using service dependency graph to analyze and test microservices. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 81–86. IEEE, 2018.

[113] A. Mastropaolo, L. Pascarella, and G. Bavota. Using deep learning to generate complete log statements. *arXiv preprint arXiv:2201.04837*, 2022.

[114] 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 7, pages 4739–4745, 2019.

[115] R. Mercer, S. Alaee, A. Abdoli, S. Singh, A. Murillo, and E. Keogh. Matrix profile xxiii: Contrast profile: A novel

time series primitive that allows real world classification. In *The IEEE International Conference on Data Mining*, 2021.

[116] H. Mi, H. Wang, Y. Zhou, M. R.-T. Lyu, and H. Cai. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1245–1255, 2013.

[117] T. Mikolov, E. Grave, P. Bojanowski, C. Puhrsch, and A. Joulin. Advances in pre-training distributed word representations. *arXiv preprint arXiv:1712.09405*, 2017.

[118] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[119] V. I. Munteanu, A. Edmonds, T. M. Bohnert, and T.-F. Fortis. Cloud incident management, challenges, research directions, and architectural approach. In *Proceedings of the 7th International Conference on Utility and Cloud Computing (UCC)*, pages 786–791. IEEE Computer Society, 2014.

[120] A. Nandi, A. Mandal, S. Atreja, G. B. Dasgupta, and S. Bhattacharya. Anomaly detection using program control flow graph mining from execution logs. In *Proc. of SIGKDD'16*, pages 215–224, 2016.

[121] A. Natarajan, P. Ning, Y. Liu, S. Jajodia, and S. E. Hutchinson. *NSDMiner: Automated discovery of network service dependencies*. IEEE, 2012.

[122] S. Nedelkoski, J. Bogatinovski, A. Acker, J. Cardoso, and O. Kao. Self-attentive classification-based anomaly detec-

tion in unstructured logs. *arXiv preprint arXiv:2008.09340*, 2020.

[123] K. A. Nguyen, S. S. i. Walde, and N. T. Vu. Integrating distributional lexical contrast into word embeddings for antonym-synonym distinction. *arXiv preprint arXiv:1605.07766*, 2016.

[124] D. Ohana, B. Wassermann, N. Dupuis, E. Kolodner, E. Raichstein, and M. Malka. Hybrid anomaly detection and prioritization for network logs at cloud scale. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 236–250, 2022.

[125] A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *Proc. of DSN'07*, pages 575–584. IEEE, 2007.

[126] G. Pang, K. M. Ting, and D. W. Albrecht. Lesinn: Detecting anomalies by identifying least similar nearest neighbours. In *IEEE International Conference on Data Mining Workshop, ICDMW 2015, Atlantic City, NJ, USA, November 14-17, 2015*, pages 623–630. IEEE Computer Society, 2015.

[127] D. Park, Y. Hoshi, and C. C. Kemp. A multimodal anomaly detector for robot-assisted feeding using an lstm-based variational autoencoder. *IEEE Robotics and Automation Letters*, 3(3):1544–1551, 2018.

[128] Y. Peng, J. Yang, C. Wu, C. Guo, C. Hu, and Z. Li. detector: a topology-aware monitoring system for data center networks. In D. D. Silva and B. Ford, editors, *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 55–68. USENIX Association, 2017.

[129] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Proc. of EMNLP'14*, pages 1532–1543, 2014.

[130] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

[131] A. A. Petr Lapukhov. Netnorad: Troubleshooting networks via end-to-end probing, 2016. Available online at: `https://engineering.fb.com/2016/02/18/core-data/netnorad-troubleshooting-networks-via-end-to-end-probing/`.

[132] T. Pevnỳ. Loda: Lightweight on-line detector of anomalies. *Machine Learning*, 102(2):275–304, 2016.

[133] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 465–475. IEEE, 2003.

[134] J. E. Prewett. Analyzing cluster log files using logsurfer. In *Proc. of the 4th Annual Conference on Linux Clusters*. Citeseer, 2003.

[135] PyTorch, 2016. Available online at: `https://pytorch.org/`, last accessed in May, 2021.

[136] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer. {FIRM}: An intelligent fine-grained resource management framework for {SLO-Oriented} microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825, 2020.

