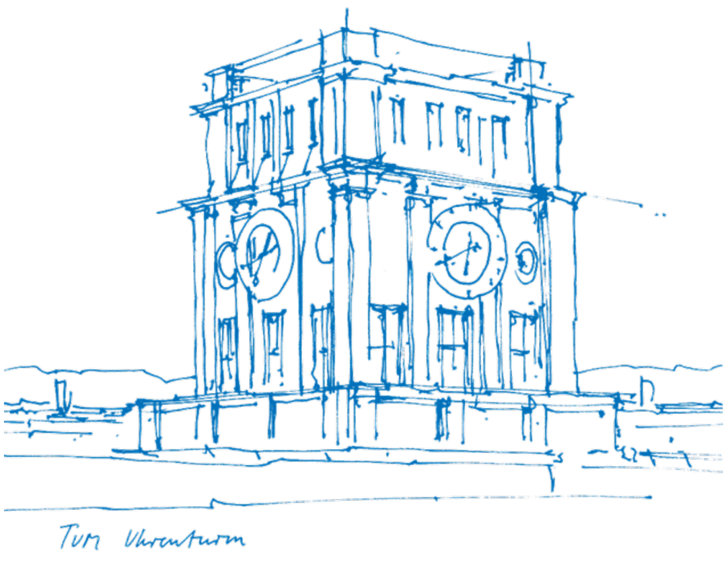
TUM School of Computation, Information and Technology
Technical University of Munich

AI-based Proactive Failure Management in Large-scale Cloud Environments

Paolo Notaro



AI-based Proactive Failure Management in Large-scale Cloud Environments

Paolo Notaro

TUM School of Computation, Information and Technology
Technische Universität München

AI-based Proactive Failure Management in Large-scale Cloud Environments

Paolo Notaro

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:
Prof. Dr. Helmut Seidl

Prüfer der Dissertation:
1. Prof. Dr. Hans Michael Gerndt
2. Prof. Dr. Dieter Kranzlmüller

Die Dissertation wurde am 13.09.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 27.05.2024 angenommen.

Zusammenfassung

Um die wachsende Nachfrage nach Cloud-Computing-Diensten zu befriedigen, haben moderne Computerinfrastrukturen einen rasanten Anstieg in Umfang und Komplexität erfahren. Dies hat auch zu einem raschen Anstieg des Volumens und der Heterogenität der Überwachungsdaten geführt, die für den effektiven Betrieb von IT-Diensten mit hoher Fehlertransparenz erforderlich sind. Dies stellt eine Herausforderung für Operations & Maintenance (O&M)-Teams dar, die für die Verwaltung und Reparatur der von der Cloud-Umgebung bereitgestellten Rechendienste verantwortlich sind. Ausfälle verursachen nicht nur weitreichende Serviceunterbrechungen, sondern erfordern auch komplexe und zeitaufwändige Untersuchungen, um die richtigen Reparaturlösungen zu ermitteln und anzuwenden.

Einfache automatisierte Tools können die Betreiber auf verschiedene Weise unterstützen, z. B. bei der Erkennung von Anomalien, der Korrelation der Ursachen und der automatischen Anwendung von Abhilfemaßnahmen. Allerdings ist die Durchführung von IT-Operationen nach diesem Paradigma bei den heutigen ultragroßen Systemen nicht mehr machbar. Dies hat führende Industrieunternehmen dazu veranlasst, das Feld in Richtung autonomer und intelligenter IT-Managementsysteme zu erforschen.

AI for IT Operations (AIOps) befasst sich mit Ansätzen, die auf Big Data, Machine Learning (ML) und anderen fortgeschrittenen Analysetechnologien basieren, um IT-Operationen zu verbessern. AIOps bietet hohe Vorteile, indem es einen großen, vielfältigen Satz von Überwachungsdaten (z. B. Protokolle, Metriken, Traces) und den höheren Grad an Verallgemeinerung nutzt, der durch KI-Algorithmen bereitgestellt wird. Bisherige Beiträge von AIOps beschränken sich jedoch auf eine kleine Anzahl von Aufgaben, wie z. B. die Erkennung von Anomalien und die Ursachenanalyse, die speziell auf die Reaktion auf Ausfälle ausgerichtet sind. Die Erforschung proaktiver Ansätze für das Fehlermanagement, die dazu beitragen könnten, die Nichtverfügbarkeit und Beeinträchtigung von Diensten zu verringern, beschränkte sich bisher hauptsächlich auf die Online-Fehlervorhersage von Hardwarekomponenten und die Verfügbarkeit von Knoten, während nur wenige Beiträge alternative Techniken zur Fehlervermeidung in Betracht gezogen haben. Darüber hinaus erschwert das Fehlen einer umfassenden Taxonomie von AIOps und eines gemeinsamen Terminologieschemas den Vergleich und die Anwendbarkeit von AIOps auf verschiedene Probleme.

Diese Dissertation befasst sich mit den oben genannten Herausforderungen, indem sie zunächst den vollen Umfang der bisherigen AIOps-Anwendungen durch eine systematische Durchsicht der vorhandenen Literatur erfasst, um die Gültigkeit und das Potenzial proaktiver Ansätze für das Fehlermanagement zu bestätigen. Die bisherigen Beiträge werden mit Hilfe der Systematic Mapping Study (SMS)-Methodik identifiziert und strukturiert, die es ermöglicht, eine Taxonomie gemeinsamer Ziele und Probleme abzuleiten sowie die AIOps-Beiträge auf der Grundlage von Zielsystemen, Datenquellen und KI-Methoden zu unterteilen. Die systematische Durchsicht der Beiträge ermöglicht es auch, blinde Flecken und unzureichend untersuchte Bereiche zu identifizieren, um vollständig zu verstehen, welche offenen Probleme in einem O&M-Kontext noch größere Forschungsanstrengungen erfordern.

Durch die Entwicklung neuer proaktiver Methoden auf der Grundlage von KI werden dann einige dieser wichtigen Probleme angegangen. Für jede der drei Schichten eines Cloud-Computing-Stacks wird eine Lösung vorgeschlagen. Auf der Infrastrukturebene wird ein Online-Algorithmus zur Vorhersage von Ausfällen in optischen Transceivern, grundlegenden, aber störanfälligen Komponenten moderner Rechenzentrumsinfrastrukturen, eingeführt. Auf der Plattformebene wird das Problem des sicheren Zugriffs auf O&M über die Command-Line Interface (CLI)-Schnittstelle mit Hilfe eines umfangreichen Sprachmodells zur Risikoklassifizierung angegangen, das die potenzielle Bedrohung durch die Ausführung von Befehlen abschätzt und deren Ausführung während des Abfangens verhindert. Auf der Softwareebene wird eine neuartige Muster-Korrelations-Engine für automatisierte Root Cause Analysis (RCA) und proaktive Failure Management (FM) vorgeschlagen, um automatisch Kausalitätsbeziehungen zwischen Symptomen und Fehlern zu identifizieren, die zu Ausfällen führen. Für jedes der vorgestellten O&M-Probleme bestätigen umfangreiche Experimente die Gültigkeit und Effektivität der vorgeschlagenen Lösung und demonstrieren die Funktionalität dieser KI-Methoden zur effektiven Behebung von Ausfällen und zur Reduzierung der Notwendigkeit menschlichen Eingreifens und der Gesamtauswirkungen von Ausfällen in der gesamten Cloud-Architektur.

Abstract

To satisfy the growing demand for cloud computing services, modern computing infrastructures have experienced a rapid increase in scale and complexity. This has also led to rapid growth in the volume and heterogeneity of monitoring data, necessary to effectively operate IT services with high failure visibility. This represents a challenge for Operations & Maintenance (O&M) teams, responsible for managing and repairing the computing services provided by the cloud environment. In addition to causing widespread service disruption, failures require IT operators to perform complex and time-consuming investigations to identify and apply correct repair solutions.

Simple automated tools can assist IT operators in various ways, including the detection of anomalies, correlation of root causes, and automatic application of remediation actions. However, performing IT operations according to this paradigm is no longer feasible at the current ultra-large scale of modern systems. This has motivated industry leaders to explore the field in the direction of autonomous and intelligent IT management systems.

AI for IT Operations (AIOps) deals with approaches based on Big Data, Machine Learning (ML), and other advanced analytics technologies to enhance IT operations. AIOps provides high benefits by taking advantage of a large, diverse set of monitoring data (e.g., logs, metrics, traces) and the higher degree of generalization provided by AI algorithms. However, past AIOps contributions are restricted to a small set of tasks, such as anomaly detection and root cause analysis, which are specifically designed to respond in reaction to failures. The exploration of proactive approaches for failure management, which could help reduce service unavailability and degradation, has been so far limited mostly to online failure prediction of hardware components and node availability, while few contributions have considered alternative techniques for failure prevention. Moreover, the lack of a comprehensive taxonomy of AIOps, and a common terminology scheme hinders the comparison and applicability of AIOps to different problems.

This dissertation addresses the above challenges, firstly by understanding the full extent of past AIOps applications through a systematic review of the existing literature, to confirm the validity and potential of proactive approaches to failure management. The past contributions are identified and structured using the Systematic Mapping Study (SMS) methodology, which enables to derive a taxonomy of common goals and problems, as well as to divide AIOps contributions based on target systems, data sources, and AI methods. The systematic review of contributions also allows to identify blind spots and under-investigated areas, to fully understand which open problems in an O&M context still require major research effort.

Then, through the development of new AI-based proactive methods, some of these important problems are addressed. One solution is proposed for each of the three layers of a cloud computing stack. At the infrastructure level, an online algorithm for predicting failures in optical transceivers, fundamental yet failure-prone components of modern datacenter infrastructures, is introduced. At the platform level, the problem of secure O&M access via Command-Line Interface (CLI) interface is addressed by means of a Large Language Model (LLM) for risk classification, which estimates the potential threat caused by the execution of commands and prevents their execution during interception. At the software level, a novel pattern correlation engine for automated Root Cause Analysis (RCA) and proactive Failure Management (FM) is proposed to automatically identify causality relationships between symptoms and errors leading to failures. For each of the O&M problems introduced, extensive experiments confirm the validity and effectiveness of the proposed solution, demonstrating the functionality of these AI methods for addressing failures effectively and reducing the necessity of human intervention and the total impact of failures across the entire cloud architecture.

Acknowledgements

First and foremost, I would like to thank my supervisor, Prof. Dr. Michael Gerndt, for advising me over the last three and a half of years of my doctoral program. He has been highly supportive during our frequent discussions regarding my research, providing new directions and giving me motivation and feedback for my publications.

I would like to thank my colleagues of the AIOps team at Huawei, and especially Prof. Dr. Jorge Cardoso who, in his role of mentor, has shown great dedication on teaching me how to conduct applied research, and how and when a research project becomes a publication. Grateful thanks to Soroush for advising me and provide valuable feedback for my publications. Thanks also to Qiao, German, and Vittorio for our fruitful discussions over our respective projects. There is great value in the possibility to discuss openly and cooperatively about complex (and sometimes apparently undecidable) problems.

I would like to extend my thanks to my research lab colleagues at TUM, in particular to Anshul, Mohak, Srishti, and Jianfeng, and everyone I met at CAPS, as they supported me with advices and helpful discussions in different topics related to AI and cloud computing. We have shared good moments of social bonding which have proven to be supporting both academically and personally.

An enormous thank you goes to my family and friends who deeply supported me in the last months of drafting of my doctoral dissertation. Thanks to my family for supporting me and allowing to pursue my research activities.

A final thank you to my life partner Sara, who has been my daily support and motivation for completing this and all other important achievements in my life.

Contents

1 Introduction

1.1 Context

1.1.1 Large-scale Computing

Over the last decade, cloud computing has expanded to become ubiquitous across all sectors of the digital economy [1, 2]. Thanks to their cost efficiency, enterprises increasingly rely on Internet-delivered computing services [1] to meet a wide range of IT needs, such as e-mail, storage, corporate software, and security [2].

To satisfy this high computing demand, IT systems have grown larger and more complex. Strives for higher scalability and efficient resource utilization have also moved the attention towards decentralized systems, following the model of edge and fog computing. At the same time, public cloud providers have aimed for uncharted territory by adopting ultra-large scale computing systems, which incorporate from hundreds of thousands to millions of computing nodes into their datacenters.

The law of large numbers dictates how the expected number of failures is bound to rise with a corresponding increase in scale. Moreover, because the need to accommodate this large scale requires the use of commodity hardware and fast software development/deployment cycles, the chances of failures to occur are further aggravated.

The new dimension of modern systems, therefore, raises the questions of how to deal with increasing quantity of failures, and how to efficiently allocate and distribute all kinds of resources (computation, storage, hardware and power-related) in large-scale shared computing environments.

1.1.2 Artificial Intelligence

Almost in parallel with the adoption of cloud, another paradigm shift, related to the introduction of large and powerful intelligent software, so-called AI algorithms, has been taking place.

Starting in the early 2010s, the widespread presence of large amounts of data and the availability of more affordable and faster computers incentivized the development of new and complex intelligent algorithms.

Research on intelligent machines dates back to the 1950s [3], with the theoretical work of Turing on theory of computation and machine intelligence (including the famous definition of the Turing Test) [4]. After an initial period of high enthusiasm and expectations in the 1950s and 60s, since the early 1970s AI had undergone a series of setbacks and generalized loss of interest (so-called AI winters), due to unattended promises of solving complex mankind problems very soon.

The season inaugurated by the recent advances in deep learning, starting in early 2010s with the introduction of modern neural architectures [5–7], re-ignited the interest towards AI with a much larger focus on Machine Learning (ML) approaches, able to extrapolate patterns directly from data. This kind of inductive approach had for the first time sufficient hardware support and availability of very large datasets to express its true potential. The reborn interest in ML generated a large number of contributions for different Computer Vision (CV) applications, starting with image classification and evolving into detection, and segmentation, to then be extended to other application domains, such as Natural Language Processing (NLP), bioinformatics, and others.

Nowadays, AI algorithms are used in state-of-the-art solutions [8–10] for targeted advertising, content generation, autonomous driving and system control, game playing, search and recommendation, robotics, in addition to the above cited CV, NLP, and biomedical applications.

AI has highly benefited from cloud computing, as the growing dimension of ML models required increasingly high quantity of computing resources; at the same time, cloud computing enabled AI companies to provide machine learning models as a service to end users effectively.

1.1.3 Reliability and O&M

Because of the mass adoption of cloud computing and the strict dependability requirements imposed by modern challenges (such as the Internet of Things (IoT), Autonomous Driving, 6G, and Smart Grid), the reliability of IT infrastructures is nowadays more critical than ever.

At the scale of modern systems, failures are inevitable and can cause a significant amount of disruption during the normal operation of an IT service, which can rapidly turn into customer dissatisfaction. Because of large and complex nature of these modern distributed systems, IT administration has become more difficult and prone to the appearance of failures and performance issues. Operations & Maintenance (O&M) teams, responsible for overseeing and repairing IT services, are particularly challenged by the large quantity and multi-modality of monitoring data to track in real-time, which can easily be overwhelming even to specialized IT operators.

Unexpected faults also cause IT operators to interrupt their monitoring operations and devote their time to solve the encountered issues. Even in the presence of perfectly fault-tolerant systems, the occurrence of failures increases Operating Expense (OPEX) because of detection, diagnosis and reparation efforts [11–13], and complexity on the infrastructure stack to deal with the necessary hardware and software redundancy.

Cloud Computing can gain large benefit from software-automated solutions for O&M. Automatic analysis tools can assist IT operators by detecting anomalous patterns in execution traces, by performing in-depth analysis of root causes behind the appearance of errors, or by suggesting relevant remediation actions. Automated O&M software could also predict future failures, help preventing them, or divert the system towards less error-prone configurations. Moreover, intelligent scaling, scheduling and allocation techniques can be adopted to tackle the problem of resource provisioning.

Until a few years ago, solutions to support all these tasks were typically limited to the manual intervention of human operators, the use of rule-based algorithms, or simple thresholding techniques in order to trigger alarms and quick response actions.

1.1.4 The Advent of AIOps

Because of these limitations, performing IT operations according to this paradigm is no longer feasible at the current ultra-large scale of modern systems. This has motivated industry leaders to explore the field in the direction of data-driven, intelligent management systems, additionally inclined by the reborn interest towards AI and ML in the last decade.

This led, in 2017, Gartner to coin the term AI for IT Operations (AIOps) [14], to define *"big data, modern machine learning and other advanced analytics technologies to directly and indirectly enhance IT operations"*. Later on, other leading industry leaders have complemented their views on this new concept [15–21], and several research groups started taking advantage of the recent advances in Machine Learning and AI to explore open problems related to AIOps, such as Online Failure Prediction (OFP) [22–24], or anomaly detection [25].

In short, AIOps relies on data-driven technologies (ML, Big Data, Data Mining, Analytics and Visualization) to observe the operational status of the infrastructure, minimize the impact of failures during day-to-day operations, and manage the allocation of computer resources.

AIOps provides several advantages compared to traditional O&M:

- **Responsiveness**. AIOps allows to respond quickly and automatically, because it reacts independently and automatically to real-time problems, without requiring long manual debugging and analysis sessions. Failures may be detected before they start to manifest undesired behavior, so that they may be addressed *proactively*;

- **Adaptability**. AIOps offers a wide and diverse set of tools for several applications, thanks to the high applicability of AI algorithms. AIOps supports an efficient design and implementation of software systems by estimating the presence of faults and performance issues. AIOps also allows to forecast failures in the close future and to take preventive online actions, such as live migration and software rejuvenation. It also allows to detect and pinpoint performance bottlenecks and anomalies in real time, to locate faulty components, to identify AIOps causes and to suggest effective repair actions. Finally, AIOps offers different uses for efficient resource provisioning, including server consolidation, task scheduling, and workload estimation;

- **Efficiency**. ML is able to autonomously discover hidden patterns from the data and that may in conclusion enable more effective operations than manual and ruled-based approaches for O&M tasks, thanks to the higher degree of generalization provided. This aspect is especially favored by the almost ubiquitous presence of data sources, namely logs, Key Performance Indicator (KPI) metrics, execution traces, tickets, etc. By learning to predict from the existing data and act, ML allows to improve performance measures such as the Mean Time to Detect (MTTD) and Mean Time to Repair (MTTR), therefore improving service quality and reducing operating costs.

Moreover, AIOps can rely on a wide and diverse range of modern AI methods. To cite a few examples, deep learning and dimensionality reduction may be applied to detect anomalies based on the correct operational behavior of programs [26–28]; through forecasting, ML is able to predict future software [29] or hardware faults [11, 30], such as hard disk failures; similarity matching [31, 32] and inductive reasoning [33] can be used for Root Cause Analysis (RCA); autoregressive models [34, 35], Reinforcement Learning (RL) [36] and collaborative filtering [37] may be employed for resource provisioning.

1.2 Challenges and Motivation

The application of AI and ML for IT operations also poses several challenges in terms of applicability, runtime performance, and accuracy.

Because the advent of AIOps is relatively recent, it is still partially unclear how and where AI techniques can be applied successfully for IT Operations, as O&M techniques vary consistently depending on the abstraction level of the cloud offering, in terms of operations to perform and data sources to monitor. Therefore, it is important to evaluate the applicability of AI depending on the specific cloud model in use, to ensure that each offering can count on the most beneficial AIOps techniques, especially in relation to 1) which O&M operation must be automated and 2) which data sources are available and how they can be fully utilized.

It is also an open question how AI solutions, which are often CPU and memory intensive, can effectively run in production systems. In this context, performance of AIOps is also subject to the distributed nature of computing, where caching, concurrency, diversity of configuration and retry patterns are present. Automated solutions are also required to be scalable with respect to input data size, which can, in practice, reach processing rates of TBs per second. For instance, evaluating the health status of the whole hardware infrastructure requires to gather information from millions of components and perform an inference task for each and one of them. Then, when failures occur - at this scale, failures do occur on a daily basis - sufficient resources must be allocated to resolve them.

Finally and most importantly, AIOps solutions are required to be accurate when their findings correspond to remediation actions. If an incorrect action is taken based on the false prediction of an AI-based algorithm, the system may not resolve a determined problem as expected or, in the worst case, the algorithm may cause additional overhead in terms of costs and availability of computing resources, which are connected to monetary and fiduciary loss with customers.

Connecting the previous points of performance and accuracy, AIOps solutions must be cost-effective in connection to their accuracy, as the two variables are coupled to each other when it comes to evaluate whether an O&M is economically convenient to deploy. This means ensuring the expensive computational resources required to operate AI models do not overweigh the benefits provided by the automated resolution of a manual task (with potential accuracy errors).

One of the great advantages of AIOps is the possibility to automatically introspect the state of system in advance, to prevent failures before their occurrence. However, the majority of existing approaches [38] have focused on *reactive* approaches for failure management, which operate after failures have already occurred. The true power of introspection lies in the ability to discover hidden patterns before their consequences

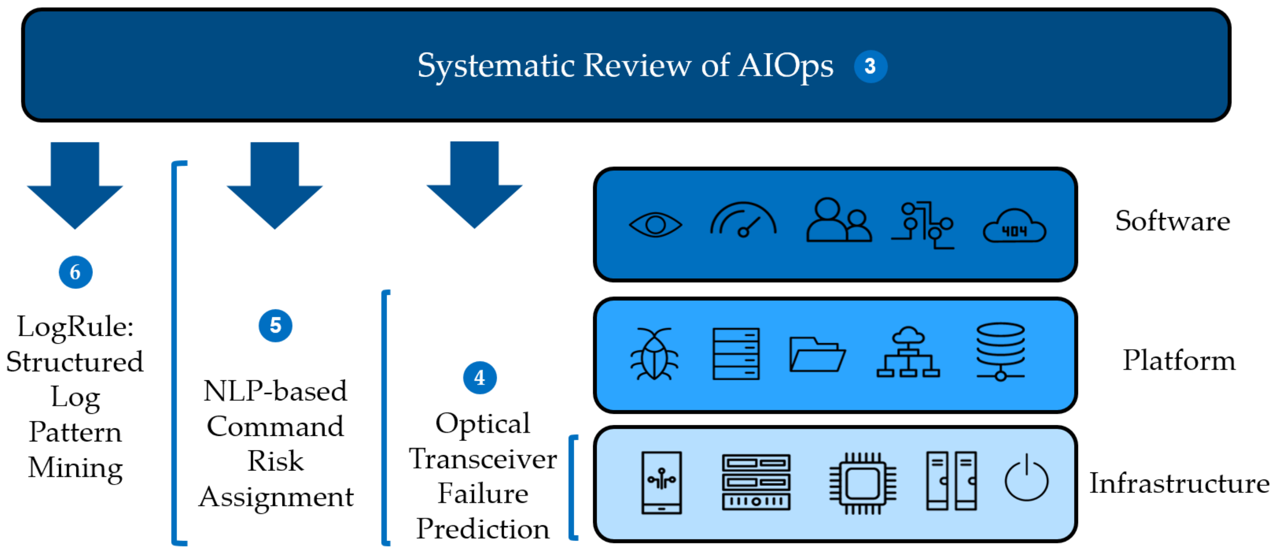

Systematic Review of AIOps 3
6
LogRule: Structured Log Pattern Mining
5
NLP-based Command Risk Assignment
4
Optical Transceiver Failure Prediction
Software
Platform
Infrastructure

Figure 1.1 Graphical summary of the contributions of this dissertation. The numbers in circles indicate the chapter where the contribution is presented inside this document.

manifest themselves. For this reason, more research should investigate the true ability of AIOps methods in anticipating failures rather than reacting to it.

1.3 Contribution of this Dissertation

Given the aforementioned aspects, this doctoral dissertation deals with existing challenges by exploring new AIOps solutions for large-scale cloud environments, where the large quantity of monitoring data allows AIOps to thrive. On this premise, it establishes the necessity to deeply understand the landscape of existing tasks, methods, target components, and macro-areas present inside the AIOps field.

Therefore, a systematic review of these different categories is conducted, followed by an evaluation of their relevance and prominence in the active research discussions. It is then important to identify unexplored or sub-covered topics and approaches which however show great potential from the industry application point of view. The main research questions concern how and to what extent the recent wave of AI methods and techniques can be applied to maintain and operate large-scale IT services, while pointing out limitations and possible caveats regarding applicability in the target domain.

In second instance, based on the insights gathered from this systematic review, this dissertation addresses open O&M problems in the large-scale cloud context, through the design of suitable *ad hoc* solutions, supported by adapting existing uses of AI in other research fields, while fulfilling special focusing criteria, such as applicability, performance and accuracy. In this context, the objective is to advance the current state of the art for AIOps through in-depth analysis and comparison of available tools, and 1) provide a solution, in the lack of existing approaches or 2) achieve an incremental improvement, by exploiting the benefits provided by AI. Solutions are studied to be applicable for all layers of the cloud computing stack (Figure 1.1), to tackle problems which are specific for each cloud offering.

Given the importance of reliability and dealing with failures at the O&M level, as described in the previous sections, particular emphasis is posed towards methods for dealing with failures. In particular, the aspects highlighted in the previous section motivate the need to explore more extensively the potential of AI-driven algorithms for *proactive* management of failures.

To this end, the dissertation contributes in characterizing and exploring the advantageous effects of proactive AIOps, by studying applicability scenarios, in the form of open O&M problems, and potential solutions, in the form of AI algorithms.

The contribution of this dissertation can be summarized as follows:

- **a systematic review of existing AIOps contributions** (Chapter 3), independent of computing environment (cloud, High Performance Computing (HPC), grid, cluster, *etc.*), with solutions classified by AI algorithms, data sources (such as logs, metrics, KPIs, source code), and target components (such as hardware, network, storage, virtualization). From a research perspective, this systematic review allows to draw actionable insights into what kind of algorithms have been less explored in the past and are more likely to produce effective solutions to open problems, given the criticality of target components and availability of data sources. The results of this systematic review confirm the importance of focusing on proactive failure management and motivate some of the design choices of the algorithm proposed in the following chapters;

- **the discussion of past AIOps solutions for proactive failure management**, in relation to their real-world application to different layers of a cloud stack model (Figure 1.1), composed of infrastructure layer (Chapter 4), platform layer (Chapter 5), and software layer (Chapter 6).

- **the introduction of several novel solutions for proactive failure management** on different cloud computing layers, through the exploration, implementation and evaluation in a real cloud production context. In particular,
  - At the cloud infrastructure level, **novel hardware failure prediction algorithms are proposed** for online estimation of future failures in critical and fault-prone components, in first instance optical transceivers (Section 4.7), and collaterally hard drives (Section 4.5), and Dynamic Random Access Memory (DRAM) chips (Section 4.6). The problem of OFP is thoroughly described and modeled through several assumptions regarding real-time operation, cost-benefit analysis, and failure modes of components. Then, ad hoc component solutions are proposed and evaluated;
  - At the cloud platform level, **the introduction of a NLP model for Command Risk Classification** (Section 5.4), which allows to evaluate and block dangerous Command-Line Interface (CLI) operations during the remote maintenance of cloud services. This solution leverages the recent advances of Large Language Models (LLMs) to provide high generalization power to a core security problem in the O&M context. The LLM solution described is general, so that it can be extended to several CLI-related applications to tackle equally numerous platform-level administration challenges, including system auditing, Standard Operating Procedure (SOP) verification, content extraction and more.
  - At the cloud software level, **the introduction of a novel RCA engine based on Association Rule Mining (ARM)**, called LogRule (Section 6.4), which is able to mine causal relationships between failures and observable symptoms to present a set of relevant root cause explanations. LogRule is able to reduce the runtime and improve the accuracy of provided explanations significantly, compared to existing methods. This approach can be applied for runtime verification to anticipate failures, for applications that include access log analysis, database pattern analysis, hardware failure prediction and more.

For each of these novel methods, the underlying context is introduced and the target problem is in-depth described, highlighting limitations of current approaches. Different potential solutions are evaluated and selected based on empirical and data-driven methodologies, to propose the most efficient, accurate and applicable solution. The final solutions are thoroughly evaluated and numerous use cases are presented.

A complete list of publications relevant to this dissertation is outlined in Appendix A.

1.4 Structure of the Dissertation

The remainder of the dissertation is structured as follows:

- **Chapter 2** presents a background on topics related to this dissertation. It covers Cloud Computing fundamentals, such as virtualization and cloud architecture; O&M and monitoring are defined with their main challenges and tools, including AIOps itself; AI and ML fundamentals, both as theoretical concepts and in relation to AIOps, are also presented;
- **Chapter 3** presents a systematic review of AIOps. Relevant AIOps contributions are retrieved using systematic search and selection criteria. Taxonomy and features of AIOps contributions are investigated from a research perspective, identifying macro-areas, common problems, and potential directions of improvement.
- **Chapter 4** discusses proactive Failure Management (FM) techniques at the infrastructure level. The infrastructure is decomposed into its fundamental subunits (hardware, Operating System (OS), Virtual Machine Manager (VMM)), and failure modes for each of these units is discussed. The rest of the chapter focuses on hardware failure prediction, where the failures of optical transceivers, hard drives, and DRAM chips are discussed. Predictive solutions for these components are discussed.
- **Chapter 5** discusses proactive FM techniques at the platform level. Platform-level failures and system administration challenges are presented. Related security concerns during O&M are discussed, and the problem of assessing risk to CLI operations introduced. Two LLM-based solutions for command risk classification, relying on command and documentation data respectively, are introduced and evaluated.
- **Chapter 6** discusses proactive FM techniques at the software level. Challenges related to operate large-scale services with high reliability and fault tolerance are illustrated. In this context, RCA techniques for incident response are discussed. To this end, ARM is introduced as an effective solution for automatic root cause determination. The LogRule algorithm is presented, including its working principles, application to different modalities of data, applications, and uses cases for proactive FM.
- **Chapter 7** summarizes the outcomes of the previous chapters and provides a perspective for future directions.

2 Background

This chapter presents a background on topics related to this dissertation. Section 2.1 presents cloud computing fundamentals, including definition, related technologies, architecture, and several cloud models. Section 2.2 describes Operations & Maintenance (O&M) in the cloud context, including a background on cloud monitoring and automated O&M. Section 2.3 presents a background of AI and Machine Learning (ML), providing a description of concepts, algorithms and tools frequently mentioned throughout the dissertation.

2.1 Cloud Computing

2.1.1 Definition

The National Institute of Standards and Technology (NIST) defines Cloud Computing as *"a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"* [39].

Some essential characteristics of this definition include [39, 40]:

- *on-demand network access*, i.e., users must be able to access compute and storage facilities with sufficient agility (as measured, e.g., by latency) and autonomy. Such access is provided over the Internet and should include *ubiquitous* access to and from personal computers, laptops, phones, tablets and more;
- *a shared pool of configurable pool resources*; i.e., pooling multiple compute resources together allows applying economy of scale through unified management of a large virtual resource;
- *rapidly provisioned and released*, i.e., the cloud service must provide rapid elasticity in relation to customer demand. Users can request extra resources in a self-managed and autonomic fashion, so that from the consumer's perspective, the supply of computing resources is limitless.

Cloud computing offerings are subject to Service Level Agreeements (SLAs), which describe minimum levels of service, described in terms of Key Performance Indicators (KPIs) (availability, throughput, latency), which must be provided to the final customers. Cloud computing relies on several related technologies to fulfill such requirements, and several cloud computing models have been introduced, as discussed in the next sections.

2.1.2 Cloud Technologies

Several computer system technologies enable the cloud to operate effectively. This section provides an overview of the most important and relevant ones to the discussion of this dissertation.

Datacenters

Datacenters are physical facilities that enterprises use to house and manage a large concentration of servers, providing computing and storage resources to run computing applications at large scale [12, 41]. Datacenters constitute the structural and operational foundations of cloud computing platforms [42].

Datacenters are different from supercomputers, as they are typically large and highly-parallel infrastructures, with high permanent storage capacity deployed separately, while supercomputers tend to focus on

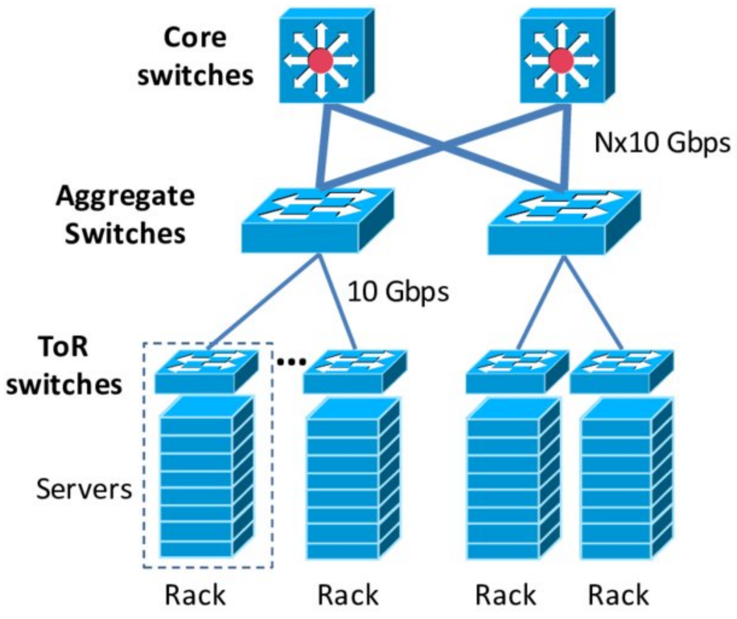


Figure 2.1 Typical intra-datacenter network architecture [43]. Servers are connected to ToR switches, which are then connected to aggregation and core switches. In the majority of datacenter applications, these links are realized by means of optical fiber. Each link requires at least one optical transceiver at each end of the link.

low-latency, high-bandwidth requirements which enable effective and fast communication across computing nodes.

Traditional datacenters host a large number of relatively small- or medium-sized applications, each running on a dedicated hardware infrastructure that is decoupled and protected from other systems in the same facility. They are usually utilized by multiple organizational units or enterprises.

Modern datacenters (or *warehouse-scale computers* [12]) usually belong to a single large company (Google, Facebook, Microsoft, Amazon, Alibaba, etc.), which operates a reduced number of large-scale applications. The same large-scale application may be distributed across several datacenters in different regions. While the specific hardware implementation of datacenters differs significantly, modern datacenters tend to use relatively homogeneous hardware and software systems within the same installation. Moreover, all software and hardware systems share a common O&M layer.

The scale of datacenters varies depending on needs. Datacenters can scale into two directions: they can *scale up*, i.e., they employ more powerful (and more expensive) hardware, in order to increase computational power and parallelization within the same machine; or they can *scale out*, i.e., they increase a large number of low-end, commodity servers to accommodate demand on separate machines.

Datacenters are typically composed of hundreds of thousands of low-end servers [12] deployed in compact-format (e.g., 1U) enclosures. Several servers are mounted inside a *rack* and interconnected to a local switch, known Top-of-Rack (ToR) switch. ToR switches are then connected via high-bandwidth (typically fiber-based) links to second-level switches (*edge aggregation switches*), which may correspond to a local cluster or the entire datacenter. Multiple second-level switches may be further aggregated to form a hierarchy of a tree-like interconnect topology. Because of their cost efficiency and high bandwidth, modern datacenters utilize fiber-optics interconnects for medium-range links (up to 100 m), such as connections between ToR switches and the edge aggregation switch, as well as for longer links (up to 2 km), e.g., the connection between edge aggregation switches and spine switches. An example diagram of the network infrastructure is depicted in Figure 2.1. Other hardware resources, such as Hard Disk Drives (HDDs) or Graphics Processing Units (GPUs), may be deployed within server enclosures or in separate racks for networked access.

The servers, the hardware components, the network interconnects, as well as the power supply units, the cooling systems, the racks and all other infrastructural components are placed in datacenter buildings composed of more rooms, which are designed to optimize resource utilization based on the energy (cooling, energy consumption, power dissipation) and capacity (workload, physical space) requirements.

Virtualization

Virtualization is the abstraction of computer resources [44, 45]. It introduces a software abstraction layer between the hardware and the guest Operating System (OS) with the applications running on top of it, which isolate the physical computer resources from the computing workload.

Virtualization offers several benefits for datacenter management [40]:

- applications can be confined within virtualized instances, which allows to increase security and isolate any detrimental effect of poor performance on the rest of the datacenter;

- physical utilization can be better managed, because it enables the consolidation of diverse platforms and software workloads onto a unified hardware layer, leading to increased energy efficiency;

- virtualization allows guest OS images to be restored rapidly in the event of a disaster. This also facilitates the capture of provenance data for forensic investigation purposes or for scenario replications.

One fundamental method to implement virtualization is the use of a specialized software program known as *hypervisor* or Virtual Machine Manager (VMM) [45]. A hypervisor is responsible for instantiating and managing virtual instances on the physical hardware. These virtual instances, or Virtual Machines (VMs), represent the visible offering to the end customer.

The hypervisor can either reside directly on the hardware (type-1 or *bare-metal*), or on top of an existing operating system (type-2 or *hosted*) [46]. In the case of type-2 hypervisors, an additional OS level is required between the hardware and the hypervisor layers.

Type-1 hypervisors are generally faster, as they have direct access to underlying physical resources, and more secure and stable, because the presence of an OS in type-2 hypervisors increases the attack surface and the possibility of OS-level performance issues. However, type-1 hypervisors do not offer much flexibility and customization, as they require a VMM compiled specifically for the underlying type of physical machine. Because the type-1 VMM software is highly specific, it is typically less supported and new features are rarely presented. Examples of type-1 hypervisors include KVM [45], Microsoft Hyper-V [47] and Xen [48].

On the other hand, type-2 hypervisors are more affordable and easier to set up and manage. Because the OS typically support a variety of hardware configurations, type-2 hypervisors running on top of an OS can also support different hardware configurations. Type-2 hypervisors, however, introduce an additional computing layer which may cause additional failures. Examples of type-2 hypervisors are Oracle VirtualBox [49] and Parallels Desktop [50].

Regardless of hypervisor types, VMs provide functionalities fully equivalent to physical servers, as their internal applications have full access to core hardware (CPU, memory) and peripheral devices (storage, network interfaces), as well as guest OS functionalities (filesystem, process space). However, because each VM provides its own OS, they result in additional overhead in the creation and execution of workloads.

To this end, OS-level virtualization (or *containerization*) has been introduced. Containers are isolated user-space instances within the same OS. They provide the same functionalities of OS-residing VMs, without the overhead of deploying individual OSes in each virtual instance. All other features, e.g., networking, filesystem, CPU, or process space, remain isolated across virtual instances, so that the containerization effectively provides isolation of processes within the same OS. Other advantages of containers include fast start-up, more efficient storage and portability. Examples of OS-level virtualization technologies include Docker [51] and containerd [52].

Other Cloud-enabling technologies

Orchestration tools, such as Kubernetes [53] or Docker Compose [51], enable the automated deployment, scaling and management of containerized applications. They help streamline complex application architectures in cloud environments.

Automation and DevOps tools facilitate continuous integration and delivery of new software functionalities, while allowing for rapid and consistent application development.

Software Defined Networking (SDN) virtualizes the network infrastructure by making it programmable and easily manageable. It improves network flexibility and adaptability without modifying the physical cloud infrastructure.

A *microservice architecture* breaks down an application into smaller, loosely coupled services which implement primitive functionalities (such as a database, a frontend service, or a messaging system). Each microservice can be developed, deployed and scaled independently of the others. A microservice architecture is beneficial as it is scalable, it emphasizes agility and provides fault isolation across components.

2.1.3 Cloud Architecture

A cloud computing system may be modeled as a layered stack of functional components [54], vertically administered by O&M.

A traditional cloud system model is composed of three layers: the infrastructure layer, the platform layer, and the software layer. Figure 2.2 exemplifies the three-layer stack model, compared with traditional, on-premise computing. The next sections describe these three layers in detail.

Infrastructure Layer

The infrastructure layer is responsible for processing, storage, network, and all other physical resources necessary for computing [39, 55, 56]. It is also responsible for dividing these physical resources into separate resource pools and assign them to different customers.

In the Infrastructure-as-a-Service (IaaS) cloud model, the infrastructural resources are offered directly to the customer as a service. The cloud resource offering must be able to scale up fast and effectively to meet the variable computing demand of the customers while maintaining operational expenditure low. Cloud infrastructures rely on virtualization to achieve such economy of scale and isolate sensitive content between customers, as described more in detail in Section 2.1.2.

Tools for infrastructure-level servicing include KVM [45], Openstack [57], Docker [51], Kubernetes [53].

Platform Layer

The platform layer is responsible for providing software resources to build and deploy applications. It is composed of the OS and all the application frameworks necessary to enable development and execution of software programs (so-called *middleware*), including libraries, databases, runtime environments, messaging systems, and build tools.

Platform-as-a-Service (PaaS) provides this integrated environment, where it is possible to develop, test, run, and host applications [40], directly to end customers as a service. While the set of tools made available to users is standardized and comprehensive, platform environments of different customers are isolated, so that security, privacy and flexibility requirements are satisfied.

This highly interconnected and intricate environment can be the source of several failures, caused by incompatibility or misconfiguration between different components. Therefore, platform management requires constant supervision to ensure that the different application elements are correctly integrated and produce the expected functionality. To this end, O&M and IaaS customers utilize Command-Line Interface (CLI) tools to maintain and operate platform-level services and the application deployments dependent upon them.

Software Layer

The software layer provides the customer-facing software functionalities, built on top of infrastructure and platform layers.

Software-as-a-Service (SaaS) provides (in addition to platform and infrastructure), development and deployment of software applications. While PaaS and IaaS offerings are intended for developers, SaaS offers a complete software products to end consumers.

The software layer highly relies on technologies such as HTTP backend servers, microservice architectures, authentication systems, and Application Programming Interfaces (APIs).

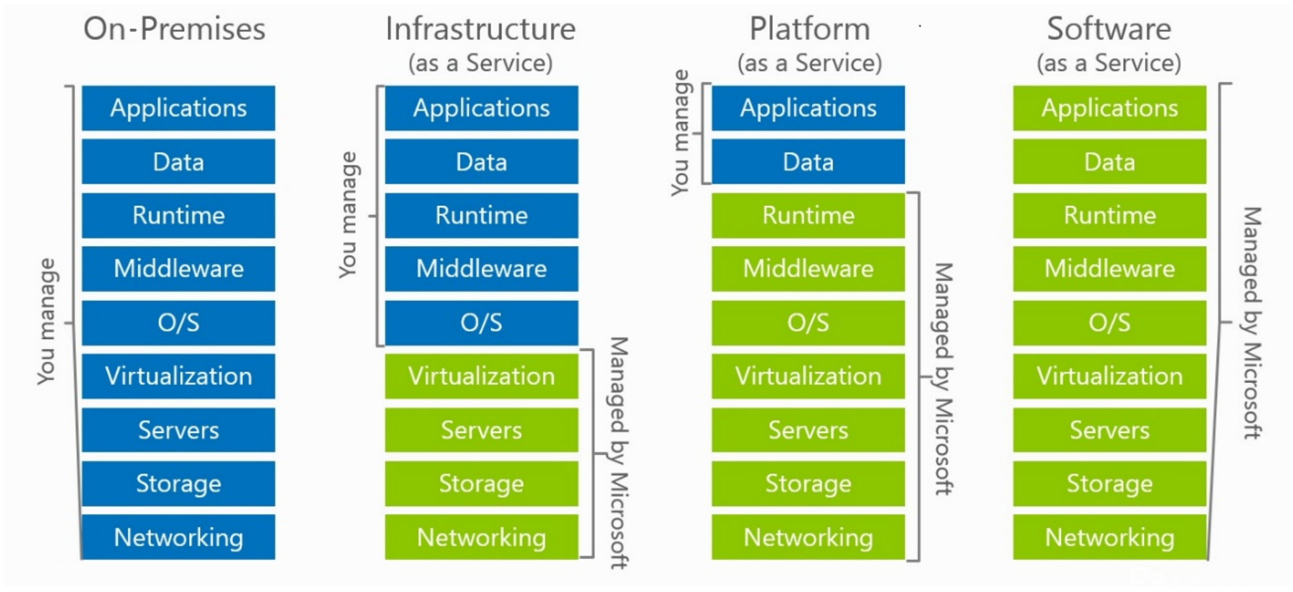


Figure 2.2 Cloud Computing Stack Model for Microsoft Azure [58, 59], composed of three layers (infrastructure, platform, and software), with corresponding resources functionalities provided by each layer.

2.1.4 Cloud Computing Models

This section presents different models for cloud computing. These models include service models, which associate cloud layers to product offerings; ownership models, which describe different relationship between cloud provider and customers; and pricing models, which determine how the cost for cloud computing services is calculated.

Service Models

Service models may be derived from the cloud computing stack model described above (Section 2.1.3). By associating each layer to a corresponding service, cloud offerings can be differentiated as:

- **Infrastructure-as-a-Service (IaaS)**, i.e., "the capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications" [39].

  An IaaS provider supplies these infrastructural resources, to enable the IaaS customer to execute their workload without managing their own hardware fleet. Moreover, the customers are responsible only for the facility and provisioning costs corresponding to the infrastructural resources used. This allows to reduce costs compared to on-premise infrastructure and other cloud offerings.

  Differently from other models, where the OS, middleware and runtime are also managed by the cloud provider, in the IaaS model the customer has direct control over such aspects. As in other cloud models, the customer is relieved from the management of physical resources and the necessity to ensure a scalable and secure computing environment, as these needs are handled by the cloud provider. These needs include (but are not limited to) access to high-speed, fault-tolerant Internet access, efficient power and cooling environment, safety systems and on-call O&M staff [55].

  Bare-metal services provide hardware access at its most basic, i.e., the cloud provider rents the raw, physical server directly [40]. These services are more similar to traditional datacenter or 'hosting' services, compared to cloud computing offerings. For the majority of potential cloud consumers, there is a desire to move away from infrastructure detail and progress upwards in the cloud stack model.

- **Platform-as-a-Service (PaaS)**, i.e., "the capability provided to the consumer [...] to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider" [39]. In PaaS, the hardware and a software platform are provided and managed by an outside cloud service provider [60]. The user handles the applications running on top of the platform and the data on which the app relies.

  PaaS gives users a shared platform for application development and management, removing the necessity to build and maintain the runtime environment associated with the process. This includes, e.g., software development tools, runtime environments, libraries, and middleware such as databases, Business Intelligence (BI) services, CLI interfaces, or message passing systems. These tools and the application runtime are accessible to users typically over a web browser interface, where the customer can upload, modify and execute their data and software.

  Examples of PaaS offerings include Google App Engine [61] and Heroku [62].

- **Software-as-a-Service (SaaS)**, i.e., "the capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface" [39]. SaaS delivers a software application directly to its users [60]. The cloud service provider manages the development and maintenance of the application via software updates, bug fixes, and by providing adequate resources to execute it. Users are completely relieved from all infrastructural, development and deployment burdens, and they are only responsible for operating and configuring the software according to its functional capabilities. Applications of this level typically include web browser applications, mobile apps, and pay-per-use software.

Serverless computing [63] abstracts server management and infrastructure and runtime management of software away from users. A fundamental concept of serverless computing is **Function-as-a-Service (FaaS)**, which allows developers to build, execute and manage applications as functions, without having to maintain their own infrastructure [64]. FaaS is typically implemented through lightweight, stateless containers which implement the desired functions and manage server-side logic and state.

Ownership Models

Different ownership models for the cloud exist, which vary in terms of cost, security, performance, and flexibility [39, 40, 60].

A *public cloud* is available to the general public and it is typically managed by a single organization (the *cloud provider*), such as a business or an academic or governmental institution. Public cloud providers own and manage the cloud infrastructure. This enables reduction of costs, but may cause performance interference and higher exposure to cybersecurity attacks. Examples of public cloud include Amazon Web Services (AWS) [65], Google Cloud Platform [61], Microsoft Azure [58], and IBM Cloud [66].

A *private cloud* serves single organizations exclusively. Cloud resources may be located on- or off-premise and could be owned; they could be managed directly by the consuming organization or by a third party, in case the organization decides to adopt the infrastructure cost-saving potential of a virtualized architecture on top of existing hardware. Another reason to adopt a private cloud is the desire to prevent confidential data to be held alongside potential competitors or political adversaries. Private clouds, therefore, provide a higher level of security and isolation at a higher deployment and maintenance cost. Public cloud providers such as AWS, Cisco [67], Google, IBM, Red Hat [60] also provide private cloud solutions.

A *community cloud* is a cloud computing model composed of a number of parties where resources are shared to contribute to a common interest or cause. This is frequently used in research organizations to conduct large-scale experiments or by private parties interested in contributing in a cause (e.g., extraterrestrial life search [40] or cryptocurrency mining).

Hybrid clouds are composed of more than one type of cloud infrastructures (public, private, and community) [39]. Multiple computing environments may be connected through Local Area Networks (LANs), Wide

Area Networks (WANs), Virtual Private Networks (VPNs), and/or APIs to create a apparently unified IT environment [60]. Therefore, an IT system becomes a hybrid cloud when applications can freely move in and out of separate environments. This allows to serve critical or confidential workloads on-premise or in reserved areas, while accommodating general or additional workloads on public cloud infrastructures. A *multicloud* deployment involves more than one public cloud provider. It enables redundancy and reduces service impact in case of large provider outages.

Pricing Models

Public cloud providers offer several pricing models, which can be selected based on the different needs of customers [68].

A *pay-as-you-go model* bills customers for cloud services based on their actual usage. This is proportional to the utilization of power, storage, networking, and computing resources used. The price of a resource is typically defined per unit of time (e.g., $0.10/hour). Reserved but under-utilized resources, such as idle VMs and hard drives, are billed (possibly to a reduced price) for occupying the cloud resources. A related concept is *pay-for-what-you-use*, which computes the total service cost based on the number of requests issued, as in traditional API pricing models. This pay-as-you model is particularly applicable to invocation-based cloud products, such as FaaS offerings and the above-mentioned APIs.

Fixed or prepaid subscriptions allow customers to pay for services upfront. Subscription prices are determined beforehand for a specific time period and do not depend on the actual usage. This includes e.g., software licenses or high-volume PaaS and SaaS computing offerings for large organizations. Subscription-based models may allow users to retain reserved instances for long periods of time (i.e., years), which may help ensure reduced costs in the case of predictable, low-level workloads.