[137] J. Qiu, Q. Du, K. Yin, S.-L. Zhang, and C. Qian. A causality mining and knowledge graph based method of root cause diagnosis for performance anomaly in cloud applications. *Applied Sciences*, 10(6):2166, 2020.

[138] T. Rakthanmanon and E. Keogh. Fast shapelets: A scalable algorithm for discovering time series shapelets. In *proceedings of the 2013 SIAM International Conference on Data Mining*, pages 668–676. SIAM, 2013.

[139] H. Ren, B. Xu, Y. Wang, C. Yi, C. Huang, X. Kou, T. Xing, M. Yang, J. Tong, and Q. Zhang. Time-series anomaly detection service at microsoft. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3009–3017, 2019.

[140] Y. Research. A benchmark dataset for time series anomaly detection, 2015. Available online at: `https://yahooresearch.tumblr.com/post/114590420346/a-benchmark-dataset-for-time-series-anomaly`, last accessed in August, 2021.

[141] K. Rodrigues, Y. Luo, and D. Yuan. Clp: Efficient and scalable search on compressed text logs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 183–198, 2021.

[142] A. Roy, D. Bansal, D. Brumley, H. K. Chandrappa, P. Sharma, R. Tewari, B. Arzani, and A. C. Snoeren. Cloud datacenter SDN monitoring: Experiences and challenges. In *Proceedings of the Internet Measurement Conference 2018, IMC 2018, Boston, MA, USA, October 31 - November 02, 2018*, pages 464–470. ACM, 2018.

[143] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren. Passive realtime datacenter fault detection and localization. In

14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17), pages 595–612, 2017.

[144] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[145] K. Rusek, J. Suárez-Varela, A. Mestres, P. Barlet-Ros, and A. Cabellos-Aparicio. Unveiling the potential of graph neural networks for network modeling and optimization in SDN. In *Proceedings of the 2019 ACM Symposium on SDN Research, SOSR 2019, San Jose, CA, USA, April 3-4, 2019*, pages 140–151. ACM, 2019.

[146] B. Russo, G. Succi, and W. Pedrycz. Mining system logs to learn error predictors: a case study of a telemetry system. *Empirical Software Engineering*, 20(4):879–927, 2015.

[147] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *Proc. of SIGKDD'03*, pages 426–435, 2003.

[148] W. Shang, M. Nagappan, and A. E. Hassan. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, 20(1):1–27, 2015.

[149] J. Shen, T. Yang, Y. Su, Y. Zhou, and M. R. Lyu. Defuse: A dependency-guided function scheduler to mitigate cold starts on faas platforms. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 194–204. IEEE, 2021.

[150] A. Siffer, P. Fouque, A. Termier, and C. Largouët. Anomaly detection in streams with extreme value theory. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*, pages 1067–1075. ACM, 2017.

[151] J. Sillito and E. Kutomi. Failures and fixes: A study of software system incident response. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 185–195. IEEE, 2020.

[152] Stanford. *Evaluation of clustering*, 2008 [Online; accessed November-2020]. `https://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-clustering-1.html`.

[153] 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 ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2828–2837, 2019.

[154] P. Tammana, R. Agarwal, and M. Lee. Simplifying datacenter network debugging with {PathDump}. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 233–248, 2016.

[155] P. Tammana, R. Agarwal, and M. Lee. Distributed network monitoring and debugging with switchpointer. In S. Banerjee and S. Seshan, editors, *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 453–466. USENIX Association, 2018.

[156] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, and D. Xiang. Netbouncer: Active device and link failure localization in data center networks. In J. R. Lorch and M. Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 599–614. USENIX Association, 2019.

[157] Tensorflow, 2015. Available online at: `https://www.tensorflow.org/`, last accessed in May, 2021.

[158] C. Trubiani, P. Jamshidi, J. Cito, W. Shang, Z. M. Jiang, and M. Borg. Performance issues? hey devops, mind the uncertainty. *IEEE Software*, 36(2):110–117, 2018.

[159] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin. Attention is all you need. In *Proc. of NIPS'17*, 2017.

[160] E. Viglianisi, M. Dallago, and M. Ceccato. Resttestgen: automated black-box testing of restful apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 142–152. IEEE, 2020.