Spot pricing determines the price of a cloud computing product based on the current demand. In periods of low computing demand, when cloud providers wish to sell off spare capacity, large discounts may be offered to customers, which can drastically reduce cost (up to -90%). Spot instances, however, may be interrupted at short notice and therefore workloads must fulfill fault tolerance or stateless principles.

2.2 Cloud Operations and Maintenance (O&M)

This section presents a background on O&M in the cloud computing environment. Quality of Service (QoS) and service requirements are discussed, and how failures impact on achieving such requirements. Then, O&M is introduced, including its techniques and tools.

2.2.1 Quality of Service in the Cloud

Because cloud providers charge users based on the utilization of their service, both users and cloud administrators are interested in ensuring that the paid service provides the expected quality to the end user.

QoS measures the ability of a service to meet certain quality requirements for aspects such as performance, availability, reliability, or cost [69]. Several metrics may be considered to evaluate QoS, including rejection rate, Mean Time Between Failures (MTBF), response time, throughput, economic cost, or energy consumption. Then, service providers and customers have to negotiate a SLA that allows them to formally specify the QoS and agree on the requirements.

A Service Level Agreeement (SLA) contains the details of the contract made between the cloud service provider and the customer. The terms regarding quality and scope of the service may also include a resource provisioning arrangement, which specifies allocable resource quantities. Both parties must understand the constraints and agree upon the limits of resource availability [70].

Cloud providers are therefore responsible to guarantee high-level availability and performance in their cloud computing services [71], hosted in large datacenters, to the standards defined in the SLA. Common objectives include a maximum allowed downtime window per year (defined in terms of availability, e.g., 99.999%, or 'five nines'), a maximum latency, a minimum throughput and so on. To this end, O&M activities are devoted to ensuring the correct runtime of the cloud environment, and the fast recovery in case of incidents.

SLAs may be difficult to achieve, as cloud services are not immune to failures. Because of the large scale of devices and services being executed, failures are, albeit rare, inevitable. To this end, specialized human operators are necessary to resolve errors during the operations of large-scale computing services.

2.2.2 Operations and Maintenance (O&M)

O&M, or Operations, Administration, and Maintenance (OAM) [72], is the set of all processes and tools used inside an organization for the purpose of fault detection, isolation, and performance measurement.

Cloud O&M (or *Cloud Operations*) teams are responsible for managing the cloud environment and mitigating the impact of cloud failures. Their responsibilities include:

- *real-time service monitoring*, i.e., the continuous tracking of the operational status and health of the cloud infrastructure, services, and applications. Advanced monitoring tools may be employed to report unusual or under-performing behavior, based on anomaly detection algorithms;
- *incident response*, i.e., the rapid on-site intervention when failures occur. This includes gathering sufficient evidence of the incident, and diagnosing the root cause of the failure, also known as Root Cause Analysis (RCA), and the implementation of remediation actions;
- *predictive analytics*, i.e., the study of historical data and failure patterns, to forecast future potential failures;
- *data backup and restoration*, i.e., the implementation of robust data redundancy strategies to safeguard against data loss. In case of failures, O&M operators initiate restoration to ensure data integrity;
- *coordination and communication*, i.e., O&M teams coordinate efforts to address reliability and availability issues and must keep customers informed about the status of failures and ongoing recovery efforts.

2.2.3 Cloud Failures

Cloud failures can stem from a variety of sources, including hardware malfunctioning, software bugs, human error, network issues, or external factors such as cyber-attacks and natural disasters.

The consequences of cloud failures are far-reaching. Downtime leads to loss of revenue, customer dissatisfaction, and damage to the company's reputation.

In the discussion of this dissertation, a variety of error-related terms such as fault, failure and root cause are used. It is therefore important to clarify the meaning of such terminology. To this end, the convention of Salfner et al. [73] is here adopted for the characterization of failures and related terms. According to this convention,

- *errors* are deviations from the expected system state. In particular, "an error is the part of the total state of the system that may lead to its subsequent service failure" [73];
- *failures* are manifestations of undesired deviations during the delivery of a service;
- *faults* (or *root causes*) are the primary causes of undesired behavior (i.e., the errors).

Moreover, failures may be divided into *soft* (or *transient*) failures, when they cause a temporary deviation from the expected behavior, to then resume normal operation; and *hard* failures, which lead to a permanent state of unexpected behavior (such as broken component or a crashed software instance).

Hardware and software faults can affect Internet services in varying degrees, resulting in different service-level failure modes [12]. Barroso et al. [12] classify service-level failures into the following categories, in increasing level of severity:

- *corrupted*, when committed data are impossible to regenerate, lost, or corrupted;
- *unreachable*, when the service is down or otherwise unreachable by users;
- *degraded*, when the service is available but in some degraded mode;
- *masked*, when failures occur but are completely hidden from users by fault-tolerant software and hardware mechanisms;

Failure Management (FM) is the union of all techniques to deal with failures, either in anticipation (*proactive* FM) or in response to failure (*reactive* FM).

A *repair* is a recovery action taken to remove the failure and return to normal operations. Failures requiring repairs are typically tracked inside a *ticket* system, where O&M operators or even users may open tickets to track the status of repair of a particular issue. Individual tickets typically correspond to unique problems with individual resolution actions, such as component replacements, service reboots, or code fixes.

2.2.4 Cloud Monitoring

Monitoring is the act of collecting, processing, aggregating, and displaying real-time quantitative data about the state of a distributed system [74]. The system state is the composition of a very large and diverse set of resource states, including servers, storage, virtual instances, network infrastructures, cooling and heating systems, applications, and services.

Monitoring tools are used to provide high-level information in a simple and graphical representation by means of graphs, statistical tools, reports, and dashboards. Monitoring quantities include resource usage (in particular over- and under-utilization), workload, performance bottlenecks, SLA status, costs, and malicious user activities. A typical monitoring tool is a monitoring dashboard, a graphical interface tools which allows to visualize important monitoring information effectively, by means of statistics and graphs.

The objectives of monitoring are the detection of failures and performance issues, the verification of SLA, the study of long-term trends, troubleshooting, and system breach analysis. Effective monitoring allows gathering large quantities of symptoms for cloud-related problems, but it does not necessarily discover the root problem itself. The monitoring of data is required so that an expert can understand the root causes of the underlying issue. Because relying on on-call personnel is typically expensive and time-consuming, an effective monitoring infrastructure alerts about problems only when it is clear that human intervention is required. Over-reporting may hinder the ability of operators to respond to important events, either by reporting false positive events, which occupy operator time, or by hindering true positive, which require more critical attention. Under-reporting, however, reduces the detection and responsiveness to failures.

Two are the main types of monitoring: *white-box monitoring* and *black-box monitoring*,

White-box monitoring is based on metrics exposed directly from the system internals. Its validity depends on the ability to inspect the internal components of the system. It is more suitable to detect imminent failures and problems suppressed by mitigation strategies, which may re-appear frequently while hidden and cause performance degradation. White-box monitoring is also important for telemetry purposes, e.g., understanding the normal behavior of the system. Such information may also prove fundamental for distinguishing abnormal behavior during failures and incidents.

White-box monitoring data is very useful for modeling failure characteristics and anticipating imminent failures. The clear disadvantage is the high cost of instrumentation and collection, as well as privacy and security concerns connected to the very granular level of information collected (e.g., host- or customer-level).

Black-box monitoring traces external behavior as an end user would see it. As the internal information about the system is hidden, a black-box monitoring tool can only detect failures when they start to affect the end customers. This makes it ideal for operator alerting, as it discourages the discovery of non-critical problems, and at the same time provides higher-level information, such as service reachability and latency, which are directly connected to SLA requirements. Because black-box monitoring tracks the consequences of failures, it is not useful for proactive anticipation of problems. Google [74] defines four “golden signals” to black-box monitor:

- *latency*, i.e., the time necessary to service a request. It should be grouped by request status, i.e., the latency of failed requests and the latency of successful requests. This allows distinguishing slow successful requests and estimating the frequency and latency of failed requests;
- *traffic*, i.e., The measure of demand placed on a system. It can be measured in requests per unit of time, number of live connections, of total I/O throughput;
- *errors*, e.g., the number of failed requests. Particular care should be taken to include *silent errors* as well (errors which are implicitly correct, e.g., 200 return code HTTP requests with invalid response payload);
- *saturation*, i.e., the fraction of resources utilized by the system, focusing on the most constrained resources (e.g., storage or memory).

2.2.5 White-Box Monitoring Data

In the cloud context, a large set of heterogeneous items must be effectively monitored to ensure high issue visibility, each with different critical aspects to track (given below in parentheses).

At the infrastructure level, these include hardware components (storage drives, network devices, power supply units, memory chips), hosts (availability, resource usage, critical errors, reboots), virtualization software (VMM state, VM state as for a physical host).

At the platform level, OS (system logs and system call tracing), middleware services (messaging services, queues, databases), runtime environments (Java Runtime Environment (JRE), Python *venvs*, Docker logs) and overall workload (commands, queries, system calls per second) must be monitored.

At the software level, user accesses (logins, operations), application logs, access logs, service KPIs and performance metrics may be used.

On a general O&M level, SLA reports must be monitored to verify if expected objectives are guaranteed; failure tickets must be routed, handled and closed; costs must be kept under control.

Different terms related to monitoring data are often encountered. These terms may have an ambiguous meaning depending on the context (such as logs or traces). To be consistent in the discussion, the following convention is used throughout the dissertation:

- *source code* represents any unit of software source code used as input to a prediction system, independently of the form and extension (e.g., function, file, module, class, etc.);
- *testing resources* comprise tools used to perform in- and post-release software debugging, in particular unit test suites, execution profiles or run description reports;
- *system metrics* measure various numerical quantities at the hardware, OS, software and environment level, describing resource utilization and the overall process state of the system;
- *KPIs* provide information about the status of services and the associated requirements that need to be met during runtime operations. They quantitatively measure the quality of served requests with parameters such as latency, uptime, failure rate, availability, etc.;
- *network traffic* is the collection of network packets exchanged over the Internet by different hosts. It includes the payload and control information such as ports, addresses, protocol standards and other parameters;
- *topology* is spatial information describing the relations between components inside a working system;
- *incident reports* or *tickets* are collected with the help of the service desk and internal problem management systems to identify common problems and facilitate resolution. Usually, they describe the problem with text and categorical attributes, and they may need to be associated with a resolution team or routing sequence;

- *event logs* (or simply *logs*) are collections of human-interpretable printing statements describing software events occurring in runtime operations. They are typically stored as independent files and log entries (i.e., lines) are associated with a predefined format (or *log key*);
- *execution (distributed) traces* are hierarchical descriptions of the modules and services invoked to satisfy a user request. They are usually annotated with the service name or category and the time duration of each module (called *span*).

2.2.6 Automated O&M and AIOps

Because of the high quantity of monitoring data and diversity of operations to perform, in modern cloud environments a significant fraction of O&M elementary tasks is automated through CLI tools, scripts, and libraries.

These automated tools may be used for automated alerting of specific critical variables, such as network latency or CPU utilization, so that when a specific threshold is reached, operators are informed about a potential problem. This alerting, however, still requires operator to monitor alerts, investigate the discovered issue manually and take a corresponding repair action in response to the problem.

Automated scripts may perform a recovery routine, e.g., a reboot, a host migration, or a service restart. However, such scripts typically hard-code the necessary solution and require operator to execute them, as they cannot be triggered automatically or they require high privilege.

To this end, AI for IT Operations (AIOps) has been introduced to enhance the introspection and generalization abilities of O&M software, and perform the detection-to-recovery cycle end-to-end.

AIOps is the application of advanced analytical technologies (ML, Big Data, Data Mining), and more broadly AI, for the support of IT operations [14] connected to O&M.

To this end, effective AIOps relieves the burden of operators from performing manual and repetitive jobs, such as monitoring key metrics, initiate remediation procedures, locate component faults and so on.

The main tasks in AIOps include anomaly detection, i.e., the identification of deviations from normal behavior, an important task for guaranteeing SLA and detecting failures; root cause analysis, i.e., the study of what caused a particular error or performance problem; and failure prediction, i.e., the anticipation of failure through forecasting and classification models.

An overview of the main topics and challenges in AIOps, as identified from the systematic literature review [38, 75], is presented in Sections 3.4.2 and 3.4.3.

Among all O&M data sources listed in Section 2.2.5, most commonly utilized metrics in AIOps include system metrics, typically handled in the form of real-valued time series; logs, handled either as collections of structured data, to form tabular datasets, or as sequences of composite data (i.e., natural language text augmented with categorical, numerical and temporal data); and traces, treated as hierarchical, tree-like structures or as sequences of actions.

2.3 Artificial Intelligence (AI)

This section presents background knowledge on the AI and ML fields, necessary to understand the technical details behind the related work and the proposed AIOps solutions.

2.3.1 Foundations of AI

AI is the study of building machines that think and act (at least) as rationally as humans [3]. This includes abstract abilities such as reasoning, problem-solving, memorization, and learning.

Because of the complexity of constructing a program with such characteristics applicable to all domains (a so-called "general intelligence" system, AGI), AI has been historically divided into several sub-problems, such as perception (vision, speech, etc.), learning, and reasoning.

As described in the introduction (Section 1.1.2), AI algorithms have a large number of applications in numerous fields, including Computer Vision, Natural Language Processing (NLP), bioinformatics, system control, advertising, and more. Because of its high applicability, many have raised questions related to security risks, biases, and copyright violations induced by the use of AI.

AI relies on a wide range of mathematical and engineering tools [3], including search and optimization methods, stochastic modeling, logic; specialized hardware, such as Tensor Processing Unit (TPU) and GPU devices; specialized software languages, such as Lisp [76], Prolog [77], and libraries, such at PyTorch [78], scikit-learn [79], Natural Language ToolKit (NLTK) [80], and Tensorflow [81].

With respect to algorithms, two main categories of approaches may be defined: search and optimization approaches, which can be defined as "Traditional AI" and are discussed in Section 2.3.2, and statistical learning or ML approaches, discussed in Section 2.3.3.

2.3.2 Traditional AI

Traditional AI approaches rely on symbolic and goal-based methods to solve high-level problems and achieve some degree of generalization, as expected from an intelligent system. They may be divided into logic-based and search-based approaches [3].

Rules of formal logic can be used to derive propositions and facts from given assumptions, a task known as logical inference. Logic approaches include propositional, first-order and fuzzy logic. Propositional and first-order logic are both forms to express formal logic and deterministic reasoning. They may be applied for causal inference problems, such as automated theorem proving. Fuzzy logic expresses intermediate degrees of truth and can be used to model and treat uncertainty during inference. Similar approaches related to probabilistic reasoning include Bayesian Networks and Hidden Markov Models (HMMs), which model the relation between observable knowledge and uncertainty states using the laws of probability.

Search approaches explore solutions in large search spaces to find optimal solutions or to satisfy constraint problems. These include both mathematical optimization problems, whose purpose is to find a numeric solution to a mathematically-defined goal, and path discovery problems, which describe a list of actions or directions to take to reach a specific goal. Optimization can be approached using evolutionary computation (such as genetic algorithms) or other swarm intelligence algorithms (e.g., ant colony optimization or particle swarm optimization). Path discovery includes graph-based approaches such BFS and DFS [3], which may potentially include a heuristic function to estimate proximity to the end goal (e.g., A* search). Path discovery algorithms are used in navigation, planning, and game applications.

2.3.3 Machine Learning

Machine Learning (ML) is the study of programs that can improve performance on a predefined task automatically, by learning from the data [3, 82, 83]. Such programs typically process the data based on statistical modeling assumptions, and are for this reason also called *ML models*.

Compared to Data Mining, a related topic in statistical learning, ML focuses more on the prediction outcome of the model, while Data Mining focuses on the insights that can be extracted from the data. Because predictions typically correspond to the next actions to take based on the analysis of input data, ML is more actionable and applicable for automated interaction with the environment (e.g., the cloud).

Training and Inference

During the normal execution of a ML algorithm, called *inference*, the ML algorithm is given some input D-dimensional data $\mathbf{x} = \{x_1, x_2, \dots, x_D\} \in \mathbb{R}^D$ so that it can produce $y(\mathbf{x})$, the predicted solution to the task, which may correspond to a score (e.g., for house price or movie rating prediction tasks) or a category index (like in the case of image classification or segmentation) assigned to the sample.

A preliminary learning phase called *training* may be performed before inference. During training, a large quantity of data samples $\mathbf{x}^{(\mathrm{i})}$ is provided to the algorithm in order to learn how to map the inputs to the prediction value y. The mapping is adjusted by updating the parameters of the model, in order to reach a specific mathematical objective (called *error function* or *loss function*), e.g., minimization of total prediction error. The input data samples are usually stored together to form a *dataset* $\mathbf{X} = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(\mathrm{i})}, \dots, \mathbf{x}^{(\mathrm{N})}\}^T \in \mathbb{R}^{N \times D}$.

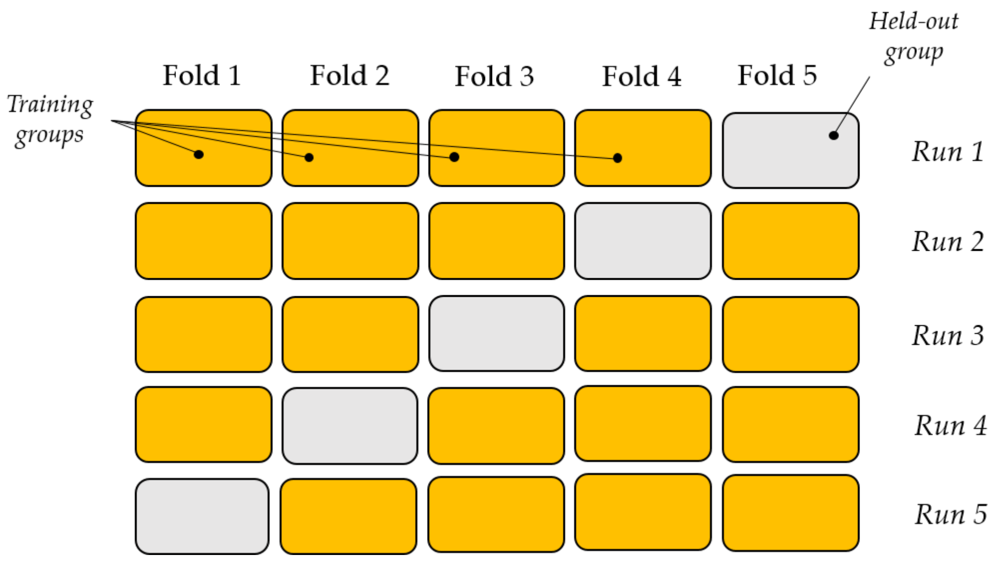


Figure 2.3 Example of k-fold cross-validation ($k = 5$). The original training set is divided into k folds. The ML model is trained k times, holding out a different fold for each run. The algorithm is evaluated on the held-out fold in each run, and the evaluation results are averaged out at the end of the process.

For the purpose of training and evaluating the learning ability of a ML model, the complete dataset $\mathbf{X}$ is preliminarily divided into three disjoint subsets $\mathbf{X}_{\mathbf{train}}, \mathbf{X}_{\mathbf{dev}}, \mathbf{X}_{\mathbf{test}}$. The *training set* $\mathbf{X}_{\mathbf{train}}$ is used for the training phase, during which the total prediction error of the training set is minimized. The *validation set* or (*development set*) $\mathbf{X}_{\mathbf{dev}}$ is used to verify the generalization ability of the ML model to unseen data and to tune construction variables of the learning algorithm, the so-called *hyperparameters*. After this optimization has taken place, the data from the *test set* $\mathbf{X}_{\mathbf{test}}$ is used to perform a final evaluation report. This corresponds to the expected performance during inference. The presence of the test set is necessary to evaluate the final algorithm performance, as the choice of optimal hyperparameters may be specific to the validation set.

A model which performs its task well on all three datasets is said to *generalize* to the task [83]. If a model performs well on the training set, but not on the validation set (or performs well during validation, but not during the test evaluation), is said to be *overfitting* to the training set (or to the validation set). If a model cannot perform well on the training set, it is said to be *underfitting*, i.e., it cannot reduce its error function sufficiently and cannot perform the task even on training data. Overfitting and underfitting are influenced by many factors, including model capacity, i.e., the ability of a model to fit a variety of functions; and *regularization*, i.e., any modification to the algorithm in order to reduce its generalization error. Examples of regularization techniques include *ridge* and *lasso regression* [82], which are forms of penalty introduced in the error function, to incentivize the construction of simpler mappings inside the model (according to Occam's razor).

In cases where training data samples are limited, an alternative to the validation set called *cross-validation* is possible [82]. In k-fold cross-validation, the training set is divided into k disjoint parts, so-called *folds*, and the model is trained k times using $k - 1$ folds as training set, and evaluated on the remaining fold, to obtain k evaluation results which are averaged out. An example of k-fold cross-validation is shown in Figure 2.3. With this technique, it is possible to use the entire training set for validation purposes, without preliminarily setting validation data aside. The disadvantage of cross-validation is the k-fold runtime required to obtain the different fold evaluations, which may prove unfeasible in the case of computationally expensive models (e.g., neural networks).

Machine Learning Approaches

ML approaches may be divided on the basis of the external supervision that is provided during the training phase. *Supervised Learning* involves all ML methods where there is a direct supervision on the desired task, i.e., each data sample $\mathbf{x}^{(i)}$ is annotated with a target label t_i which corresponds to the correct answer for the task (e.g., a prediction score, or a class). Typical supervised learning tasks include *classification*, i.e., the estimation of a category; and *regression*, i.e., the estimation of a scalar value (e.g., a rating or a metric).

Unsupervised Learning involves all ML methods where there is no clear guidance or label for the specific learned task. In such scenarios, the key to the task lies directly in the input data. Typical unsupervised tasks include clustering, i.e., the assignment of samples to groups of similar instances; dimensionality reduction, i.e., the process of reduction of input dimensions D while preserving the information in the data.

Besides supervised and unsupervised learning, intermediate and hybrid possibilities exist. *Reinforcement Learning (RL)* teaches software agents to take optimal action in an explorable environment. The agent is free to act and experiment within the environment and the supervision is provided in the form of a reward based on the actions taken. *Semi-supervised Learning* corresponds to learning tasks where only part of the training dataset is labeled. *Self-supervised Learning* corresponds to tasks where part of the input data is used as supervision to train a supervised learning model.

Classical ML encompasses all ML methods that are not covered by the most recent wave of novel neural-based methods, which are discussed in the deep learning section below.

The discussion here is limited to classical ML methods that are used or mentioned in this dissertation: linear models, pattern mining models, and other classical ML models (similarity-based approaches, tree-based approaches, and kernel-based approaches).

Linear Models

Linear models attempt to map inputs $\mathbf{x^{(i)}}$ to labeled targets t_i using a linear function. In the case of a real target $t_i \in \mathbb{R}$, this task is called *linear regression*, i.e.,

$$t_i \approx y_i(\mathbf{x^{(i)}}) = \mathbf{w} \cdot \mathbf{x^{(i)}}, \tag{2.1}$$

where $\mathbf{w} \in \mathbb{R}^D$ are the linear parameters (or *weights*) of the model. The approach can be extended to model non-linear dependencies between input and targets, to obtain a so-called *generalized linear model*:

$$t_i \approx y_i(\mathbf{x^{(i)}}) = \mathbf{w} \cdot \Phi(\mathbf{x^{(i)}}), \tag{2.2}$$

where $\Phi : \mathbb{R}^{D_1} \mapsto \mathbb{R}^{D_2}$ is an arbitrary (possibly non-linear) feature map to apply to the input data. Feature maps are used to encode additional information, such as domain knowledge, in the model. The process of selecting effective input features for a ML model is called *feature engineering* [84].

Linear models may also be applied for classification. In such case, a *logistic regression* model estimates the probabilities of belonging to a specific class k by applying an additional normalization function to the output, e.g., the logistic (*sigmoid*) function σ for binary classification, or the *softmax* function for multi-class classification , which condenses the C raw output score(s) z produced by a linear model to $0{-}1$ class probability range(s) p_k:

$$p = \sigma(z) = \frac{1}{1+\exp(-z)} \qquad \text{or} \qquad p_k = \text{softmax}(\mathbf{z})_k = \frac{\exp(z_k)}{\sum_{j=0}^{C}\exp(z_j)} \text{ for } k = 1, 2, \ldots, C \tag{2.3}$$

During the training of both regression and classification models, the linear weights of the model $\mathbf{w}$ are adjusted to minimize the difference between predictions and target values. The error is typically minimized iteratively using first-order gradient-based optimization methods, such as gradient descent [82, 85]

$$\mathbf{w_{t+1}} \leftarrow \mathbf{w_t} - \eta \cdot \frac{\nabla L}{\partial \mathbf{w}} \tag{2.4}$$

where L is the error function to minimize, $\mathbf{w_t}$ are the weights at time step t, and η is the *learning rate*, a tunable hyperparameter. Stochastic Gradient Descent (SGD) allows to converge to a local minimum in the high-dimensional landscape of the error function. For regression problems, the Mean Squared Error (MSE) is frequently used as the error function:

$$L_{MSE} = \frac{1}{N} \sum_{i}^{N} \left(y(\mathbf{x}^{(\mathrm{i})}) - t_i \right)^2 \tag{2.5}$$

while for classification problems, the *cross-entropy* loss function is typically used:

$$L_{CE} = -\frac{1}{N} \sum_{i=1}^{N} t_i \cdot \log p_{t_i} \tag{2.6}$$

Pattern Mining Models

Pattern mining is a group of unsupervised ML methods that are used to identify patterns and rules present within the data [86]. Here two main algorithms are discussed, Association Rule Mining (ARM) and Sequential Pattern Mining (SPM), as they will be used as a tool to infer causality in different AIOps methods proposed.

ARM, or *association rule learning*, is a classical ML method for discovering correlations in structured item data [87]. The objective of ARM is the identification of interesting relations of the form $X \implies Y$, where X and Y are elements observed inside a large database of items, called *transactional database* (as its constituents are called *transactions*); the implication symbol means that there is a certain degree of association between observing X first and then Y after.

Here the theoretical ARM notions are formalized as they were originally proposed [87–90]. A *database* $\mathbb{D} = \{t_1, t_2, t_3, \dots, t_n\}$ is a multiset of n transactions $t_i \subseteq \mathbb{B} = \{b_1, b_2, \dots, b_k\}$, where the *item base* $\mathbb{B}$ is the union of all items b_i present in $\mathbb{D}$,

$$\mathbb{B} = \bigcup_{t \in \mathbb{D}} t, \qquad k = |\mathbb{B}| \tag{2.7}$$

An *itemset* is any subset $X \subseteq B$. A *pattern* is an itemset contained in at least one of the transactions of $\mathbb{D}$. The main goal of ARM, i.e., to find rules of the form $X \implies Y$, can be formalized as follows: given a predefined pattern $Y \subseteq \mathbb{B}$, find all patterns $X \subseteq \mathbb{B}$ such that $Y \not\subseteq X$ and $X \subseteq t_i \implies P(Y \subseteq t_i) \geq 1 - \epsilon$ for a sufficiently small ϵ, i.e., the observation of X is a good predictor of the presence of Y.

A common use of ARM is *market basket analysis*, a tool used by retailers to estimate products that are likely to be purchased together. In this context, items correspond to purchasable products (e.g., `‘onion’`) and transactions correspond to lists of products bought together (e.g., `{‘onion’, ‘milk’, ‘butter’}`). The objective is to evaluate if, e.g., `{'butter'}` $\implies$ `{‘milk’}` is a valid association, given the evidence provided by transactions.

In RCA based on structured logs (Section 6.3.4), a structured log is treated as a transactional database: individual log entries are transactions, and each log entry is a set of key-value pairs, where the key-value-pairs are considered as the items composing that transaction. X and Y are sets of key-value pairs present in log entries (e.g., `{ip=172.146.XXX.XXX, port=YYYY}`); in particular, Y is a set of key-value pairs indicating a failure state or an event of interest (e.g., {`status=FAIL, service=FTP`}). An example of ARM concepts in the RCA scenario can be found in Figure 6.2 (Section 6.2.1).

ARM is composed of two phases: frequent pattern mining and rule selection.

Frequent pattern mining is the process of discovering the most common patterns in a database. Several frequent pattern mining algorithms exist, most notably Apriori and FP-Growth [87, 89]. These approaches iteratively construct the set of the most frequent patterns observed in $\mathbb{D}$.

Apriori is one of the earliest approaches to frequent pattern mining [87]. In Apriori, an exhaustive set of possible sub-patterns (called *candidate set*) is incrementally constructed until a predefined minimum frequency (or *support*) is reached. The FP-Growth [89] algorithm improves on Apriori by constructing a specialized data structure called *FP-Tree*, which allows to store and mine only the patterns present inside the database, thus rendering the candidate generation process redundant, which enables a more efficient pattern mining process.

The second step of ARM is rule selection. The frequent patterns obtained in the previous steps are here used as potential candidates $X_c \subseteq \mathbb{B}$ to construct rules $X \implies Y$. Various statistical measures may then be employed to identify the degree of causality between each candidate X_c and the target Y. Three fundamental metrics are introduced here: support, confidence, and lift [87, 88].

Support [87], denoted by $\mathcal{S}(X)$, measures the relative frequency of a pattern in the database $\mathbb{D}$. Hence,

$$\mathcal{S}(X) = \frac{|\{X \subseteq t \mid \forall t \in \mathbb{D}\}|}{|\mathbb{D}|} = P(X), \tag{2.8}$$

where $|\cdot|$ denotes the cardinality of a set.

Support is downward-closed, i.e., $X_1 \subseteq X_2 \implies \mathcal{S}(X_1) \geq \mathcal{S}(X_2)$, because larger patterns are less likely to be entirely contained inside a transaction, therefore, in this case the numerator in (2.8) would be smaller. Similar to probabilities, support may also be calculated conditionally on a given subset of the database $\mathbb{D}$. For given transactions containing the predefined itemset Y, the conditional support is defined as

$$\begin{aligned}\mathcal{S}(X \mid Y) &= \frac{|\{X, Y \subseteq t \mid \forall t \in \mathbb{D}\}|}{|\{Y \subseteq t \mid \forall t \in \mathbb{D}\}|} \\ &= \frac{\mathcal{S}(X \cup Y)}{\mathcal{S}(Y)} = P(X \mid Y).\end{aligned} \tag{2.9}$$

Conditional support measures the fractional contribution of a pattern X to the pattern Y. For example, given X as item used to predict the purchase of a second item Y with $\mathcal{S}(X \mid Y) = 0.2$, X appears in 20% of the transactions containing Y. Confidence [87] is the ratio of transactions where observing X implies observing Y as well. Hence,

$$\mathcal{C}(X, Y) = \frac{\mathcal{S}(X \cup Y)}{\mathcal{S}(X)} = P(Y \mid X). \tag{2.10}$$

Lift, first introduced by [88], is defined as:

$$\mathcal{L}(X, Y) = \frac{\mathcal{C}(X, Y)}{\mathcal{S}(Y)} = \frac{\mathcal{S}(X \mid Y)}{\mathcal{S}(X)}, \tag{2.11}$$

which represents the increase in the probability of observing X, before (denominator) and after (numerator) having observed Y. If Y is a failure pattern, lift measures how much more likely it is to observe X in the presence of a failure, compared to the baseline case. In a monitoring context example, if a specific pattern of temperature (e.g., $t > 50$°C) is usually observed only in 1% of all observations, and it is observed in 10% of failure-related observations, then the pattern $X = \{t > 50\}$ has a lift of $10\%/1\% = 10$, to indicate that it is 10 times more likely to observe high temperatures in failure contexts. Large lift values (> 1) indicate strong positive association between the pattern and failure, while small values ($0 < l < 1$) indicate a negative association. A lift value of 1 indicates variable independence.

These three metrics may be used to evaluate the degree of causality relating X and Y. Based on these notions, ARM concepts can then be extended to model a sequential data problem. *SPM* [91–94] studies the discovery of rules of the form $(X_1 \implies X_2 \implies \dots \implies X_t) \implies Y$, where such items are observed at least once *in this order* in a set of sequences $\mathrm{S_i}$ of the form $\mathrm{S_i} = [t_1, t_2, \dots, t_j, \dots, t_L]$, $t_j \subseteq \mathbb{B}$. These sequences constitute the transactions of the sequential database $\mathbb{D} = \{S_1, S_2, \dots, S_n\}$, and $\mathbb{B}$ is again its item base determined by the union of all items in all transactions. Based on this domain translation, SPM algorithms can discover frequent sequential patterns $\mathbf{X} = \{X_1 \rightarrow X_2 \rightarrow \dots \rightarrow X_t\}$, and then the same statistical metrics (support, confidence, and lift) can be computed to evaluate their relation to Y.

SPM has applications in business intelligence, predictive maintenance, biomedicine, and telecommunications [93]. Algorithms to discover frequent sequential patterns include Generalized Sequence Patterns (GSP) [91], Prefix-projected Sequential PAttern Mining (PrefixSpan) [92], and Sequential PAttern Discovery using Equivalence classes (SPADE) [93]. GSP in an Apriori-like algorithm translated to the sequential mining domain; PrefixSpan is a so-called pattern-growth method, which utilizes the same principles of FP-growth, i.e., the extension of existing patterns to reduce the pattern search space. SPADE utilizes equivalence classes to reduce the number of database scans and improve the runtime of sequential pattern mining.

Other Classical ML Models

Three important categories of classical ML approaches are tree-based approaches, similarity-based approaches, and kernel-based approaches [82, 83].

Tree-based approaches [82,83] determine a prediction through a hierarchical series of logical tests, learned and stored in a *decision tree*. Each test determines a split in the feature space (e.g., $x_3 > 5$) and in the training set, by which some samples are assigned to the left split, and the other to the right split. By doing so, decision trees partition the sample space into increasingly smaller sub-areas, to the point where a specific class (or value) can be confidently assigned to the input sample. The optimal splitting decisions (with corresponding thresholds) are determined based on entropy and sample purity metrics (such as the Gini index).

Random Forests (RFs) [95] combine multiple decision trees trained on random subsets of the training set (known as *bagging*) to boost the performance of the model and decrease prediction variance. The final prediction is obtained by voting or averaging out the prediction of individual trees. Additionally, random forests may apply feature bagging, i.e., the random selection of feature subsets to specific trees.

Tree-based methods are simple, accurate and interpretable algorithms, however, they are not indicated if the decision problem is not easily separable according to the input features.

Similarity-based approaches, such as k-Nearest Neighbors (kNN), allow estimating scalar quantities and categories based on the similarity to other samples in the training set. kNN in particular tracks the k closest matches to the input sample in terms of a predefined distance (e.g., Euclidean, Manhattan, Mahalanobis, Levenshtein, $\ldots$) and produces a prediction response by averaging or voting on the labels of neighboring samples. It is a simple method that requires no training; however, it suffers from inefficient runtime due to the $O(N)$ computation of distance to all samples in the training set (which can be mitigated by the use of some data structures, such as a *k-d tree*).

A Support Vector Machine (SVM) is a kernel-based method similar to binary logistic regression, in that it estimates the output class based on the result of the linear parametric inequality $\mathbf{w} \cdot \mathbf{x} + \mathbf{b} > 0$. The left term of this inequality can be rewritten to be proportional to the dot product of $\mathbf{x}^{(\mathrm{i})}, \mathbf{x}^{(\mathrm{j})}$ samples. This dot product can be replaced by the dot product in an arbitrary feature space Φ (even infinite-dimensional). Applying this "trick" (so-called *kernel trick* because the dot product in the feature space is called a *kernel*) allows efficient modeling of non-linear, high-dimensional dependencies with efficient runtime, by using arbitrary non-linear *kernels* (such as the Gaussian Kernel or the Radial Basis Functions). SVMs are popular and efficient algorithms for classification. They cannot be directly applied to multi-class classification problems and their parameters are difficult to interpret.

A related technique is Kernel Density Estimation (KDE), a technique to estimate Probability Density Functions (PDFs) of random variables using kernel-based smoothing. PDF estimation allows to obtain a graphical representation of a population of samples and to draw statistical insights about the data distribution, e.g., estimating typical ranges for monitored values and their variability in the population.

Deep Learning

Deep learning is the study of deep neural networks (also known as *artificial neural networks*), in analogy with the structure of the human brain [82, 83]. Deep learning has seen a tremendous growth in its popularity and application, largely as the result of more powerful hardware, larger datasets and improved techniques to train deeper networks.

Deep neural networks are composed of several network layers, which produce increasingly abstract and complex representations of the input feature space [83]. These representations (also called *activations*) are obtained by applying parametric mathematical operations, such as weighted matrix multiplications, which are learned during training to minimize the output prediction error.

To this end, all notions regarding loss function minimization and first-order gradient optimization described for linear models apply to deep learning models as well, with some more considerations [85, 97].

First, differently from linear models, the optimization parameters (or *weights*) of a neural network are distributed across several computation layers. As a consequence, the computation of the weight gradients is different for each layer and increasingly complex for shallower layers, which are farther away from the loss

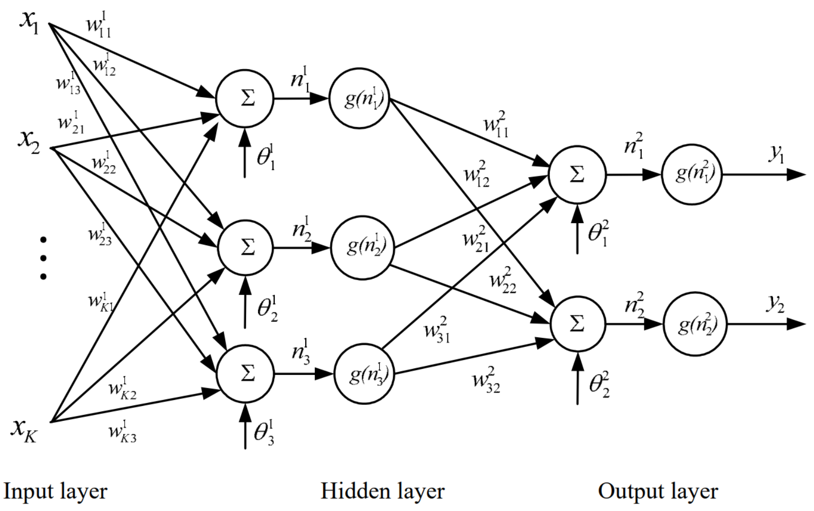


Figure 2.4 Example of MultiLayer Perceptron (MLP) architecture [96] with K input features, one hidden layer, and 2 output features. For each layer l, the outputs from the preceding layer are multiplied by a weight matrix w^l and added to a bias column θ^l (i.e., a fully-connected layer). The resulting output n^l is transformed using the activation function g to obtain the final layer activation, which is fed into the next layer $l + 1$.

function. To address this, the *backpropagation algorithm* [97] is used. It propagates the loss function gradient backward from the last layer to the first, by applying the chain rule to compute the derivative of composite functions, which allows to divide-and-conquer the problem and re-use existing computations. For this reason, in the context of neural network training, a *forward pass* refers to the normal input-to-output propagation, which computes the loss function and the sample predictions (used during inference as well), and as *backward pass* the output-to-input propagation, which computes the weight gradients with respect to the loss function and is used to apply gradient descent [85].

Second, deep learning models embed a large number of parameters, ranging from a few tens of thousands of LeNet-5 [98] to trillions of parameters of modern language models [9]. The large number of parameters and the presence of non-linear transformations renders the loss function landscape during optimization difficult to handle, with different problems arising, such as vanishing and exploding gradients, saddle point, and ill-conditioning [83]. Therefore, several techniques must be adopted to ensure that the approximate convergence point (local minimum) does not differ significantly from the optimal solution (global minimum). These techniques include increasing model capacity, network weight initialization, learning rate scheduling, and the use of more advanced optimizers (such as Adam [99]), which introduce gradient momentum and adapt learning rate values to the optimization trajectory.

Third, deep learning models require a large quantity of data to be trained, due to the high number of parameters involved. This means the full estimation of the gradient from the training set is not feasible due to memory constraints. SGD overcomes this issue by computing the gradient for each sample separately, which however leads to noisy estimates of the gradient. In modern applications, an intermediate solution called *mini-batch SGD* is used, where a smaller set of samples (or *batch*) that can fit in memory is used to estimate the gradient. Samples in the training set are selected iteratively to construct mini-batches and reduce gradient noise. Several iterations (or *epochs*) over the training set are typically required. Mini-batch optimization has been shown empirically to have a regularization effect as well.

Four are the main kinds of neural network architectures relevant to this dissertation: MLPs, Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and Transformers [7].

The quintessential type of deep learning model is the MultiLayer Perceptron (MLP) [83] (see Figure 2.4). MLPs are composed of several *fully-connected layers*, which apply linear transformations to the data via multiplication of a weight matrix. These fully-connected layers are alternated with *activation function* layers, which apply non-linearity transformations to the linear output layers. These activation functions are responsible to select and filter important features from the applied transformations.

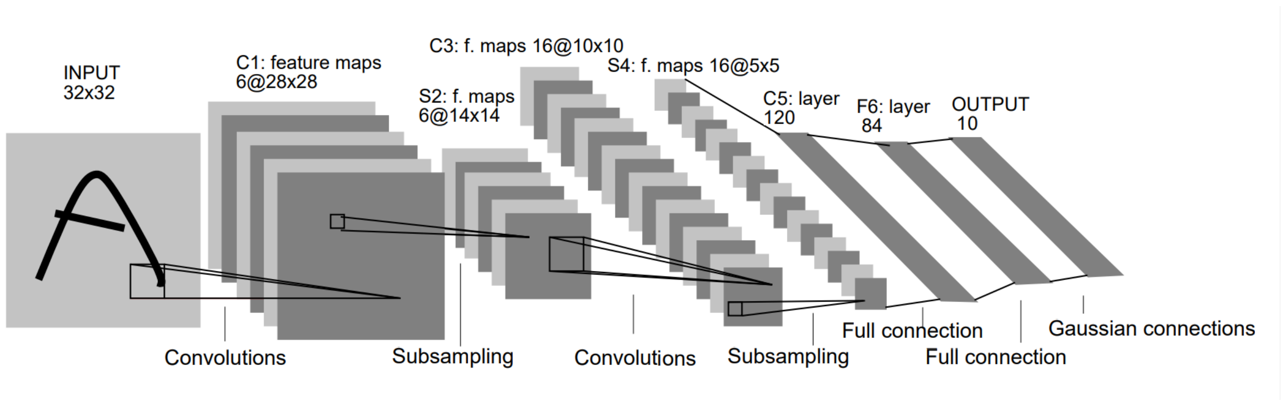


Figure 2.5 Example of CNN architecture (LeNet5 [98]) for handwritten character recognition, a classic computer vision task. LeNet5 utilizes alternate 2D convolutional and pooling layers for efficient spatial feature extraction. The last layers ('full connection') implement fully-connected layers to map the latent representations to the expected 10 prediction classes.

Common choices of activation functions include the logistic (or sigmoid) function σ (Equation 2.3), the hyperbolic tangent (tanh) function, the Rectified Linear Unit (ReLU) [100] and Gaussian Error Linear Unit (GELU) function [101]. While the exponential functions provide continuous differentiability, they suffer from gradient saturation at the extrema. ReLUs are not differentiable at zero and do not propagate information for negative values (due to their null value). GELUs are a hybrid solution, as they are fully differentiable and saturate only on one side. They are however more complex to implement.

Other layer operations are frequently implemented between linear layers. Dropout [102] is the probabilistic zero-ing of some activations to reduce co-adaptation across layers and improve robustness and generalization. Batch Normalization [103] normalizes the intermediate activations of a network layer. It helps address internal covariance shifts and accelerates training.

Convolutional Neural Network (CNN) architectures constitute a second category. They have been used for image recognition tasks [5, 98, 104], and time-series classification [105]. The fundamental transformation operated in CNNs is *convolution*, i.e., a particular type of shift-invariant transform which processes input considering locality of data and by re-using weights across different parts of input (see Figure 2.5). Convolution is a translation-invariant operation, which is more suitable for inputs with spatial information, such as sequences, images, or 3D shapes. The shared parameters of convolutional layers, together with the application of *pooling* layers, allow to highly reduce the number of parameters and the chances of overfitting. Other advanced CNN layers include 'skip' residual connections and average global pooling, introduced in ResNet [104].

One of the limitations of CNNs models is their limited flexibility with respect to input and output size. For 1D sequences, frequently used in NLP and biomedical tasks, this problem is overcome by Recurrent Neural Networks (RNNs) [6, 83]. RNNs process input sequences one time step at a time, so that network weights are shared across time steps (Figure 2.6a). This allows RNNs to handle arbitrarily-long input sequences and to produce variable-length sequences as outputs as well, which allows to efficiently model sequence-to-sequence problems effectively, such as translation and text generation. Moreover, the RNN can rely on knowledge from previously observed samples by storing information in its internal state.

RNNs back-propagate gradient through time, which causes them to have inherent limitations in terms of learning, such as the gradient vanishing problem, for which it is difficult to model long-term dependencies effectively [107]. This has motivated the introduction of more advanced RNN layers (Figure 2.6b), such as LSTM cells [6], which allow the gradients to flow more consistently across time steps, GRU units implement a functionality similar to LSTMs, with fewer parameters. RNNs have been used for time-series classification [108], neural machine translation [109], and forecasting [110]. However, as already mentioned, the vanishing gradient problem still prevents RNNs from propagating relevant context information far along the sequence. Moreover, the sequential approach of RNNs, with dependency on previous tokens for current computations, prevents from taking full advantage of parallelism to speed up inference.

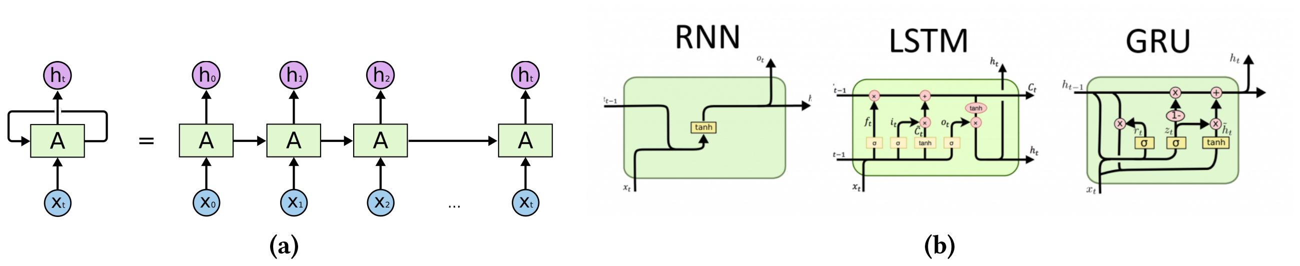


Figure 2.6 RNN working principles [106]. On the left, an unrolled RNN layer, which shows how each time step input of a t-long sequence X is used to process existing state information and produce a hidden-state representation h. On the right, different kinds of RNN cells, which can be applied for the 'A' block: traditional RNN, Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU).

To overcome these limitations, the *attention* mechanism [7] has been introduced. Attention is applied to sequences to compute soft weights to assign to each token (or time step) of a sequence (Figure 2.7a). These weights are used to put emphasis (from here the metaphor of 'attention') on specific parts of the previous input without relying on memory states. The network learns to focus on these specific parts by learning how to compute attention weights in different contexts. Attention layers allow modeling long-term dependencies while parallelizing network layer computations.

The original attention paper also proposes the Transformer architecture [7], which constitutes the foundation for modern language models [8, 9, 111] (see Section 2.3.4). The Transformer is composed of embedding, encoder, and decoder blocks as described in Figure 2.7b. Because of the attention block, Transformers are able to access any predefined point along the sequence.

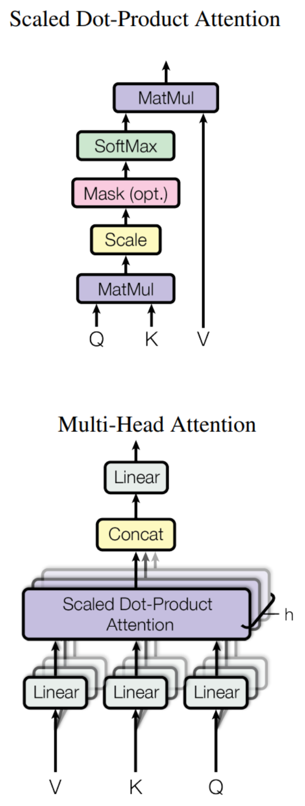
Transformers have widely replaced RNN-based architectures for language-related tasks and have been extended to other domains, such as computer vision, and biological sequence analysis.

2.3.4 Natural Language Processing (NLP)

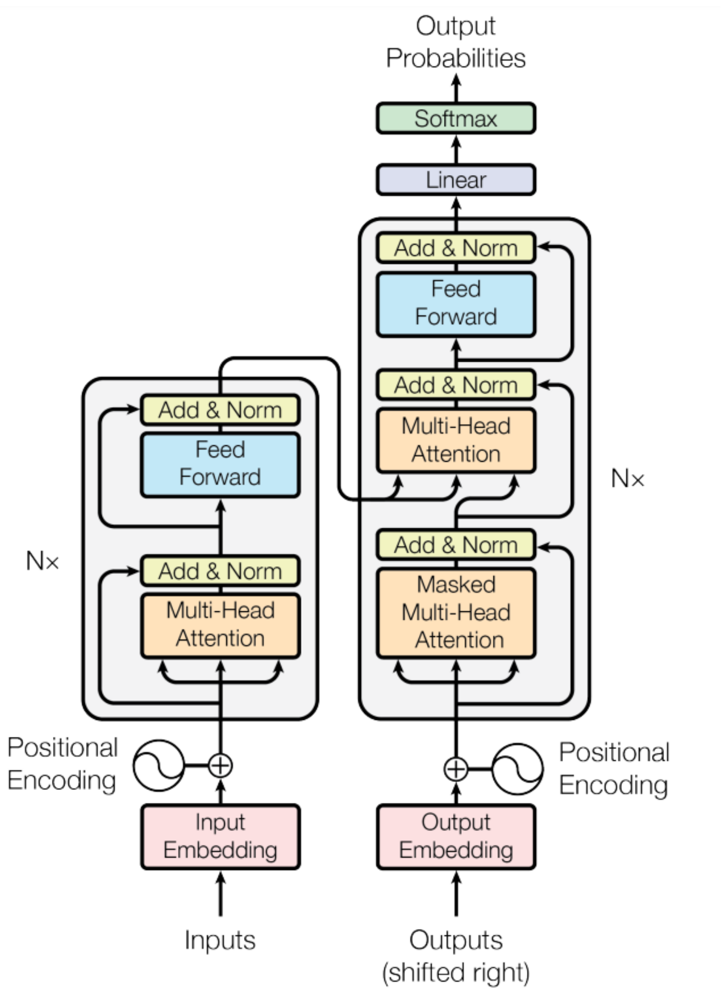
NLP is a subfield of AI that studies computer algorithms to process and interact with human language [112]. Common NLP problems include translation, speech recognition, text-to-speech synthesis, parsing, and *sentiment analysis*, a category of text classification used to determine the polarity of a given text, where the polarity can represent different aspects of language, such as positivity/negativity, objectivity/subjectivity, or hateful/supportive content. Syntactical analysis tasks include Part-of-Speech (PoS) tagging, Named-Entity Recognition (NER), sentence and work segmentation (or *tokenization*). Generative tasks include question-answering, text summarization and text-to-image generation.