[161] C. Wang, X. Peng, M. Liu, Z. Xing, X. Bai, B. Xie, and T. Wang. A learning-based approach for automatic construction of domain glossary from source code and documentation. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 97–108, 2019.

[162] H. Wang, Z. Wu, H. Jiang, Y. Huang, J. Wang, S. Kopru, and T. Xie. Groot: An event-graph-based approach for root cause analysis in industrial settings. In *2021 36th*

IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 419–429. IEEE, 2021.

[163] Wikipedia. Representational state transfer, 2022. Available online at: `https://en.wikipedia.org/wiki/Representational_state_transfer`.

[164] T. Wittkopp, P. Wiesner, D. Scheinert, and O. Kao. A taxonomy of anomalies in log data. *arXiv preprint arXiv:2111.13462*, 2021.

[165] W. Wu, G. Wang, A. Akella, and A. Shaikh. Virtual network diagnosis as a service. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–15, 2013.

[166] B. Xia, Y. Bai, J. Yin, Y. Li, and J. Xu. Loggan: A log-level generative adversarial network for anomaly detection using permutation event modeling. *Information Systems Frontiers*, 23(2):285–298, 2021.

[167] H. Xu, W. Chen, N. Zhao, Z. Li, J. Bu, Z. Li, Y. Liu, Y. Zhao, D. Pei, Y. Feng, et al. Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications. In *Proceedings of the 2018 World Wide Web Conference*, pages 187–196, 2018.

[168] J. Xu, Y. Wang, P. Chen, and P. Wang. Lightweight and adaptive service api performance monitoring in highly dynamic cloud environment. In X. F. Liu and U. Bellur, editors, *2017 IEEE International Conference on Services Computing, SCC 2017, Honolulu, HI, USA, June 25-30, 2017*, pages 35–43. IEEE Computer Society.

[169] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proc. of SOSP'09*, pages 117–132, 2009.

[170] Y. Xu, K. Sui, R. Yao, H. Zhang, Q. Lin, Y. Dang, P. Li, K. Jiang, W. Zhang, J.-G. Lou, et al. Improving service availability of cloud systems by predicting disk error. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC)*, pages 481–494, 2018.

[171] G. Yang, H. Jin, M. Kang, G. J. Moon, and C. Yoo. Network monitoring for sdn virtual networks. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 1261–1270. IEEE, 2020.

[172] H. Yang, L. Wen, and J. Wang. An approach to evaluate the local completeness of an event log. In *2012 IEEE 12th International Conference on Data Mining*, pages 1164–1169. IEEE, 2012.

[173] 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 *Proc. of ICSE'21*, pages 1448–1460. IEEE, 2021.

[174] N. Yang, R. Schiffelers, and J. Lukkien. An interview study of how developers use execution logs in embedded software engineering. In *Proc. of ICSE-SEIP'21*, pages 61–70. IEEE, 2021.

[175] 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 *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 653–665. IEEE, 2021.

[176] Z. Yang, R. Algesheimer, and C. J. Tessone. A comparative analysis of community detection algorithms on artificial networks. *Scientific reports*, 6:30750, 2016.

[177] D. Yankov, E. Keogh, and U. Rebbapragada. Disk aware discord discovery: Finding unusual time series in terabyte sized datasets. *Knowledge and Information Systems*, 17(2):241–262, 2008.

[178] K. Yao, H. Li, W. Shang, and A. E. Hassan. A study of the performance of general compressors on log files. *Empirical Software Engineering*, 25(5):3043–3085, 2020.

[179] C.-C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, D. F. Silva, A. Mueen, and E. Keogh. Matrix profile i: all pairs similarity joins for time series: a unifying view that includes motifs, discords and shapelets. In *2016 IEEE 16th international conference on data mining (ICDM)*, pages 1317–1322. IEEE, 2016.

[180] S. Yen, M. Moh, and T.-S. Moh. Causalconvlstm: Semi-supervised log anomaly detection through sequence modeling. In *Proc. of ICMLA'19*, pages 1334–1341. IEEE, 2019.