As AI, NLP started as a study of symbolic and rule-based methods for language-related tasks, to then expand into the realm of statistical methods and ML. Regular expressions [112] are a traditional method for pattern matching, based on quantifier (?+*) and grouping syntax. Edit distance [112] is a dynamic programming approach used to estimate the number of changes between two texts. It is used for grammar error correction and similarity search. Context-free grammars [112], and in particular Probabilistic Context-Free Grammars (PCFGs), may be used to mathematically describe context relationships between entities in a sentence, given a set of abstract rules relating to how to associate terms. They may also be used for parsing and causal inference.

A *Naive Bayes* classifier [82, 112] is a ML model frequently used in NLP tasks (e.g., sentiment analysis, or spam detection). Naive Bayes classifiers rely on the Naive Bayes assumption, i.e., the input features are not correlated, which highly simplifies variable interdependencies to construct a tractable model for both discriminative and generative tasks. Because features are independent, the conditional class probability only depends on whether specific feature values are observed or not (and possibly how many times). For NLP, input features are typically subunits of text (such as words, punctuation or characters) called *tokens*. If tokens are words, this specific feature representation is called Bag-Of-Words (BoW), because it only stores which words have been observed and how many times (e.g., $\{\texttt{'spam'}: 4, \texttt{'egg'}: 2\}$). The downside of a BoW model is that the order of tokens is not considered. To address this issue, combinations of n adjacent tokens, known



(a) Architecture of an attention layer [7]. The three inputs Query (Q), Key (K), and Value (V) are processed to resemble the access to a 'fuzzy' retrieval system, where input values *V* are re-weighted based on the specific information which must be accessed (as specified by *K* and *Q*). This re-weighting mechanism is repeated *h* times in each multi-head attention block.



(b) Architecture of the Transformer network [7]. It is composed of an encoder block (left) and a decoder block (right). Both blocks are preceded by embedding layers, which represent sequence tokens in a latent vector space while adding position information (positional encoding operation). Both blocks implement self-attention, i.e., attention with query, key, value all equal to the input. The decoder block then differentiates by applying a second attention layer with key and query from the encoder step. This scheme is repeated *N* times. The architecture is completed by traditional MLP layers (fully-connected and softmax).

Figure 2.7 Attention block (left) and transformer architecture (right), as presented in the original paper [7].

as *n-grams* or *shingles*, may be used. *n* may either be a predefined length, or the discovery of frequent token combinations may be left open to any length (as in Byte-Pair Encoding (BPE), see Section 5.4.3).

Input tokens may also be represented as vectors (known as *embeddings*) rather than predefined categorical values. Term Frequency-Inverse Document Frequency (TF-IDF) [113] represents words based on their occurrence counts in large text corpora. Words that co-appear together then have similar meanings and representations. Word2Vec [112, 114] is a word embedding technique based on context and similarity learning. Word2Vec learns compact vector representations of words by training a logistic regression classifier, which must predict whether two words are present in the same context or not, by evaluating their similarity via cosine distance. The resulting weights can be used as representations for words in other NLP models.

Recent advances in deep learning have encouraged the adoption of several transformer-based architectures specialized for speech and text analysis, such as Bidirectional Encoders Representations from Transformers (BERT) [111] or Generative Pre-training Transformer (GPT) [8, 9]. Thanks to the large quantity of text data and the effective representation learning provided by attention, these models are able to capture complex relationships between tokens, to solve a variety of NLP problems at a state-of-the-art level.

The learning phase of transformers for NLP is typically divided into *pretraining* and *finetuning*. During pretraining, the NLP model is trained in a self-supervised fashion, i.e., it is trained to recognize parts of the input which have been removed, e.g., predict the next sentence, or predict the masked tokens in an input sentence. This enables the model to learn the language syntax and the contextual relationships between tokens present in the language. Because these contextual tasks are self-supervised, the pretraining step does not require any labeled data, and a large corpus of the target language is sufficient. In the second phase, the pretrained NLP model is trained to perform the target NLP task via finetuning. The knowledge learned during pretraining is retained by preserving the weights of all or a part of the network, which are partially updated or extended with additional layers to specialize on the target task. This enables a higher degree of generalization.

The pretrain-finetune scheme is used by many NLP models, including BERT. BERT is a Large Language Model (LLM) based on self-supervised pretraining as described above [111]. BERT has empirically demonstrated to improve performance in many language understanding tasks, including question answering, text classification, sentence pair completion, named entity recognition, etc. [111, 115, 116]. Some other models, such as GPT4 [9] incorporate Reinforcement Learning from Human Feedback (RLHF) in addition to supervised learning to improve the performance of the model in autonomous agent scenarios (e.g., chatbots and generative AI).

2.3.5 Evaluation Metrics for ML Models

Throughout the dissertation, quantitative results are frequently presented for the proposed ML approaches and the related work under discussion. This section provides an overview of the evaluation metrics employed for evaluation and comparison of ML models [73, 82, 83, 112].

For regression tasks, a frequently adopted metric is the MSE, i.e., the average of the squared difference between target and predicted values (as defined for the loss function in Equation 2.5):

$$MSE = \frac{1}{N}\sum_{i}^{N}\left(y(\mathbf{x}^{(i)}) - t_i\right)^2 \tag{2.12}$$

A measure widely adopted for classification problems (in AIOps, e.g., software defect prediction, root-cause diagnosis, command risk classification) is *accuracy*, i.e., the ratio of classified samples which are assigned to the correct class ($\frac{1}{N}\sum_N \#correct$).

In some contexts, however, accuracy is a misleading metric to evaluate the quality of a predictor. This is the case, for example, for problems with a high predominance of one class, where trivial models can be constructed to reach high accuracy by simply exploiting data skewness. This consideration applies specifically to detection problems (e.g., online failure prediction and anomaly detection) where the positive class, i.e., the detected event (e.g., a failure), may appear less frequently than the negative class, even though it constitutes the most critical aspect from the perspective of evaluation. In such cases, it is common to adopt measures that are more representative of the minority class. To this end, the notion of contingency table is introduced [73]:

		Predicted Class	
		+	-
True Class	+	True Positive (TP)	False Negative (FN)
	-	False Positive (FP)	True Negative (TN)

Table 2.1 Contingency table applicable for binary classification tasks.

then, accuracy can be defined as:

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.13}$$

To quantify the ability of a predictor to identify positive instances correctly, the *precision* measure, also called *Positive Predictive Value (PPV)*, is usually employed, while to measure the ability of a detector to report true positive samples, the *recall* measure, also known as *True Positive Rate (TPR)*, or *sensitivity*, is used. They are defined as:

$$P = PPV = \frac{TP}{TP + FP} \qquad\qquad R = TPR = \frac{TP}{TP + FN} \tag{2.14}$$

Precision and recall may be traded off by adjusting sensitivity thresholds, so that an increase in precision can be obtained by reducing recall and vice-versa. Therefore, they are not particularly useful when used in isolation. One possibility to simultaneously evaluate both measures is to use the *F1-score* (or *F-score/F-measure*), i.e., the harmonic mean of precision and recall:

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R} \tag{2.15}$$

Another metric that frequently replaces precision is the *specificity*, or *True Negative Rate (TNR)*, which measures the ratio of negative instances correctly identified over the total negative population. Conversely, the *fall-out* (or *false alarm rate*, or *False Positive Rate (FPR)*) measures the probability of reporting a false positive (false alarm) in the presence of a negative sample:

$$TNR = \frac{TN}{TN + FP} \qquad\qquad FPR = \frac{FP}{TN + FP} = 1 - TNR \tag{2.16}$$

A final possibility is to use Receiving Operating Characteristic (ROC) curves, parametric line plots which describe the variation of two metrics in relation to changes in the sensitivity threshold. Precision-recall curves are possible, but it is also common to plot the recall against the fall-out (TPR vs. FPR). From this type of curves, the Area under the Curve of Receiving Operating Characteristic (AUCROC) measure can be computed via integral approximation. A higher AUCROC score then indicates an overall better classifier independently of sensitivity thresholds.

3 Systematic Review of AIOps

In the previous chapters, the concept of AI for IT Operations (AIOps) and the reasons behind its emergence, have been presented. It was also mentioned how its large applicability for different tasks renders it a heterogeneous and unstructured research field, with ambiguous problem statements and complexity in comparing solutions directly. In this chapter, the structure and characteristics of AIOps are investigated from a research perspective, to understand past contributions in terms of macro-areas and common problems, as well as delineate future directions to effectively support IT Operations.

3.1 Introduction

In recent years, the AIOps paradigm has quickly spread across several technology companies, which have first started to adopt AIOps internally to maintain their on-premise computing facilities efficiently [15, 16, 24, 117], to then over the past few years deliver AIOps tools as products. At the same time, in the academic context different research groups have taken advantage of recent advances in Machine Learning (ML) and AI to explore open AIOps problems.

However, the concept of applying AI tools to support IT operations is not new [75]. First contributions in this direction date to the mid-1970s, when several works have proposed to localize software bugs in source code by applying statistical models and code complexity metrics [118–123]. Since the early 1990s, several online software [124–127] and hardware [128–130] failure prediction models have been proposed. Other failure prevention methods also date back to the same period [131–137]. Other areas of AIOps, such as anomaly detection, event correlation, or problem identification, have attested contributions for at least 25 years [138–144]. This steady and growing flow of contributions, however, was never recognized as homogeneous for a long time. Because of the large diversity of tasks, multi-modality of data, and target systems to support, AIOps have grown apart in terms of terminology and methodology.

This has the consequence that AIOps is nowadays a largely unstructured field of study. Despite recent efforts to categorize and collect similar contribution in specialized scientific conferences and industry meetups [145, 146], the landscape of potential applications remains highly unclear. The very same definition of AIOps is subject to debate [15–21]. At the same time, the high number of application areas renders the search and collection of relevant papers difficult.

Some previous systematic works [73, 144, 147–165] only treat single tasks or subareas inside AIOps (see Section 3.2). The additional scarcity of previous comparison studies and surveys motivates the need for a comprehensive study, able to collect, categorize and summarize past contributions, to facilitate the comparison and sharing of all available approaches for each Operations & Maintenance (O&M) task. These considerations motivate the necessity for a complete and updated study of AIOps contributions.

This chapter explores the AIOps landscape through an in-depth systematic literature review. First, a Systematic Mapping Study (SMS) [166], a research methodology used to delineate common trends, problems and tools inside a research topic, is conducted for the AIOps field. After the identification of over 1000 AIOps contributions through the mapping study, the insights drawn from such analysis are presented and discussed, including the identification of the most common tasks, data sources, and target components. This enables the creation of an AIOps taxonomy and the identification of failure management as a fundamental macro-area of interest. Inside failure management, several areas of future research interest are identified and discussed.

The present chapter is structured as follows. This short introduction corresponds to Section 3.1. Related work in categorizing AIOps is described in Section 3.2. An in-depth description of the mapping study methodology is presented in Section 3.3, reporting and motivating the planning choices regarding problem definition, search, selection and mapping. In Section 3.4 the results of the mapping study are presented, by reporting temporal and categorical trends by introducing the AIOps taxonomy empirically derived from the observed AIOps contributions, and by discussing the most commonly treated categories and sub-problems of AIOps. Section 3.5 summarizes the outcomes of the discussion.

3.2 Related Work

As the AIOps areas mentioned above are large both in number and range of applications, it is reasonable to expect numerous works focusing on filtering and categorizing best approaches and practices [75]. Several other surveys and mapping studies have been in fact conducted in the areas covered by AIOps [73, 144, 147–165]. However, no previous work has provided an updated comprehensive review of AIOps approaches.

Table 3.1 summarizes the most relevant survey and systematic review contributions regarding IT Operations and AI, organized by main topic and other focuses. Most works treat single tasks [73, 144, 151–155, 157–161] or general goals [156, 162–165] inside AIOps, which are specific to particular intervention methods.

A second category of works [147–150] treat failure management integrally, but some works inside this group are outdated and do not reflect the current progress of the field, while the most recent works do not focus on AI-based approaches or do not offer a comprehensive list of contributions. The closest match to the current analysis is represented by the work of Mukwevho et al. [150], who present a survey on fault management in cloud systems. Differently from them, the mapping study here presented does not focus on any particular computing system, while it focuses on the manifestation of faults (i.e., errors and failures) rather

Ref.	Year	Type	Main Topic	Focuses
			IT Operations	
[147]	2007	Survey	IT Operations	AI, Operational Research
[148]	2011	Survey	IT Operations	AI, Operational Research
[149]	2013	SMS	Failure Management	Temporal & Geographical Trends, Service Level, Others
[150]	2018	Survey	Failure Management	Cloud Computing
			Failure Prevention	
[152]	2011	Survey	Failure Prevention	Combinatorial Testing
[153]	2015	Survey	Failure Prevention/Detection	Software
[154]	2016	Survey	Fault Injection	Software
[155]	2017	Survey	Software Defect Prediction	Machine Learning, PROMISE dataset
			Failure Prediction	
[156]	2007	Survey	Failure Prediction	AI, Integrated Systems
[157]	2007	Survey	Failure Prediction	Clusters
[73]	2010	Survey	Failure Prediction	Online Methods
[158]	2019	Survey	Failure Prediction	High Performance Computing
			Failure Detection	
[159]	2015	Survey	Anomaly Detection	Bottleneck Identification
[144]	2009	Survey	Anomaly Detection	–
[151]	2019	Survey	Anomaly Detection	Deep Learning
[160]	2013	Survey	Anomaly Detection	Network
[161]	2008	Survey	Internet Traffic Classification	Machine Learning, IP Networks
			Root Cause Analysis (RCA)	
[162]	2017	Survey	RCA	–
[163]	2016	Survey	Fault Localization	Software
[164]	2015	Survey	Fault Diagnosis	Model- and signal- based approaches in Industrial Systems
[165]	2015	Survey	Fault Diagnosis	Knowledge-based and hybrid approaches in Industrial Systems

Table 3.1 Related AI surveys and SMSs in areas covered by the AIOps survey [75].

than root causes (see Section 2.2.3 for terminology). Moreover, AI and ML approaches are integrated in the conventional scheme of failure management approaches, rather than being treated as a separate category. All these considerations motivate the need for an in-depth study in this area, like the one conducted and below summarized [75].

3.3 Systematic Mapping Study in AIOps

This section presents the methodology of the mapping study conducted for the AIOps field [38, 167].

3.3.1 Definition and Planning

A Systematic Mapping Study (SMS) is a scientific methodology widely adopted in many research areas, including software engineering [166]. The main objective of a SMS is to provide an overview of a specific research area, obtain a set of related papers, and delineate trends inside such area. Relevant papers are collected via predefined search and selection techniques, and research trends are identified using categorization techniques based on different aspects of the identified papers, e.g., topic or contribution type. The SMS methodology was selected because it allows gathering contributions and obtaining statistical insights about AIOps, such as the distribution of works in different subareas and the presence of temporal trends for particular topics. SMSs have also been shown to increase the effectiveness of follow-up systematic literature reviews [166]. To this end, the SMS has also been used to collect references for a survey on failure management in AIOps separately published [75].

SMS planning is the decomposition of the SMS review into sequential steps to perform. According to the outline proposed by Petersen et al. [166], a SMS is composed of:

- *formulation*, i.e., express the goals intended for the study through research questions. Equally important is to clearly define the scope of the investigation;
- *search*, i.e., define strategies to obtain a sufficiently high number of papers within the scope of the investigation. This requires the selection of one or more search strategies such as database search, manual search, reference search, etc.;
- *selection* (or *screening*), i.e., define and apply a set of inclusion/exclusion criteria for identifying relevant papers inside the search result set;
- *data extraction and mapping*, i.e., gather the information required to map the selected papers into predefined categorization schemes. Finally, results are presented in graphical form, such as histograms or bubble plots.

The AIOps mapping study uses the scheme with some minor modifications. The next sections illustrate and motivate the choices regarding these four steps for the SMS on AIOps.

3.3.2 Formulation

The main goal of the formulated mapping study is to identify the extent of past research in AIOps. In particular, it is desirable to identify a representative set of AIOps contributions which can be grouped based on the similarity of goals, employed data sources and target system components. It is also important to understand the relative distribution of publications within these categories and the temporal implications involved. Formally, the following research questions are formulated [38]:

RQ1. What categories can be observed while classifying AIOps contributions in the scientific literature?

RQ2. What is the distribution of papers in such categories?

RQ3. Which temporal trends can be observed for the field of AIOps?

In terms of scope, the boundaries of AIOps are expressed as the union of goals and problems in IT Operations when addressed using AI techniques. To circumvent ambiguity about the term AI, the same inclusive convention described in Section 2.3.1 was adopted, which includes in AI both data-driven approaches, such as

ML and Data Mining, as well as goal-based approaches, such as logic, search and optimization. However, the majority of search efforts are concentrated on the first category due to its stronger presence and connection to AIOps methodologies (e.g., monitoring, data collection, generalization).

3.3.3 Search and Selection

Selection Criteria

As the paper selection principles are applied in connection with multiple search techniques, they are illustrated beforehand here for clarity. In terms of inclusion criteria, only one relevance criterion is defined, based on the main topic of the document. Following the discussion on scoping presented above, this inclusion criterion is composed of two necessary conditions:

- the document references one or more AI methods. These mentions can either be part of the implementation or as part of its discussion/analysis (e.g., in a survey). Any mention to AI algorithms employed by others (i.e., mentioned in the related work section or as baseline comparison) that is not strictly the focus of the document, is not considered valid;
- the document applies its concepts to the IT administration of some kind of large computing system. Therefore, papers with no specific target domain or with a target domain outside IT Operations are excluded. For reference purposes, a non-exhaustive list of IT environments considered relevant and frequently mentioned in the explored papers is here presented: Cloud Computing, Web Server, Grid Computing, High Performance Computing (HPC), Cluster Computing.

In terms of exclusion criteria, the following are defined as exclusion rules:

- The language of the document is not English;
- The document is not accessible online;
- The document does not belong to the following categories: scientific article (conference paper, journal article), book, white paper;
- The main topic of the document is one of the following: cybersecurity, industrial process control, cyber-physical systems, and optical sensor networks.

For the special case of survey and review papers, both are considered relevant for the mapping study, but they are excluded from the final result set, as these articles are useful to find other connected works through references, but they do not constitute novel method contributions to the field.

Database Search

Database search represents the first and most important step of the search process, as it provides the highest number of results and performs an initial screening of non-relevant papers. Database search is composed of three steps: keywording, query construction and result polling. For keywording, the PICO scheme [166] is used to derive a set of keywords for AI and a set of keywords for IT Operations, both listed in Table 3.2.

Then, following the scoping considerations, search queries are constructed so that they return results where *both* AI and IT Operations are present. In particular, logic conjunction of keywords (`AND`) is applied across all combinations of the two keyword sets (e.g., "logistic regression" and "cloud computing"). This helps enforce precision in the search results. For keywords with synonyms and abbreviations, all equivalent expressions are permitted via `OR` disjunction. In addition, general search queries are also performed, when related to the topic as a whole (e.g., "AIOps"). Some queries with common terms are grouped to reduce the number of queries.

Three online search databases that are appropriate for the scope of the investigation are selected: *IEEE Xplore* [168], *ACM Digital Library* [169], and *arXiv* [170]. For each query, the analysis is restricted to the top 2000 results returned. Results are aggregated from all searches in one large set of papers, removing duplicates and annotating for each item corresponding search metadata (e.g., above number of hits, index position in corresponding searches, etc.). The result from this step consists of 83817 unique articles. For each item, the

AI Keywords	IT Operations Keywords
(“AI” OR “artificial intelligence”)	(“DevOps” OR “site reliability engineering” OR “SRE”)
“classification”	(“IT operations”)
“clustering”	(“anomaly detection” OR “outlier detection”)
“logistic regression”	(“cloud computing”)
“regression”	(“cloud”)
(“DL” OR “deep learning”)	(“fault detection” OR “failure detection”)
(“ML” OR “Machine Learning”)	(“fault localization” OR “failure localization”)
(“inference” OR “logic” OR “reasoning)	(“fault prediction” OR “failure prediction”)
(“supervised” OR “unsupervised” OR “semi-supervised” OR “reinforcement”) AND (“learning”)	(“fault prevention” OR “failure prevention”)
(“support vector machine” OR “SVM”)	(“log” OR “logs” OR “log analysis”)
(“tree” OR “tree-based” OR “trees” OR “forest”)	(“metrics” OR “KPI” OR “key performance indicator”)
((“bayesian” OR “neural”) AND “network”)	(“remediation” OR “recovery”)
(((“hidden” AND “markov”) OR (“gaussian” AND “mixture”)) AND “model”)	(“root-cause analysis” OR “root cause analysis”)
((“datacenter” OR “data center”) AND “management”)	(“service desk automation”)
	(“tracing” OR “trace” OR “traces”)

Table 3.2 The two keyword sets obtained via Population, Intervention, Comparison Outcome (PICO) used for database search in the mapping study [38].

title, authors, year, publication venue, contribution type and citation count (from Google Scholar [171]) are collected.

Preliminary Filtering and Ranking-based Selection

In the filtering step, the quality of the selection of papers is improved [38]. First, papers are automatically excluded based on publication venue, for those venues that are clearly not relevant for topic reasons (e.g., meteorology). Papers are also excluded based on the year of publication (year $<$ 1990) as it precedes the advent of large-scale IT services. By doing so, approximately 8000 elements can be excluded.

Usually, at this point, a full-text analysis would be performed on all the available papers to screen relevant contributions using the above-cited selection rules. Although results have been partially filtered, it is still not feasible to perform an exhaustive selection analysis, even as simple as filtering by title. It is also impractical to attempt an automated selection by content, as it is not clear how to perform an efficient, high-recall, high-precision text classification without supervision. Therefore, before proceeding with the rest of the search and selection steps, a ranking procedure is applied on these intermediate results, so that the investigation of more relevant papers can be prioritized. The exclusion and inclusion rules of Section 3.3.3 are applied to the papers, examined in ranking order.

This approximate procedure, however, raises the question of when it is convenient to halt the selection procedure and discard the remaining items. To solve this, a new approach, based on observations of ranked items, is devised. The new method is based on the following assumption: a considerable ratio of relevant papers can be identified by ranking and selecting top results using different relevance criteria (conference, position index in the query result set, number of hits in all queries, etc.), but in this sorting scenario a long-tail distribution of relevant documents can also be observed, i.e., some relevant papers appear in the last positions even after sorting by relevance heuristics (see Figure 3.1). This is coherent with the known impossibility of performing exhaustive systematic literature reviews and mapping studies, as completing the long tail provides fewer results at the expense of a larger research effort. The ratio of relevant papers in the long tail is assumed to be constant and comparable in magnitude to the number of relevant papers when sampled randomly from the result set. Based on this assumption, the following procedure is devised:

- papers in the result set are screened, ranked according to different relevance heuristics (e.g., number of hits in queries), and the ratio of relevant papers identified is monitored over time;
- the result set is also examined in random order, and the same ratio is measured;

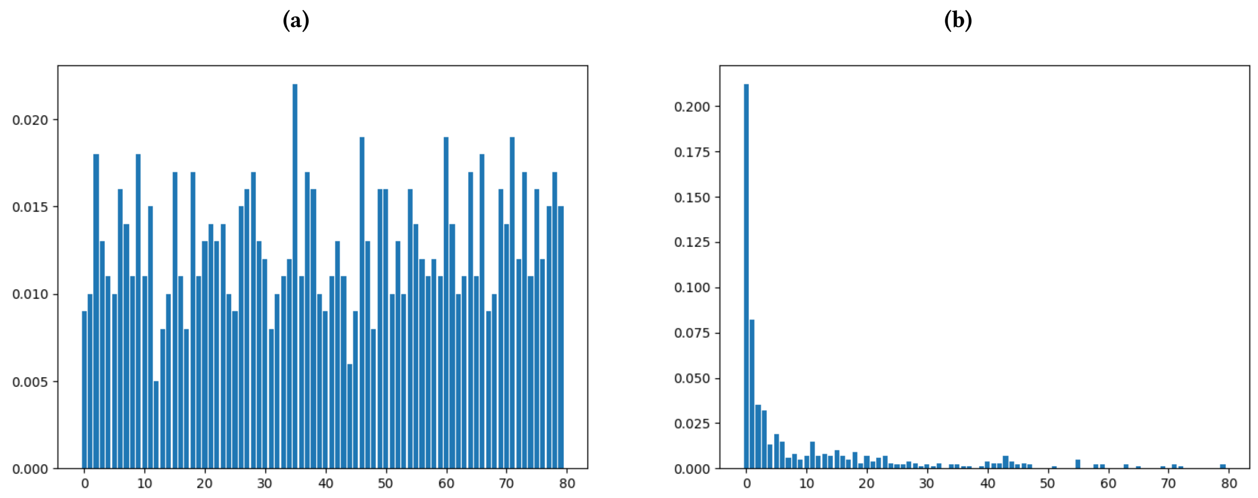


Figure 3.1 Ranking-based Selection during the mapping study [38]. The graph shows the estimated relevance probability for collected papers (y-axis), as a function of the index in the result set (x-axis, in thousands), with paper arranged (a) in random order (b) using a relevance heuristic based on search hits. Using the heuristic (b), the majority of relevant papers can be identified by examining only a small fraction of the set (the top results on the left side).

- When the two ratios are comparable, the tail of the distribution of relevant papers has been reached and the collections of new relevant papers can be halted.

As sorting criteria, the number of hits in the search performed in the previous step, as well as other more complex heuristics, taking into account the index position in result sets and the number of citations, are used. When examining a paper, the full content is investigated to identify concepts related to the selection criteria previously illustrated. As done previously with search results, relevant papers are gathered in one unique group. Using this stopping criterion, this selection step is concluded when 430 relevant papers are identified.

3.3.4 Additional Search Techniques

The "early stopping" criterion [38] summarized above enables a feasible and comprehensive selection strategy across thousands of contributions; however, it has a natural tendency towards discarding relevant papers. It is also expected that some other relevant papers, not present in the initial set of 83817, are missing because they were not identified by database search. To cover for these limitations, other search techniques are applied in addition to database search. Differing from before, the selection criteria are here applied exhaustively for each document retrieved.

For each of the 430 relevant papers identified in the previous step, a search inside the cited references is performed. In particular, backward snowball sampling [172] is adopted: all papers previously cited by a relevant paper are included in the relevant set whenever they fulfill the selection requirements mentioned above. By doing so, 631 relevant elements are obtained, for a total of 1061.

Reference search enables the identification of prominent contributions frequently mentioned by other authors. A drawback is the introduction of bias towards specific research groups and authors. It can be observed how reference search rewards specific tasks and research fields as they are typically more cited. Therefore, other search techniques are applied to compensate for these facts.

A manual search is performed by inspecting papers published in relevant conferences [167]. These relevant conferences are identified via correlation with other relevant papers and have also been confirmed by experts in the field. The latest 3 editions of each conference are examined to compensate for the over-sampling of dated papers by reference search. 5 more papers are obtained with this method.

To conclude the search phase, the initial guess on IT Operations keywords is improved via analysis of text in the collected documents. Using the relevant paper set as positive samples, a statistical analysis is performed to identify k-shingles (sets of k consecutive tokens) that frequently appear in relevant documents (Table 3.3). In particular, the probability of a document to be relevant, conditional on the set of shingles observed in the available text content, is measured. Values of $k = 1, \ldots, 5$ are chosen. These shingles are used as keywords to

k=1	k=2	k=3
tpc-w, 1.00 (*13*)	defect prediction, 1.00 (*34*)	software defect prediction, 1.00 (*22*)
log-based, 0.92 (*11*)	[work]load prediction, 1.00 (*32*)	disk failure prediction, 1.00 (*8*)
sla, 0.84 (*48*)	software aging, 1.00 (*13*)	failure prediction model, 1.00 (*7*)
stragglers, 0.83 (*10*)	resource allocation, 1.00 (*6*)	cloud resource provisioning, 1.00 (*5*)
vm, 0.83 (*59*)	hardware failures, 0.89 (*8*)	automatic anomaly detection, 0.88 (*7*)

Table 3.3 Examples of *k*-shingles obtained during the mapping study [38] iterative search, with relevance probability (and *total occurrences* in parentheses).

construct new queries along with previously used AI keywords. The collection is here limited to 20 results per query. Thanks to this step, 20 new relevant papers are identified. As a by-product, frequently cited concepts and keywords in AIOps are obtained, which are later useful for taxonomy and classification.

3.3.5 Data Extraction and Mapping

The final result set of the mapping study [38] is composed of 1086 contributions. From these papers, the available information is analyzed to draw quantitative results and answer the posed research questions. Here are described the data extraction process and the analysis techniques employed to gather insights and trends for the AIOps field.

First, the relevant papers are classified according to target components and data sources. Target components indicate a high-level piece of software or hardware in an IT system that the document tries to address (e.g., hard drives for hard drive failure prediction). Target components are grouped into five high-level categories: code, application, hardware, network and datacenter. Data sources provide an indication of the input information of the algorithm (such as logs, metrics, or execution traces).

Observed data sources are grouped according to the terminology convention described in Section 2.2.5 for O&M monitoring data. Only observed data sources are reported. Data sources are categorized in source code, testing resources, system and Key Performance Indicator (KPI) metrics, network traffic, topology, tickets, logs and traces.

The "AI Method" axis denotes the actual algorithm employed, with similar methods aggregated in bigger classes to avoid excessive fragmentation. Table 3.5 presents a selection of FM papers from the result set with the corresponding target, source and category annotation.

Then, this result set is used to infer a taxonomy based on tasks and target goals. The proposed taxonomy is depicted in Figure 3.2. AIOps contributions are divided in FM, the study on how to deal with undesired

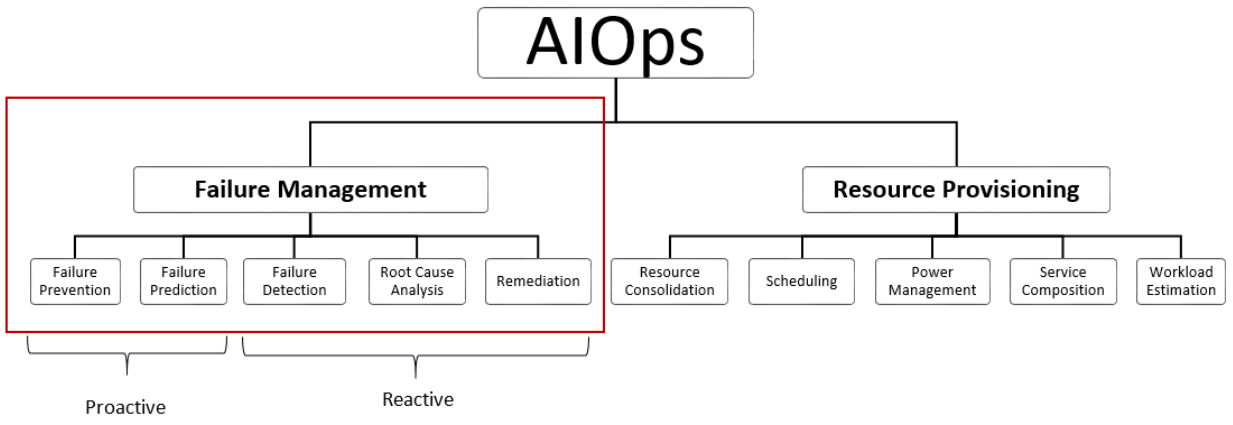


Figure 3.2 Taxonomy of AIOps as observed in the identified contributions [38]. In the red box, the focus of the Failure Management (FM) survey [75].

behavior in the delivery of IT services; and resource provisioning, the study of the allocation of energetic, computational, storage and time resources for the optimal delivery of IT services.

Although the two macro-areas share some common principles and may contribute to the same goals (e.g., allocating sufficient memory to virtual machines or services prevents the occurrence of out-of-memory errors, and achieving effective failure remediation improves utilization of vital resources, such as compute time and operators time), this distinction is useful to decouple the problem of reliability from the efficient allocation of datacenter resources, which is a precondition for running large-scale services effectively.

Within each of these macro-areas, approaches can be further divided into categories based on the similarity of goals. In failure management, these categories are failure prevention, online failure prediction, failure detection, RCA and remediation. In resource provisioning, contributions are divided into resource consolidation, scheduling, power management, service composition, and workload estimation. The analysis of FM (red box of Figure 3.2) is further expanded by applying for this macro-area an additional sub-categorization based on specific problems. Examples of subcategories are checkpointing for failure prevention, or fault localization for root cause analysis (see also Table 3.5).

3.4 Results

This section summarizes the results of the mapping study described above, including the taxonomy of approaches as presented in the original AIOps survey [75].

3.4.1 Mapping Study Results

In this section, the results of the mapping study [38] are discussed. First, the distribution of papers in the taxonomy is analyzed. The left side of Figure 3.3 depicts the distribution of identified papers by macro-area and category. Excluding papers treating AIOps in general (8), the majority of items (670, 62.1%) are associated with FM, with most contributions concentrated in online failure prediction (26.4%), failure detection (33.7%), and root cause analysis (26.7%); the remaining resource provisioning papers support in large part resource consolidation, scheduling and workload prediction. The right side of Figure 3.3 describes the most common problems in FM, i.e., software defect prediction, system failure prediction, anomaly detection, fault localization and root-cause diagnosis, which are the categories with the most attested contributions.

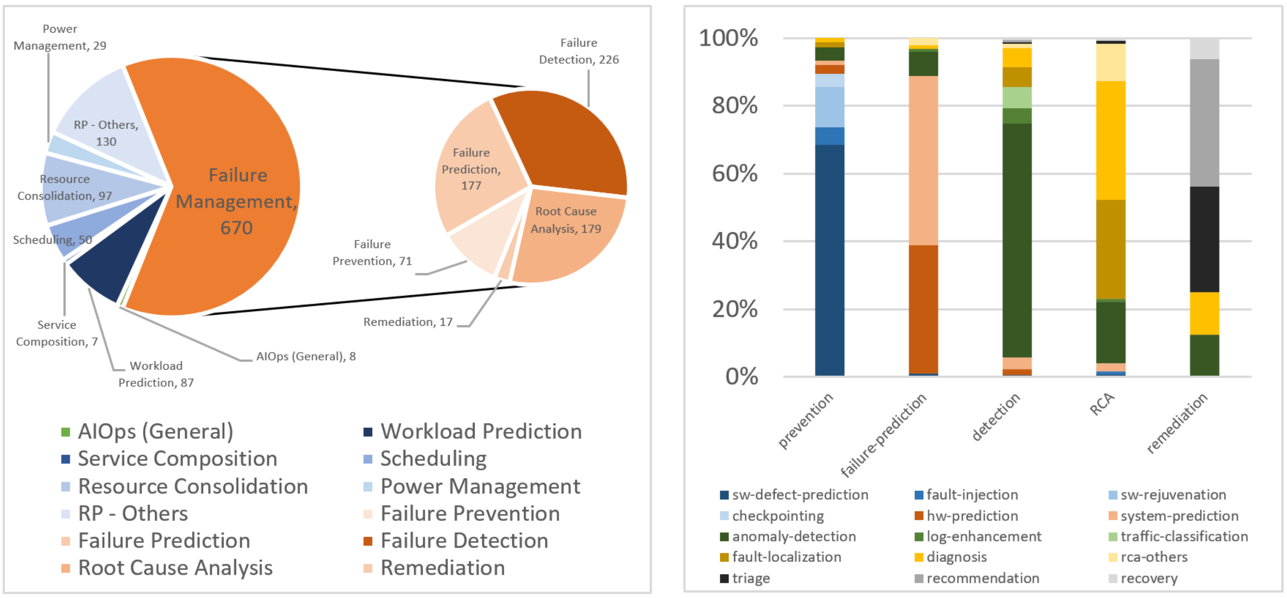


Figure 3.3 Categorization by topic of AIOps contributions [38]. On the left, the distribution of AIOps papers in macro-areas and categories. On the right, the distribution of failure management papers by category in corresponding sub-categories.

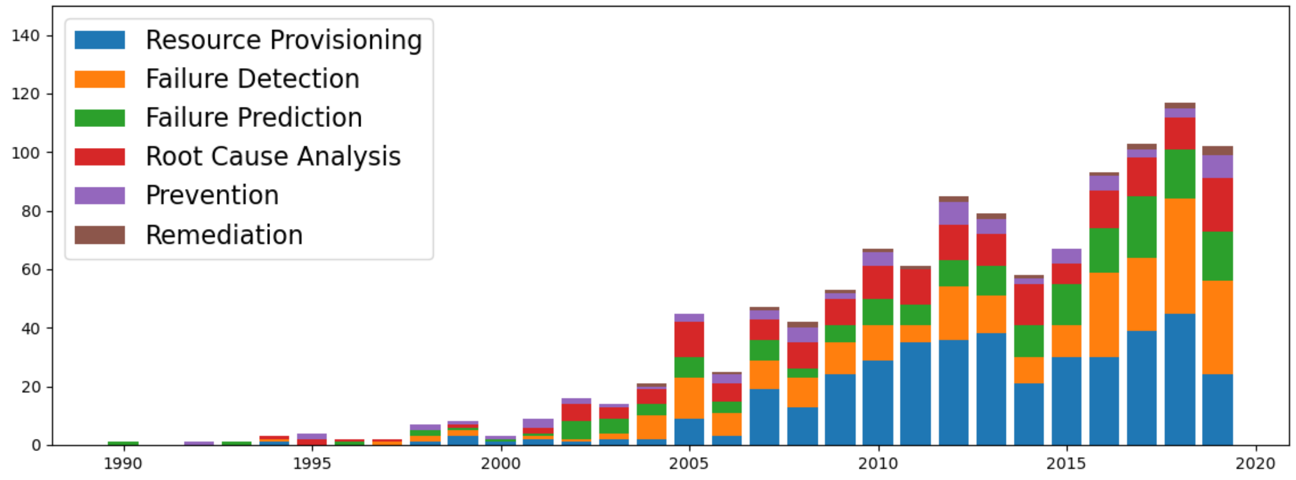


Figure 3.4 Published papers in AIOps by year and categories from the derived taxonomy [38].

To analyze temporal trends present inside the AIOps field, the number of publications in each category is grouped by year of publication. The corresponding bar plot is depicted in Figure 3.4. Overall, a large, growing number of publications in AIOps is measured. It can be observed how failure detection has gained particular traction in recent years (71 publications for the 2018–2019 period) with a contribution size larger than the entire resource provisioning macro-area (69 publications in the same time frame). Failure detection is followed by root cause analysis (39) and online failure prediction (34), while failure prevention and remediation are the areas with the smallest number of attested contributions (11 and 5, respectively).

3.4.2 AIOps for Resource Provisioning

In the next two sections, the individual macro-areas, categories and tasks of the AIOps taxonomy (Figure 3.2) are discussed. As in the original survey [75], the full taxonomy is provided and specific importance is given to the failure management macro-area (Section 3.4.3). Because of the variety of topics covered and the high number of contributions identified (despite the restriction to failure management), the discussion is here limited to those papers that are considered more relevant and innovative from a research perspective.

Resource provisioning optimizes the allocation of the diverse set of resources necessary to provide IT services, such as power, computation time, network bandwidth and virtual memory [75]. Although resource provisioning does not handle failure directly, it is still an important topic in terms of reliability, as the correct allocation of resources is fundamental for the prevention of failures and major outages.

Resource provisioning contributions include resource consolidation, scheduling, power management, configuration analysis, and workload prediction.

Resource consolidation [173–176] ensures current system resources are used efficiently. It relies on the estimation of the suitable set of resources to perform a task or execute a service. Workload can also be forecasted in advance to ensure efficient resource allocation [177–180]. Given a set of estimated resources to assign to each task, and a set of compute nodes, efficient resource consolidation can be achieved by migrating workloads to under-utilized nodes. Migration itself is a frequently analyzed topic [181–184] in terms of how and when to perform out efficiently.

Scheduling [37, 185–187] is the problem of allocation of tasks to compute nodes (such as CPUs and GPUs) from a time perspective.

Configuration analysis approaches [188–190] evaluate how to configure specific cloud systems (microservices, Virtual Machines (VMs), networks) so they can work optimally.

Finally, power management techniques [186, 191–193] incorporate energy and carbon emissions in the variables to optimize for efficient operation of datacenters.

3.4.3 AIOps for Failure Management

Failure Management (FM) is the study of techniques deployed to minimize the appearance and impact of failures [75]. Following an established convention [73, 150, 194–197], we differentiate between proactive and reactive FM approaches (differently from [150], the "resilient approaches" category for AI approaches is not considered). The FM part of the taxonomy is depicted in the red box of Figure 3.2.

Proactive FM approaches operate in anticipation of failures, by reducing the impact and frequency of future failures [75, 198]. These include the improvement of system design choices for failure tolerance, e.g., fault injection/chaos engineering, checkpointing, rejuvenation, software defect prediction, and the determination and response to failures in advance.

Reactive FM approaches operate in reaction to failures and assist human operators to improve Mean Time to Repair (MTTR). The handling of failures requires becoming aware of the failure, a task known as failure detection, which is often cast to an anomaly detection problem, due to deviation of normal behavior caused by observed failures. Additional detection techniques include analysis of incoming network traffic to detect, e.g., malicious attacks, or log enhancement analysis to improve the readability and informativeness of log statements. Then, the resolution of a failure usually requires gathering sufficient information about the specific root cause behind the appearance of the error. The methods working towards this goal belong to the wide umbrella of techniques known as RCA, which includes fault localization, root-cause diagnosis, event correlation, analysis of performance issues and several other tasks. The problem of applying correct response actions for failures is known as *remediation*, and includes techniques for handling repairs for tickets, by aiding routing to correct resolution teams or automatically taking response actions in an intelligent agent setting.

Within each of these five categories, current contributions can be grouped based on target problems (or *tasks*) that a contribution aims to solve (e.g., failure prevention includes software defect prediction, fault injection, software rejuvenation, and checkpointing). Table 3.4 categorizes the FM contributions analyzed in this work by category, task and AI methods used.

Failure Prevention

Failure prevention methods minimize the occurrence and impact of failures, by analyzing the configuration of the system, both in static aspects (like the source code) and dynamic aspects (e.g., the availability of computing resources in physical hosts) [75]. The common goal is to take or suggest preventive actions; however, the strategies to achieve this end goal vary extensively in targets and mode of application.

Preventive operations can be divided in an offline setting and an online setting. In the offline setting, a large predominance of software debugging techniques is observed, usually categorized under the name of Software Defect Prediction (SDP), determined to evaluate failure risks from source code analysis; fault injection techniques are also sporadically present, with the objective of stress-testing the system to gain additional insights and prevent future failures. In the online setting, the observed papers deal with the problem of software aging, categorized under the name of software rejuvenation; or they present checkpointing techniques, to deliver efficient restart strategies in the presence of irreversible errors.

SDP determines the probability of running into a software bug (or *defect*) within a functional unit of code (i.e., a function, a class, a file, or a module). Estimating the remaining density of bugs in a functional unit of code allows software maintainers to prioritize their efforts, concentrating on the most vulnerable modules, files, and methods. A traditional method to identify fault-prone software consists in constructing defect predictors from code metrics. Code metrics are handcrafted features obtained directly from source code, which potentially have the power to predict fault proneness in software. A potential indication of the presence of software faults may also come from the change history of source code. In particular, the code age and the number of previous defects can be indicative to estimate the presence of new bugs [200]. This type of analysis better reflects a software model where changes are continuously applied to a code repository and where new defects are potentially introduced with each new release.

Fault injection is the deliberate introduction of faults into a target working system [134] to evaluate the level of fault tolerance reached. In traditional computer systems, this evaluation represents a validation of the reactive capabilities of a system off-the-shelf. In the specific case of AIOps, fault injection also allows to evaluate the efficacy of externally deployed reactive FM mechanisms. Fault injection can be applied at the hardware-level, to emulate the appearance of faults in physical components, such as hard drives and CPUs, as it is infeasible and costly to induce real faults in this domain. At the software level, fault injections are used to understand the effect of bugs in the behavior of computer software or to model the causal dynamics of one or more software components in a shared-resource environment, such as an operating system and the executing processes.

Software aging describes the process of accumulation of errors during the execution of a program which eventually results in terminal failures, such as hangs, crashes, or heavy performance degradation [133]. Known causes of software aging include memory leaks and bloats, unreleased file locks, data fragmentation, and numerical error accumulation [194]. Several ML techniques have been applied to predict the exhaustion of resources preemptively [133, 137, 215]. Software aging can be contrasted with software rejuvenation [132], a corrective measure where the execution of a piece of software is temporarily suspended to clean its internal state. Software rejuvenation can be performed at the software and OS level. Common cleaning operations include garbage collection, flushing kernel tables, re-initializing internal data structures [137].

Category	Task	AI Method	
Failure Prevention	Software Defect Prediction (SDP)	Linear Models: [199–203] Tree-based: [29, 204–207] Bayesian Networks: [204, 206, 207, 209] Logistic Regression: [202, 205, 206, 211]	Naive Bayes: [29, 204–206] SVM: [207, 208] Others: [199, 203, 210]
	Fault Injection	Clustering: [212, 213] Tree-based: [212]	Linear Models: [213]
	Software Aging and Rejuvenation	Markov Models: [137, 194, 214] Linear Models: [133, 194, 215]	Tree-based: [215]
	Checkpointing	Markov Models: [216–218]	
Online Failure Prediction	Hardware Failure Prediction	Tree-based: [11, 23, 30, 219–221] Neural Networks: [219, 220, 224–227] SVM: [220, 224, 228] Markov Models: [228, 229]	Naive Bayes: [222, 223] Clustering: [130, 222] Logistic Regression: [220] Others: [230, 231]
	System Failure Prediction	SVM: [195, 232, 233] Bayesian Networks: [196, 235] Markov Models: [237]	Autoregressive Models: [196, 234] Neural Networks: [195, 236] Others: [195, 232]
Failure Detection	Anomaly Detection	Clustering: [238–241] Autoencoders: [242–245] Markov Models: [238, 247] PCA: [26, 241, 249] Other Neural Networks: [27, 28, 240, 252, 253]	PCFG: [71, 239] FSM: [240, 246] Tree-based: [26, 248] Others: [235, 238, 248, 250, 251]
	Internet Traffic Classification	Neural Networks: [254, 255] Naive Bayes: [257]	SVM: [256]
	Log Enhancement	Tree-based: [258]	Others: [259]
Root-cause Analysis	Fault Localization	Graph Mining: [260–262] Others: [241, 247, 265–269]	Search: [262–264]
	Root-cause Diagnosis	Pattern Matching: [238, 270] Bayesian Networks: [235, 271]	Others: [71, 271–274]
	RCA - Others	Clustering: [275–277] Logistic Regression: [32, 275]	Others: [32, 276, 278]
Remediation	Incident Triage	Markov Models: [279]	Bayesian Decision Theory: [280]
	Solution Recommendation	Text Analysis: [117, 281, 282]	Similarity-based: [282]
	Recovery	Markov Models: [272]	

Table 3.4 FM papers analyzed in the AIOps survey [75] by category, task, and AI method.

A concept linked to software rejuvenation is checkpointing, i.e., the continuous and preemptive process of saving the system state before the occurrence of a failure. Similar to software rejuvenation, checkpointing tolerates failures by occasionally interrupting the execution of a program to take precautionary actions. Different from software rejuvenation, during checkpointing the interruption period is used to save the internal state of the system to persistent storage. In case a fatal failure occurs, the created checkpoint file can be used to resume the program and reduce failure overhead.

Online Failure Prediction

Online Failure Prediction (OFP) methods are specialized in the prediction of computer system failures in real-time [75]. OFP identifies future runtime errors by assessing the current state of the system [73].

How far in time these errors can be foreseen depends on the lead time of the predictor, i.e., the time between the prediction and the instant when the failure occurs; the validity of the prediction information also depends on the prediction time, which is the length of the time window where the failure may occur. A more extended and formal description of OFP time requirements is presented in Section 4.3.1.

OFP has been frequently applied for hardware failure prediction of components such as hard drives, memory and switches, as well as servers and software systems (see also Section 4.1.2).

Future system availability can also be estimated through symptomatic evidence and dependency modeling assumptions at the software level. Past approaches for system failure prediction are mostly based on the observation of logs, which constitute the most frequent data source, KPIs and hardware metrics, which are typically used in shorter prediction windows and are more frequently associated with the failure detection problem as well. System failure prediction is applied on different abstraction levels depending on the target software component under investigation (job, task, container, VM, or node).

Failure Detection

Failure detection is the process of collection of symptoms, i.e., observations that are indicative of failures.