[181] J. Yin, X. Zhao, Y. Tang, C. Zhi, Z. Chen, and Z. Wu. Cloudscout: A non-intrusive approach to service dependency discovery. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1271–1284, 2016.

[182] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proc. of ASPLOS'10*, pages 143–154, 2010.

[183] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proc. of OSDI'12*, pages 293–306, 2012.

[184] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)*, 30(1):1–28, 2012.

[185] T. S. Zaman, X. Han, and T. Yu. Scminer: Localizing system-level concurrency faults from large system call traces. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 515–526. IEEE, 2019.

[186] Y. Zeng, J. Chen, W. Shang, and T.-H. P. Chen. Studying the characteristics of logging practices in mobile apps: a case study on f-droid. *Empirical Software Engineering*, 24(6):3394–3434, 2019.

[187] H. Zhang, L. Gong, and S. Versteeg. Predicting bug-fixing time: an empirical study of commercial software projects. In *Proceedings of the 35th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 1042–1051. IEEE Press, 2013.

[188] X. Zhang, Q. Lin, Y. Xu, S. Qin, H. Zhang, B. Qiao, Y. Dang, X. Yang, Q. Cheng, M. Chintalapati, et al. Cross-dataset time series anomaly detection for cloud systems. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC)*, pages 1063–1076, 2019.

[189] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, et al. Robust log-based anomaly detection on unstable log data. In *Proc. of ESEC/FSE'19*, pages 807–817, 2019.

[190] 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.

[191] 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 ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, pages 162–171, 2020.

[192] N. Zhao, J. Chen, Z. Wang, X. Peng, G. Wang, Y. Wu, F. Zhou, Z. Feng, X. Nie, W. Zhang, et al. Real-time incident prediction for online service systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 315–326, 2020.

[193] N. Zhao, P. Jin, L. Wang, X. Yang, R. Liu, W. Zhang, K. Sui, and D. Pei. Automatically and adaptively identifying severe alerts for online service systems. In *39th IEEE Conference on Computer Communications, INFOCOM 2020, Toronto, ON, Canada, July 6-9, 2020*, pages 2420–2429. IEEE.

[194] N. Zhao, H. Wang, Z. Li, X. Peng, G. Wang, Z. Pan, et al. An empirical investigation of practical log anomaly detection for online service systems. In *Proc. of ESEC/FSE'21*, pages 1404–1415, 2021.

[195] Q. Zheng and G. Cao. Minimizing probing cost and achieving identifiability in probe-based network link monitoring. *IEEE Trans. Computers*, 62(3):510–523, 2013.

[196] 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 IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 352–363. IEEE, 2019.

[197] H. Zhou, J.-G. Lou, H. Zhang, H. Lin, H. Lin, and T. Qin. An empirical study on quality issues of production big data platform. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 17–26. IEEE Press, 2015.

[198] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

[199] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260, 2018.

[200] 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 *Proc. of ESEC/FSE'19*, pages 683–694, 2019.

[201] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, P. Zhang, D. Cai, M. Zhang, and M. Xu. Flow event telemetry on programmable data plane. In H. Schulzrinne and V. Misra, editors, *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 76–89. ACM, 2020.

[202] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang. Learning to log: Helping developers make informed logging decisions. In *Proc. of ICSE'15*, volume 1, pages 415–425. IEEE, 2015.

[203] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu. Tools and benchmarks for automated log parsing. In *Proc. of ICSE-SEIP'19*, pages 121–130. IEEE, 2019.

[204] Y. Zhu, N. Kang, J. Cao, A. G. Greenberg, G. Lu, R. Mahajan, D. A. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-level telemetry in large datacenter networks. In S. Uhlig, O. Maennel, B. Karp, and J. Padhye, editors, *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 479–491. ACM, 2015.

[205] Y. Zhu, C.-C. M. Yeh, Z. Zimmerman, K. Kamgar, and E. Keogh. Matrix profile xi: Scrimp++: time series motif discovery at interactive speeds. In *2018 IEEE International Conference on Data Mining (ICDM)*, pages 837–846. IEEE, 2018.

[206] 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 *International Conference on Learning Representations*, 2018.

[207] R. Řehůřek. Gensim: topic modelling for humans, 2009. Available online at: `https://radimrehurek.com/gensim/`.