The detection of failures via monitoring operations can be a complex and tedious task for human operators. Chen et al. [71] report how in the administration of a commercial website, 75% of the recovery time was spent on average for detection. The automatic discovery of performance problems and errors allows operators to dedicate less time to identifying service-related problems while providing insights on which failures must be prioritized in the diagnosis step based on the frequency observed in the detection phase. Automated failure detection is based on a variety of monitoring tools, ranging from simple print statements (which constitute the fundamental unit of system logs) to more complex instrumentation techniques or entire frameworks.

According to the quantitative results of the mapping study [38], failure detection is treated as an anomaly detection problem in the large majority of contributions related to IT system management. Anomaly detection is a multidisciplinary task that deals with finding patterns in data that do not conform to the expected behavior [144].

Because obtaining labeled examples is time-consuming, anomaly detection systems typically rely on unsupervised learning. Three most prominent techniques are used in such context: clustering [238, 239], dimensionality reduction [26, 241] and neural network autoencoders [242–245].

A task connected to network failure detection is Internet Traffic Classification (ITC) [256, 257]. ITC allows categorizing packets exchanged by a network system to identify network problems, to optimize resource provisioning and improve Quality of Service (QoS). It can be applied to analyze the local network flow, the incoming server requests from the outside, or the outgoing response.

Another task connected to failure detection is log enhancement [258, 259, 283, 284]. Its goal is to improve the quality and expressiveness of system logs, which are frequently used for detection and diagnosis tasks by IT operators and AIOps algorithms.

Root Cause Analysis

Root Cause Analysis (RCA) is the process of inferring the set of faults that generated a given set of symptoms [75,162]. In a complex and distributed system, it is first required to isolate the responsible component or functional subsystem, a task known as fault localization. Only then an analysis of the possible error sources can be performed with root-cause diagnosis.

Fault localization is about identifying the set of components (devices, network links, hosts, software modules, etc.) interested by a fault that caused a specific failure [241, 247, 260, 261, 263–269]. Several approaches [263,264,269] address fault localization by correlating abnormal changes in KPI values to particular combinations of attributes (representing e.g., geographical regions, ISPs, hosts, buckets, etc.). These combinations of values are then symptomatic of a fault for that particular corner case. Software Fault Localization (SFL) [261, 265–268] analyzes software components through source code analysis. An SFL system typically returns a report containing a list of suspicious statements or components.

Root-cause diagnosis [235, 238, 270–272, 274] identifies the causes of behavior leading to failures, by recognizing the primary form of fault. For this reason, it is typically treated as a classification problem. Due to the inherent complexity and interdependency between components in software systems, it is considered a challenging task.

Several software tools can assist operators and developers while investigating detected problems and can, therefore be seen as ancillary resources for the main root-cause analysis task [32, 276–278]. These tools include information retrieval mechanisms to quickly gather evidence of recurrent problems, like their relative frequency; or dependency models for distributed systems used to understand causal relationships between events and/or components to accelerate future diagnoses.

Remediation

Thanks to the problem-specific knowledge gathered during the diagnosis step, like the identification of root causes or the isolation of a faulty component, it is possible to initiate a sequence of automatic repair actions, which are here described as remediation.

Remediation has experienced fewer scientific contributions linked with AI compared to the prevention, prediction, detection, and diagnosis tasks. This is possibly because, once the nature of the underlying problem has been clarified through diagnosis, the recovery steps are almost immediately identifiable and attainable without resorting to complex prediction models.

Remediation approaches are often linked with certain concepts of service desk management, such as ticket routing or ticket solution recommendation. Incident triage [279,280] is the step in problem resolution dealing with categorizing a reported problem. The purpose of triage is often the assignment to the correct expert resolution group. Triage may also be used to select a suitable diagnosis and remediation algorithm.

Solution recommendation approaches [117,281,282] provide methods for recommending recovery actions to occurring problems. Most solutions are based on past incident history and rely on the annotation of solutions in a previous resolution window. Recovery approaches take direct and independent actions toward the resolution of a diagnosed problem [272].

3.5 Summary

In the previous section, many AIOps contributions to deal with failures in large-scale computing environments have need described. The AIOps topic has been explored starting from definitions to capture a precise characterization of the topic in terms of goals, sources, and methods. Several contributions discussed in the survey study [75] across all identified categories, data sources, and target components, have been summarized.

In this final section, the big picture of the current status of AIOps failure management is analyzed. These observations are also used to examine the currently open challenges and suggest future directions for research.

AIOps has shown a steadily growing trend in the last years, manifested both in the number and variety of contributions present. In the last five years, at least 100 contributions have been proposed on a yearly average. It is expected that the field will continue its growth, due to increasing demand for reliability and efficiency in large-scale computing systems. The evolution of cloud technologies (e.g., in virtualization, monitoring tools) will provide large space for future improvements and the experimentation of new techniques. In order to fulfill these expectations, the field must be able to provide a solid ground for experimentation, based on a more formal standardization of problems and a stronger attitude toward benchmarks, needed for comparison and evaluation of the results achieved. Efforts in creating standard problems and benchmark datasets would therefore be rewarding.

The analysis of the topics and tasks in the current AIOps landscape, as observed and described in Section 3.4.1, equally allows to investigate possible future directions. Firstly, Table 3.5 shows how the majority of works utilize only one or few types of data. Multi-modal approaches, able to take advantage of different data sources, may prove more effective and robust to new observations, thanks to the increase in system visibility.

It has also been observed how some areas of failure management have experienced less scientific interest compared to others. A clear example is the recovery task which, although a fundamental and concrete step to deal with failures, still presents a minor group of contributions.

A similar consideration applies to failure prevention, where all the contributions are concentrated around a few tasks (four subcategories were presented). However, failure prevention can be performed in many other ways, some of which are still to be explored. Currently, the majority of approaches for failure prevention are applied online and concentrate exclusively on the current-future state characteristics. Introducing assumptions and information about the system working principles may set the ground for much more actionable insights. For example, model-based prevention would allow operators to estimate in advance the risks associated with particular actions, such as a canary release or a server shutdown.

Online failure prediction has seen a significant number of contributions, which however concentrate in a limited number of target components and data sources. In general, it can be concluded that proactive failure management requires additional exploration in new methods and problems.

In addition, the advent of virtualization technologies requires new research focusing on specific targets (e.g., hypervisors, virtual machines, containers, etc.), or addressing new tasks, such as hypervisor anomaly detection, container failure prediction and so on.

Finally, the application of novel AI approaches may prove beneficial to advance AIOps. In the decade, the rise of deep learning methods (such as Large Language Models (LLMs)) has translated into a variety of new approaches for failure prediction, anomaly detection and root cause analysis. Taking advantage of these recent advances may extend the actionability of AIOps solutions and improve their effectiveness in complex tasks, such as RCA.

Paper(s)	Data Sources									Targets					Category
	Source Code	Testing Resources	System Metrics	KPI/SLA data	Network Traffic	Topology	Incident Reports	Event Logs	Execution Traces	Source Code	Application	Hardware	Network	Datacenter	
[29, 199–204]	●									●					Software Defect Prediction
[205–211]	●									●					Software Defect Prediction
[212, 213]	●	●								●					Fault Injection
[133, 137, 215]			●								●			●	Software Rejuvenation
[194, 214]			●	●							●			●	Software Rejuvenation
[216–218]											●			●	Checkpointing
[11, 30, 219–221]			●									●			Hardware Failure Prediction
[222–224, 227, 230]			●									●			Hardware Failure Prediction
[23, 130, 225, 228]			●									●			Hardware Failure Prediction
[226]			●									●		●	Hardware Failure Prediction
[231]			●					●				●			Hardware Failure Prediction
[229]								●				●	●		Hardware Failure Prediction
[234]			●								●			●	System Failure Prediction
[235]			●	●							●				System Failure Prediction
[236]			●	●							●			●	System Failure Prediction
[196]			●			●					●			●	System Failure Prediction
[232]								●			●			●	System Failure Prediction
[237]								●			●	●	●	●	System Failure Prediction
[195, 233]								●			●				System Failure Prediction
[26]	●							●						●	Anomaly Detection
[243–245]			●									●		●	Anomaly Detection
[242, 248]				●							●				Anomaly Detection
[238]			●	●							●			●	Anomaly Detection
[241]					●	●							●		Anomaly Detection
[249]					●	●					●		●		Anomaly Detection
[27, 28, 240, 246, 252, 253]								●			●				Anomaly Detection
[251]								●	●		●			●	Anomaly Detection
[239]									●		●			●	Anomaly Detection
[71, 250]									●		●				Anomaly Detection
[254–257]					●								●	●	Internet Traffic Classification
[258, 259]	●							●						●	Log Enhancement
[261, 262, 265–268]	●	●								●	●				Fault Localization
[247]			●			●						●		●	Fault Localization
[263, 264]				●										●	Fault Localization
[269]								●						●	Fault Localization
[260]				●	●							●	●		Fault Localization
[274]	●							●			●				Root-cause Diagnosis
[270]	●		●	●							●				Root-cause Diagnosis
[271]					●	●							●		Root-cause Diagnosis
[273]									●		●			●	Root-cause Diagnosis
[275]		●				●					●				RCA - Others
[32, 276]			●	●							●			●	RCA - Others
[277]								●			●			●	RCA - Others
[278]									●		●			●	RCA -z Others
[279, 280]							●				●			●	Incident Triage
[281, 282]							●				●			●	Solution Recommendation
[117]							●	●			●			●	Solution Recommendation
[272]			●	●							●			●	Recovery

Table 3.5 FM papers analyzed in the AIOps survey [75], grouped by employed data sources, targets, and categories.

4 Infrastructure-level Proactive Failure Management

This chapter discusses the application of proactive techniques for Failure Management (FM) at the infrastructure layer of cloud systems. A background on the infrastructure layer is introduced in Section 4.1. Related work on infrastructure-level proactive failure management is summarized in Section 4.2. Section 4.3 provides a general framework for evaluating Online Failure Prediction (OFP) models. Section 4.4 introduces the problem of hardware reliability, motivating its importance and challenges and discussing the most frequently affected components. The last three sections (4.5, 4.6, and 4.7) describe real instances of hardware failure prediction problems: Section 4.5 hard drive failure prediction, Section 4.6 Dynamic Random Access Memory (DRAM) failure prediction, and Section 4.7 describes optical transceiver failure prediction.

4.1 The Infrastructure Layer

As already described in Section 2.1.3, the infrastructure layer of a cloud system is responsible for providing processing, storage, network, and other computing resources [39, 55, 56].

In other cloud models, the infrastructure is not directly accessible to customers, which however rely on its correct operation for executing their computing workload according to their selected cloud offering. The infrastructure is therefore always, directly or indirectly, offered as a service to the customer and must adhere high level of availability and scalability, as specified in the Service Level Agreeement (SLA).

The next sections describe the structure of the infrastructure layer and what are the consequences in terms of Operations & Maintenance (O&M) and failure management.

4.1.1 Structure of the Infrastructure Layer

Virtualization allows multiple users to operate on the same hardware, increase server utilization, and isolate customer workloads. Virtualization relies on a Virtual Machine Manager (VMM), or hypervisor, to create and managed virtualized instances over the hardware infrastructure. The exact implementation of the infrastructure layer depends on the type of virtualization employed [45]. A typical infrastructure-level offering comprises the following layers of abstraction:

- the *hardware* layer, composed of all physical processing, storage, and networking resources;
- the (optional) *host Operating System (OS)* layer, consisting of the operating system hosting the hypervisor (this layer is not present in bare-metal virtualization);
- the *virtualization* layer, consisting of the hypervisor hosting several virtual instances, which represent the customer workload, composed of a *guest OS* and the applications running on it. In the case of OS-level virtualization this layer corresponds to the container engine;

4.1.2 Infrastructure-level Failures

Even if faults may occur on each of these infrastructural layers, infrastructure failures are caused predominantly by hardware faults [285] and to a lower extent from software faults in the virtualization and OS layers.

Hardware faults may be due to the aging process of components or due to external factors, such as contamination, electrostatic discharge, radiation and mechanical damage. Some hardware components may also be dead-on-arrival or be subject to high infant mortality rates, which are indication of manufacturing issues.

Software faults are primarily due to software bugs in the implementation of OS and virtualization functionalities. These include memory leaks, software hangs, freezes and crashes, numerical and logical errors.

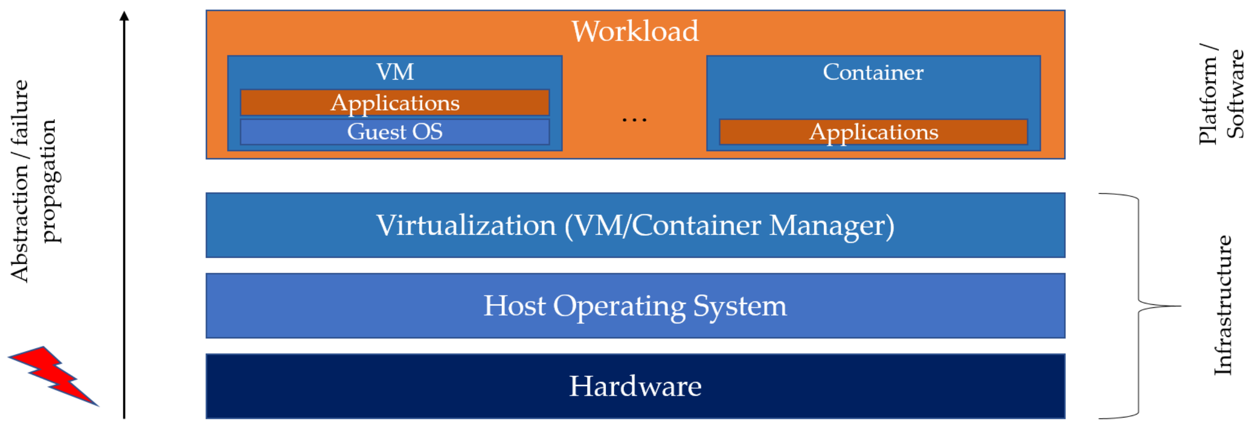


Figure 4.1 Structure of the infrastructure layer, composed of physical hardware, host OS, and virtualization sub-layers (hypervisor), which allow to host multiple tenants with separate virtual resources (e.g., Virtual Machines (VMs) or containers). If not properly handled, failures propagate from the lower layers upwards to all the upper layers.

Failures tend to propagate up in the abstraction layers. Due to this propagation and the complexity of the infrastructure model, infrastructure failures may be complex to detect and trace back to their root cause. In the absence of fault tolerance systems, failures will also propagate to the customer side of the cloud stack model (Figure 4.1). For instance, a fault originating in the hardware layer can manifest as a failure at the OS and virtualization layers, or a OS level fault will be visible at the virtualization layer and in the guest OS layer.

4.1.3 Remediation Actions

Several remediation actions can be undertaken to recover from infrastructure failures. [74,286]. They include:

- component replacement, in the case of hardware faults (in storage, network and server devices);
- Error Detection And Correction (EDAC), i.e., techniques recognizing and restoring digital data that might be corrupted during transmission due to channel noise, such as Error Correction Code (ECC) [287]. These techniques are applicable for systems applying transmission protocols such as network interfaces, Integrated Memory Controller (IMC), etc.
- restart, either at the host, VMM or OS level. This is applicable for runtime-related errors, such as memory leaks, software crashes, unresponsive peripherals, etc.;
- migration, i.e, the action of moving a virtual instance (e.g., VM) from one physical hardware environment to another. Migration can either be *live migration*, without stopping the virtual instance and disconnecting the client or application, or cold migration, i.e., the migration of a virtual instance in a suspended state;
- change in allocation policies, e.g., VM or container orchestration, which may prevent allocation to a faulty node or component.

These actions are traditionally applied in reaction to failures. This, however, has the same disadvantages which have been frequently mentioned for reactive approaches, namely the unpredictability of failures, the difficulty to completely eliminate all damage, and the high issue visibility and data correlation required. It is therefore valuable to investigate in which scenarios proactive failure management can be applied at the infrastructure level.

4.1.4 Techniques for Infrastructure-level Proactive Failure Management

Based on the results of the AIOps survey discussed in Section 3.4.3, the proactive FM techniques which are applicable for infrastructure failures are:

- *Online Failure Prediction (OFP).* It may be applied directly to the hardware, to proactively suggest component replacements o workload migration; it may be applied on the virtualization manager (VMM or container engine), or at the VM-level, to trigger a live migration towards a healthy node;
- *checkpointing*. It allows to recover from a previous state and mitigate failure damage. Checkpointing can either be at the software level or implement full system snapshots (similar to the case of VM migration). In the case of failure, a restart and a checkpoint load are triggered as mitigation action;
- *server rejuvenation.* It allows to refresh the system state and prevent fatal failure by resource exhaustion. It can consist of predictive resource exhaustion systems or pre-established refresh policies. The state refresh is the mitigation action itself, no other action is required;
- *fault injection.* Offline method for improving reliability design. It can either result in structural design changes or in the implementation of new mitigation strategies or new resource allocation policies (e.g., for VMs). No online mitigation action is taken.

In order to be applicable, proactive techniques must provide sufficiently high protection against failure to justify their overhead. In the case of OFP, a failure prediction/repair cost model can be devised to estimate the applicability of repairs, based on assumptions of the classifier accuracy metrics, component failure rates and repair costs. An example of such a cost model is presented for the Online Failure Prediction Framework in Section 4.3.

Similar considerations apply to other proactive FM techniques. Checkpointing and software rejuvenation have a runtime cost (and a storage cost, for checkpointing) which must be compensated by the reduction in failure rate and recovery time. Too complex or compute-intensive algorithms may impose an unacceptable overhead to render such actions beneficial.

The costs and benefits of fault injection are more difficult to evaluate, as it deeply relies on simulated environments, where the real-world consequences of failures can be avoided. It, however, requires time and resources to evaluate failure scenarios, which must be justified by actionable insights. Moreover, if the simulation assumptions do not correspond to the real-world system, the derived design improvements may prove highly detrimental and cause more harm than necessary.

In summary, to mitigate the impact of infrastructure failures on customers, it is important to proactively address them by:

- increasing issue visibility in all these layers, through the monitoring of resource usage, errors, and performance metrics;
- implementing fault prevention techniques, such as performing sanity checks, state refreshes, or checkpoints; deploying online failure prediction systems, to perform recovery and mitigation techniques with sufficient time in advance;
- dealing with occurred failures directly at the origin layer of the problem via fault localization and root-cause diagnosis, to minimize the occurrence of future related failures.

4.2 Related Work

The discussion in this chapter focuses on past proactive methods for dealing with infrastructure-level failures. Techniques can either focus on the hardware, the host OS layer or the virtualization layer, as described above.

4.2.1 Hardware Layer

A vast variety of contributions has been proposed to address hardware failures, by observing the current device state to detect anomalies or predict future failures in advance. Main interested components include Hard Disk Drives (HDDs) and other storage media, DRAM chips, and network devices (transceivers, switches, routers, . . .).

Hard Disk and Solid State Drives

Several studies have investigated the failure characteristics of HDDs and Solid State Drives (SSDs) to identify common patterns [75].

Pinheiro et al. [288] conducted a large-scale study on over a hundred thousand disk drives used in production by Google and varying in storage size, speed, and manufacturer. The results of this study could not identify any consistent correlation between failure rate and high temperature or high utilization levels. Some Self-Monitoring Analysis and Reporting Technology (SMART) features were shown to correlate well with a higher failure rate. However, SMART metrics were also shown to be likely insufficient to predict all single-disk failures, as the majority of failed drives did not manifest any SMART error signal before faults. A predictor based solely on SMART attributes is therefore likely to have a good specificity but a low recall, unless additional features are introduced. SMART data is still considered useful to evaluate reliability and risk trends inside a disk drive population.

Murray et al. [130] test the applicability of several Machine Learning methods using a sliding window approach, where the last *n* samples constitute the observation for predicting an imminent failure. Naive Bayes, Support Vector Machines (SVMs), and Naive Bayes Expectation-Maximization (EM) are implemented and compared. Features are selected from SMART data using statistical relevance tests. SVMs achieve their highest performance with a 50.6% recall and 100% precision. The approach was tested on a dataset composed of 369 drives (with an approximate 50/50 split), which is then also used in a work by Wang et al. [230] where an online, similarity-based detection algorithm is presented. Relevant SMART features are selected via Minimum Redundancy Maximum Relevance (mRMR), then the input data is projected into a Mahalanobis space constructed from the healthy disk population so that faulty disks deviate more from the distribution. Faulty disks are again recognized with a sliding window approach, so that when the mean deviation inside a window appears anomalous an alarm is raised, using four different statistical tests. This approach improves, at 0 false-positive rate, the detection rate up to 67% and shows how for 56% of the faulty cases it was possible to intervene with an advance of at least 20 hours before the failure.

Zhao et al. [228] treat SMART data as a time series, arguing for the importance of temporal information. Their approach employs Hidden Markov Models (HMMs) and Hidden Semi-Markov Model (HSMM) to estimate the sequence of likely events from the disk metric observations, which are obtained from a dataset of approximately 300 disks (2 thirds are healthy). One model is trained from healthy disk sequences, one from faulty disk sequences. At test time, the two models are used the estimate the sequence log-likelihood and the corresponding class is selected based on the highest score. By combining the HMM approach with an SVM, they claim to obtain a recall of 52% at 0 false alarm rate.

Comparable results are obtained by Zhu et al. [224], where MultiLayer Perceptrons (MLPs) and SVM models are constructed and trained on an in-house dataset of SMART data comprising 23395 drives (433 faulty). Assuming a 12-hour recovery window, their SVM model obtains a failure detection rate of 68.5% with a 0.03% false alarm rate. The neural network method achieves far higher detection rates (94.62–100%), at the expense of a higher false alarm rate as well (0.48–2.26%) and it is therefore indicated for monitoring with the highest reliability requirements. The same SMART dataset is used by Xu et al. in a paper [225] that introduces Recurrent Neural Networks (RNNs) to hard drive failure prediction. Similarly to [228], the method can analyze sequences directly and takes advantage of the temporal dimension of the problem to model the long-term relationships. Differently from previous approaches that apply binary classification, the model is trained to predict the health status of the disk, providing additional information on the remaining useful life of disks. On the failure prediction task, however, the approach still outperforms the other evaluated models in terms of detection rate (96.08–97.78%) and false alarm rate (0.004–0.03%).

In a work by Li et al. [219], two new evaluation metrics (migration and mis-migration rate) are introduced to measure the efficiency of data migration concerning forecasted faults. RNNs and Gradient Boosted Decision Trees (GBDTs) are implemented to accomplish three tasks: predict faulty disks, measure the rate of wrongful and missed migrations performed using the information of the classifier, and estimate the residual life of the disks, by predicting the risk level of each disk (1 to 5 for faulty disks, 6 for healthy ones). For residual life prediction, results are compared using the newly defined metrics at variable migration rates. The

RNN approach shows a higher faulty level prediction accuracy (27.02–39.90%) and the GBDT model better protection from data loss with a higher successful migration rate (84.91–87.54%).

Mahdisoltani et al. [220] tackle failure prediction in storage media at the sector level. Their method employs a few SMART features as target prediction variables rather than explanatory variables. They experiment both with HDD and SSD data, with five different Machine Learning approaches. For HDD data the analysis illustrates detection rates of sector errors similar to the ones of traditional disk failure prediction (70–90% at 2% false-positive rate). In SSDs, where Random Forests (RFs) obtain the best comparative results, the prediction is not as promising (50–60% detection rate at 2%). SSD failure characterization is also the focus of Narayanan et al.'s work [221], a large-scale study conducted on 2.5 years of production data and covering over 500,000 solid-state disks from different Microsoft datacenters. SSD failures are correlated to datacenter-, workload-, and device-level features via statistical learning. The best and only reported prediction model (random forest) obtains a precision of 87% and a recall of 71%. Features directly representing underlying problems (such as count of data errors and reallocation sectors), number of NAND writes and workload factors are reported as the most important discriminators of failures. According to the same authors, the classification rules obtained by the analysis enable the identification of likely-to-fail devices with sufficient advance to take preventive actions, especially in the case of issues heavily dependent on the workload.

All previous approaches are used in an online setting after an initial offline preparation step. This poses the problem of how to integrate additional data, especially in those cases where the scarcity of positive training examples imposes an online learning approach, able to update the characteristics of faulty disks as soon as new failures appear. To this end, Xiao et al. [23] propose the use of online RFs, a model able to evolve and behave adaptively to the change in the data distribution via online labeling. The approach is tested on a dataset covering over 10000 disks, where the results show how the algorithm can increase performance over 20 months, reaching detection rates of 93% and more with reasonably low alarm rates (0.73–0.76% in the offline setting). The prediction performances are comparable to a RF and outperform the other approaches tested (SVMs and decision trees).

Lu et al. [289] study the effect of different metrics and utilize neural networks for disk failure prediction. They integrate monitoring SMART data with disk performance metrics, such as disk throughput/latency; server-level metrics, e.g. CPU usage and page in/out activities; and location information, in the form of physical disk proximity. Different Machine Learning (ML) models, including Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM)-based networks, are evaluated on a dataset of 380k disks. The newly introduced information is shown to provide an increase in classification accuracy.

Luo et al. [290] propose the Neighbor-Temporal Attention Model (NTAM), which utilizes information of neighboring disks and an attention-based mechanism to capture temporal dependencies in SMART metrics, which are used in addition to static system and driver information. They also introduce Temporal Progressive Sampling (TPS) to deal with the data imbalance problem, by varying the lead time to failure in the collected samples. The results, evaluated by comparison to state-of-the-art approaches on two large-scale Azure production datasets, achieve a precision of 83–84% (7%) and a recall of 64% (+9%).

DRAM and Memory

Several empirical studies have treated failures in memory components [291–295]. These works on correlation analysis and memory failure modes provide a theoretical foundation to the development of memory failure prediction algorithms [296].

Works in a second category proposed methods to detect and predict memory failures. Costa et al. [231] investigate the occurrence of main-memory errors for High Performance Computing (HPC) applications. They propose and evaluate methods for temporal and spatial correlation among memory failures. Temporal correlation analyzes logs to estimate the prior rate of errors which, integrated with timing information, allows to estimate the remaining number of errors at the job level. Spatial correlation measures the probability to experience a failure after observing errors in adjacent bits. A memory migration approach, based on the available information at runtime, is presented and evaluated. The results show how 63% of memory-induced failures could be avoided thanks to the prediction and migration mechanisms deployed.

Giurgiu et al. [297] propose the first predictive model for DRAM failure prediction. Their approach utilizes Correctable Errors (CEs) obtained from event logs and sensor metrics of 50k IBM machines to predict future UnCorrectable Errors (UCEs). They devise ensemble ML approaches, such as RFs to classify sliding windows of memory features as faulty or healthy. They focus on enforcing low False Positive Rate (FPR) on predictions, achieving a final result of 96% precision. The approach, however, suffers from low recall and can therefore predict only a limited number of DRAM failures (<30%).

Boixaderas et al. [298] present an UCE prediction system including a cost-aware model. They collect CE features at the Dual In-line Memory Module (DIMM), socket, and node level from 2 years of production logs. Utilizing RFs and feature engineering, their approach reduces the lost compute time due to DRAM failures by up to 57% in a real HPC environment.

An online learning technique for anticipating DRAM failures is presented by Du et al. [299]. DRAM failures are predicted at the micro-level (i.e., cell, row, and column), by comparing current and historical memory states using a kernel function and by incorporating a model of fault propagation at the micro-level. They further present a weighted ensemble approach [300] based on row, column, and bank fault predictors, which outperforms their previous work in terms of F1-score (up to 69%) and cost reduction (up to 45%). Li et al. [301] also develop a new CE-based risk indicator for UCE prediction by applying error bit patterns, manufacturer information and part numbers.

Wang et al. [302] collect workload features from a real-world commercial datacenter. They propose to use both workload metrics (e.g., memory read/write bandwidth) and micro-level features, such as cell access patterns, to train tree-based ML models. Their comparative results show the benefits of the proposed features for the DRAM failure prediction task.

Zhang et al. [303] conduct an empirical failure analysis and spatiotemporal feature engineering on Machine Check Exception (MCE) logs and memory-related events, to predict and mitigate node unavailability in large-scale cloud infrastructures. Evaluation results show a reduction in node unavailability time up to 69% by using an ensemble of ML and rule-based approaches for failure prediction.

Optical Transceivers

A limited number of previous contributions has focused on failure detection and prediction in optical transceivers. Mendoza et al. [304] use Rx/Tx power from Small-Form Pluggable (SFP) monitoring data to detect connectors contaminated with dust and fingerprint oil. Their results show how fingerprint oil contamination cannot be detected due to accuracy limitations, in addition to presenting a general detection rule for the detection of dust contamination.

Chakravarty et al. [305] describe the optical monitoring framework of Facebook datacenters. They observe the presence of transient failures and gradual degradation leading to faults, pinpointing temperature instability and infant mortality as contributing factors. They also suggest the estimation of degradation slopes for failure prediction.

Wang et al. [306] use five dynamic Digital Diagnostic Monitoring (DDM) attributes (voltage, temperature, bias current, Rx/Tx power) to construct a failure predictor based on Double Exponential Smoothing (DES) to forecast future metric values, which are classified by means of a SVM model to predict failures. They train their models using 35-day observation windows and label faulty modules based on the next 9 days. The approach achieves an accuracy of 95.6%; however, the employed dataset utilizes a class-balanced distribution, which is not representative of the real failure distribution and does not allow to draw accurate insights about failure patterns in optical transceivers.

Li et al. [307] utilize optical DDM metrics to study massive failures, i.e., large groups of transceivers that experience common and related misbehaviors. They monitor the presence of massive failures by estimating parameters of Gaussian distributions from the transceiver population.

Tanaka et al. [308] propose an optical link diagnosis system for predictive maintenance in optical networks. The monitoring data is collected using white-box transponders and it is used to estimate fiber bending issues.

Liu et al. [309] use dynamic DDM attributes and additional gradient features to predict optical failures online. They train gradient-boosted trees using single time samples as input; however, a numerical evaluation and feature importance insights are not presented.

Others

A minor group of contributions focusing on failure prediction for other components is present in the past scientific literature. Ma et al. [223] elevate the discussion on storage reliability to the level of Redundant Array of Inexpensive Disks (RAID) groups. They model the failure probability of a vulnerable RAID group using a Naive Bayes assumption, where the failure of whole disks inside the group is independent of the others and is estimated from metrics such as reallocated sector counts. A statistical model is built from data collected on 5000 RAID groups. Results show how the present in-place mechanisms were able to prevent a vast majority of consecutive failures (98% for triple failures).

Davis et al. [226] present FailureSim, a Cloudsim-based [310] simulator for status assessment of hardware in cloud datacenters using deep learning. MLPs and RNNs are used to assess 13 different host failing states, each associated with a specific component (CPU, memory, I/O, etc.). The system, tested by assigning a variable workload to a scalable number of VMs, can identify 50% of failing hosts accurately, and predict 89% of future failures before their occurrence.

Switch failure prediction is addressed by Zhang et al. [229,311]. Their method utilizes system log history to extract log templates to correlate with faulty behavior. Their extraction method and other similar approaches are compared on the extraction tasks, and the templates so obtained are used to train a HSMM. The proposed model, which shows the best performance figures, achieves 32% precision and 95% recall, demonstrating high sensitivity to clear fault-signaling messages, but low precision overall. Because of the high interpretability of mined templates, this approach can also be used for failure diagnosis.

Zheng et al. [227] take a different direction by treating failure prediction as a regression problem. Through the use of LSTM neural networks, which are able to model long term-dependencies, they estimate the Remaining Useful Life (RUL) of system components. They experiment with three RUL public datasets and compare their method with other ML approaches in terms of Root Mean Squared Error (MSE), showing how their approach obtains the best RUL prediction performance (RMSE=2.80, 54.47% improvement over CNNs).

4.2.2 OS & Virtualization Layer

VM and VMM Failures

Jindal et al. [312] propose Indirect Anomaly Detection (IAD) to detect VMM failures online using VM resource utilization. The VM utilization is collected in time series, which are analyzed by comparison across the aligned time steps. If the same metric shows an anomalous change across most VMs residing on the same hypervisor for the same time steps, the behavior of the hypervisor itself is deemed anomalous.

Cerveira et al. [313] study the implications of hypervisor failures in a virtualization setting to derive insights on lead time (see Section 4.3.1) for prediction of software and hardware faults. Hypervisor failures are investigated through fault injection techniques to produce realistic failure data for the assessment of the distribution of lead time of the target system. The results show that failure prediction in virtualized systems is better suited to predict failures caused by software faults than failures caused by hardware faults, which have shorter lead times leading to almost instantaneous crashes.

ReHype [314] is a mechanism for preserving VM states during hypervisor failures. ReHype identifies corrupted VMM states through the detection of inconsistencies between VMs, VMM, and the hardware. The VMM state is repaired by micro-rebooting the VMM and re-applying a previously saved state. The approach is tested by modifying the Xen hypervisor [48] with minimal overhead. During the evaluation performed through fault injection, ReHype is able to detect and recover from 90% of hypervisor failures.

Rawat et al. [315] propose a time-series stochastic model for VM failure prediction. VM execution is monitored and several metrics, carrying VM health status information and usage patterns, are collected. These metrics are then processed as time series, using AutoRegressive Integrated Moving Average (ARIMA) estimators. The technique described can also be applied for the prediction of security issues in network services.

Server Failures

Several previous works have applied ML models for predicting failures, including resource exhaustion, at the server (or *node*) level. Such techniques allow identifying future faulty nodes and schedule remediation jobs and other maintenance actions. MING [286] is an ensemble model that incorporates LSTM networks for processing temporal data, Random Forest [95] for spatial data, and a ranking model to prioritize affected nodes by their probability of future failure.

In the work of Li et al. [24], traditional and deep neural ML models are applied on alert, spatial, and build data to predict node failures in a large-scale cloud infrastructure. Alert data encodes early warning symptoms that might anticipate the occurrence of a failure. Spatial data stores the location information of nodes. Build data describes the internal configuration of a node, including e.g., hardware specifications, OS version, drivers, or kernel version. Via feature engineering, 1692 features deemed important are extracted from these multi-modal sources, and collected at constant time intervals to create multivariate time series. The time-series input is then fed into the different ML models (Random Forest [95], LSTM [6], and MING [286]) for predicting imminent node failures as a binary classification problem (healthy and faulty nodes represent the target classes). The approach also deals with the problem of data skewness, as the two classes are not balanced in the observations, by oversampling the minority class. The prediction approach is very general as it identifies any type of failure node and can be adapted to different sources of data. However, the final choices regarding model selection, feature engineering and sampling are highly tailored to the specific target system described. Therefore, re-adapting this approach in a new context requires re-evaluating the numerous techniques described to find the most suitable in the new context.

Jindal et al. [316] anticipate VM memory leaks through the use of linear regression models. A novel machine learning based algorithm, called Precog, is devised to detect time windows where memory utilization is expected to exceed a predefined utilization threshold. The expected leak time window is identified by fitting a regression line to the past memory usage values. The algorithm achieves a detection F1-score of 85.7% with high precision (100%), which makes it suitable for online remediation of memory leak failures.

The case of non-linear and piecewise linear resource consumption is examined by Alonso et al. [215], which adopts an ensemble of linear regression models that are chosen using a decision tree based on a combined set of hardware and software host metrics as input features. Because they provide a balance between accuracy and interpretability, decision trees are preferred. This choice allows to support root-cause diagnosis as well.

4.3 Online Failure Prediction Framework

This section presents a framework of Online Failure Prediction (OFP), which defines notions and methodology utilized in the proposed algorithms later in this chapter. The OFP is composed of a temporal model (Section 4.3.1), a failure model (Section 4.3.2), and a cost model (Section 4.3.3).

4.3.1 Temporal Model

OFP predicts future runtime errors by assessing the current and past state of a system [73,75,317]. The state of the system is represented by a series of t temporal measurements of m distinct variables periodically collected with period Δi_p by the monitoring system. The time range of collection of the input is the called observation window Δt_d. How far in time these errors can be foreseen depends on the *lead time* Δt_l of the predictor, i.e., the time between the prediction and the instant when the failure occurs; the validity of the prediction information also depends on the prediction time Δt_p, which is the length of the time window where the failure may occur. To these prediction quality measures, the warning time Δt_w can be introduced, i.e., a metric describing the time required to perform inference and signal a future failure. Naturally, a prediction is useful only if it can warn operators with sufficient time notice before the failure occurs (i.e., $\Delta t_w < \Delta t_l$). The lead time Δt_l may be provided as a system requirement based on SLA or hardware specification, or be estimated using fault injection techniques [313]. Figure 4.2 summarizes the temporal aspects related to failure prediction.

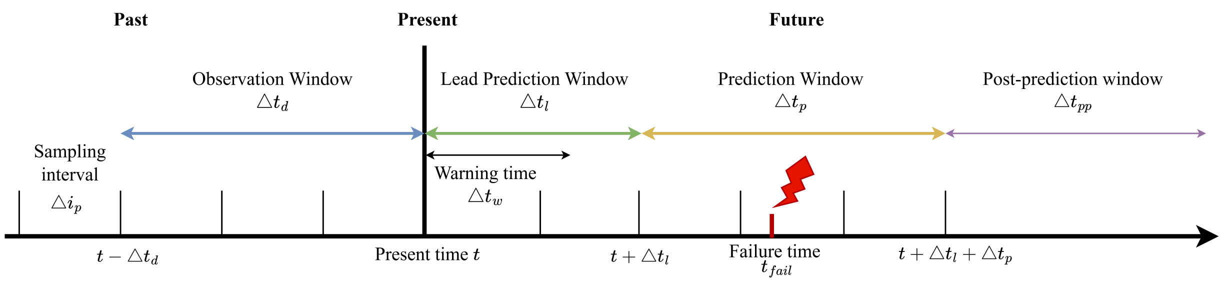

Past
Present
Future
Observation Window $\triangle t_d$
Lead Prediction Window $\triangle t_l$
Prediction Window $\triangle t_p$
Post-prediction window $\triangle t_{pp}$
Sampling interval $\triangle i_p$
Warning time $\triangle t_w$
$t - \triangle t_d$
Present time t
$t + \triangle t_l$
Failure time t_{fail}
$t + \triangle t_l + \triangle t_p$

Figure 4.2 OFP temporal model [317]. Observations are collected at every Δi_p interval. At prediction time, the samples collected for the last Δt_d are used to predict failures in the prediction window Δt_p, provided a sufficient leading time Δt_w needed to take preventive actions against predicted failures.

4.3.2 Failure Model

A failure model defines how components are believed to transition from a *healthy* state to a *faulty* state. As components tend to suffer from increasing aging and usage deterioration, and because many types of components suffer from soft failures, component failure states are practically distributed on a spectrum rather than being a binary (healthy/faulty) state, with healthier components being less prone to failure and performance deterioration, and old, aged components being more vulnerable and likely to experience failures. However, the majority of OFP methods tend to simplify this assumption and consider only healthy and faulty as valid categories of components, and treat the OFP problem as a binary classification task.

Different possibilities exist in terms of how to assign a component to the healthy or faulty class. For the purpose of constructing online failure prediction methods, a hardware component is considered faulty if it has been recognized as such by an O&M operator.

This usually means that a corresponding maintenance ticket has been opened, or the device has fallen out of pre-specified specifications (e.g., error rate or response time higher than a specific threshold). The observation windows related to a faulty module prior to the failure are labeled as the positive class, while observation windows of modules that did not fail during operation are labeled as the negative class (healthy).

After defining all the prediction parameters, and therefore assigning each observation window to one of the two classes, prediction models are evaluated by comparing the assigned class to the prediction class on test samples. To evaluate online failure predictors, the accuracy, precision, recall, and all the other metrics as defined in Section 2.3.5 can be used.

4.3.3 Cost Model

OFP must be justified by the costs involved in deploying a prediction system and the possible consequences of wrong predictions, against the standard cost of operating repairs traditionally. Such counterfactual analysis can be performed through the use of a prediction-repair cost model as illustrated below.

Let us consider an OFP scenario where failures are assumed to occur with probability p_{fail} over a predefined time period. An online algorithm produces component failure predictions at constant time steps. The quality of the predictions is measured by the True Positive Rate (TPR) and FPR metrics. The generation of predictions requires a monitoring infrastructure able to capture real-time events and symptoms of failing components, computing resources for executing the algorithm, as well as a dedicated team available to review failure predictions and replace components (Section 2.2.4). In a typical cloud administration scenario, computing resources are cheaply available and on-call personnel must be available to respond to incidents. The same applies to the monitoring infrastructure. Therefore, the overhead of these resources can be considered negligible for the moment. However, an algorithm with high FPR may still prove economically disadvantageous, because it may advise the replacement of healthy components. These unnecessary replacements may be more costly than the operating costs of handling unmitigated failures. Similarly, in the case of expensive proactive remediations, an algorithm with low TPR may not detect a sufficient number of replacements to render the online prediction advantageous. This decision problem can be modeled mathematically using the laws of probability. Assuming a replacement cost c_r and a failure cost c_f, the expected cost can be estimated based on the different prediction-failure combinations. The four possibilities may be modeled by a contingency table, as described in Table 2.1 of Chapter 2. Depending on whether a prediction algorithm is applied, the following costs apply:

Scenario	**Prediction**	**Failure**	**Cost with algorithm**	**Cost without algorithm**
True Positive	$+$	F	C_r	C_f
False Positive	$+$	$\neg F$	C_r	0
False Negative	$-$	F	C_f	C_f
True Negative	$-$	$\neg F$	0	0

Table 4.1 Table of action costs depending on scenarios of the contingency table.

The replacement of components is only convenient if it provides a decrease in expected cost. By computing the expected cost, which depends on failure probability p_{fail} and classifier performance in terms of FPR and TPR, the following relationship can be obtained (the full proof is provided in Appendix B):

$$c_r \cdot \left(1 + \frac{(1 - p_{fail}) \cdot FPR}{p_{fail} \cdot TPR}\right) < c_f \tag{4.1}$$

Note that because the multiplication factor of c_r is positive and greater than 1 (all factors in the left-side fraction are positive), the relationship implies that $c_f > c_r$ for the classifier to make economic sense, even in the case of a perfect predictor ($FPR = 0, TPR = 1$), i.e., the proactive remediation action must be cheaper than the total cost of failure. As costs of service downtime tend to range in the tens or hundreds of thousands of US dollars per hour [318, 319], and proactive repair costs are in the units to hundreds of US dollars, this assumption is typically satisfied.

4.4 Hardware Reliability

This section discusses the reliability and failures of the hardware layer, as introduced in Section 4.1.

4.4.1 Impact of Hardware Failures in Large-scale Computing

Hardware failures are common in large-scale computing infrastructures, and they are one of the most impacting factors contributing to failures and consequent service degradation, resulting in increased operational costs for both operators and end users [11, 320, 321]. 82% of server failures are due to hardware faults [11], therefore causing the large majority of observable failures in the cloud infrastructure layer.

It is estimated that 8% of servers are expected to encounter at least one hardware error per year of operation [11]; the machines affected by errors are more likely to require more than one repair per year (with an average of two repairs), showing successive correlation patterns between failures. Moreover, hardware faults tend to occur more frequently and require longer mitigation times [322]. Hardware faults tend to cause failures with lower lead times (up to 100x) compared to software faults [313], that consequently require a faster response. The problem is further aggravated by the fact that failure rates tend to be higher in real-world environments, compared to the ones reported in manufacturer data-sheets [292, 323].

Due to the high number of failures, hardware reparation costs for a 100k-scale datacenter can amount to millions of dollars. It is therefore crucial from an industrial perspective to investigate which factors influence the appearance of hardware faults with the end goal of improving design choices and deploying proactive approaches against failures. In addition, hardware faults affect on cascade both the virtualization and OS, as well as all overlying layers of the cloud model (including platform and software layers). Therefore, the reduction in occurrence and impact of hardware faults is fundamental to mitigate the occurrence of failures in other layers as well.

Hardware telemetry contains a high quantity of insightful data for diagnostics and prevention. Information such as SMART for HDDs, IMC logs for memory components, or DDM of optical transceivers provide important information for diagnosis and prediction of hardware failures.

4.4.2 Failing Hardware Components

A frequent measure to estimate component reliability is the Annualized Failure Rate (AFR) [324], which estimates the probability of witnessing a component failure in one year of operation. Table 4.2 summarizes the average AFRs for typical cloud infrastructure components. While the failure rates for single devices may be relatively low (varying from 0.13% to 21.9% annually, depending on the component [12, 317]), due to the law of large numbers and the commercial necessity to deploy commodity components in datacenters, the hardware fleet represents the most vulnerable aspect of a datacenter from a FM perspective.

Different hardware components exhibit different failure characteristics, in terms of frequency, failure patterns, and co-occurring dynamics [321]. Most hardware components also rely on different data sources. It is therefore difficult to model all hardware failures using a single well-known distribution. For this reason, hardware failures are usually handled with approaches that are tailored to the specific component domain.

HDDs are by far the most replaced components inside large cloud computing systems (70–82%) and one of the dominant reasons for server failure [11,321]. For this reason, most of the scientific interest is concentrated around hard drive failures [11,288,289,289,290,319,321,323,326,327], and the effectiveness of SMART [288,326] and other metrics [226, 289] for predicting future failures has been frequently investigated.

However, in recent years the number of failure studies conducted for other components has grown. Investigated components include DRAM memory chips, which exhibit a soft-failure dynamic that makes their failure extremely frequent (AFR 8.2%). DRAM failure prediction approaches based on correctable and uncorrectable errors have been proposed [299, 300, 300, 328, 329]. Other components such as optical transceivers [305–307, 330], SSDs [221], network switches [325], and RAID groups [223] have been analyzed (see Section 4.2.1 for a complete review).

4.4.3 Hardware Operations and Maintenance

Hardware Monitoring

Most internal server components compute metrics at the firmware or OS level. This information may be stored in logs (as in the case of DRAMs [296]) or in allocated storage sectors within the same host (as for hard drives [331] and optical transceivers [332]). Therefore, to store monitoring information in a centralized diagnostic infrastructure, the local host information must be retrieved periodically via scheduled *collection jobs* [117].

Component	Estimated AFR (%)	Ratio of Total Repairs (%)	Failure Types
HDD	1.36–8.6 [11, 288, 289]	70–82 [11, 321]	soft, hard
SSD	0.2–1 [221]	0.31 [321]	soft, hard
RAID	0.15–0.7 [223]	1.23–6 [11, 321]	soft, hard
DIMM	0.22-*8.2* [292]	3–5 [11, 321]	soft, hard
CPU	-	0.04–2.2 [321, 323]	hard
Motherboard	-	0.57 [321]	hard
Optical Transceiver	0.13–*1.65* [317]	-	soft, hard
Switch (Top-of-Rack (ToR), aggregation)	0.5–11.1 [322, 325]	-	soft, hard
Load Balancer	21.9 [322]	-	soft, hard
Others	-	13.94–18 [11, 321]	soft, hard

Table 4.2 Summary table of hardware failure characteristics by component. Italicized values indicate soft failures.

The observation window, i.e., the time window for which the telemetry data is collected to draw insights (see Section 4.3.1) varies significantly depending on the components under examination. Memory chips are observed for minutes to hours, hard drives are typically observed for days or weeks, optical transceivers may be observed for months [317].

For collecting monitoring data from network devices, an infrastructure based on Simple Networking Management Protocol (SNMP) may be used [317]. SNMP defines managed devices with an associated Management Information Base (MIB) collecting all previously registered data for the corresponding device, in a hierarchical form. An Object Identifier (OID) is a long numeric sequence that uniquely identify devices in the MIB hierarchy. OIDs are typically provided by the manufacturer or can be retrieved by querying dedicated OID repositories. OIDs are then used to retrieve MIB information for the devices of interest, e.g., by using the `snmpwalk` command. Retrieved data can be stored for offline data analysis or pipelined to live monitoring services. The network administrators can use the data stored offline to compare it with manufacturer-defined normal operating ranges and identify failures and performance degradation.

Hardware Failure Management

As hardware failures constitute an important share of the IT administration burden, several O&M solutions are in place to handle them.

The typical pipeline to deal with hardware failures comprises a detection stage, where failures are identified based on the hardware monitoring data; a reporting or alerting stage, where a corresponding alarm is generated; and a correction stage, where a remediation action is taken. A prediction stage may be added, where the online observations of hardware are used to predict failures in advance. An OFP system requires observation and response window to be carefully defined, so that correct preventive actions can be taken effectively (see Section 4.3).

Lin et al. [117] describe the framework used in Facebook production datacenters for hardware failure management (shown in Figure 4.3). A tool called MachineChecker runs a periodic set of checks on servers, such as ping reachability, power status check, SSH access, SMART, etc.. Then, whenever one of the executed checks fails, a second component FBAR, executes automated remediation routines. FBAR is also responsible for alerting and collecting diagnostic information. If FBAR actions do not solve the issue, the problem is escalated to an additional repair system. Eventually, unsolved issues are converted into repair tickets handled by human operators.

One of the challenges of hardware FM is how to handle transient failures. Transient failures may manifest during period of high workload or network latency, and may resolve independently without any remediation action being taken. If a component shows signs of failures, it may trigger remediation action which are not necessary, as the problem may independently resolve later on.

Whether the remediation action is convenient to take depends on the cost and consequences of executing such action. For example, migrating a VM to preserve the computing state before a server failure is typically a long and risky process, which may be justified only if there is strong evidence of an imminent fault. Component replacement may be convenient based on the cost of the component and the respective server workload. On the other hand, ECC techniques applied in DRAM components are cheap, non-invasive, and do not constitute a significant overhead, so they might be applied with fewer restrictions.

4.5 Hard Drive Failure Prediction

Hard Disk Drives (HDDs) are the components that are mostly frequently associated with hardware failures and repairs (see also Section 4.4.2). Servers with a higher number of disks are also more prone to experience additional faults in a fixed time period [11]. HDD failures frequently result in data loss, service unavailability, and increased operational costs [289,323]. For this reason, hard drives represent the most investigated target in both hardware reliability studies [11,288,289,321,323] and hardware failure prediction [289,290,319,326,327].

This section discusses HDD failure prediction, introducing notions such as employed data sources, common methods, and failure correlation patterns. A series of experiments conducted and lessons learned from HDD failure prediction are summarized at the end of the section.

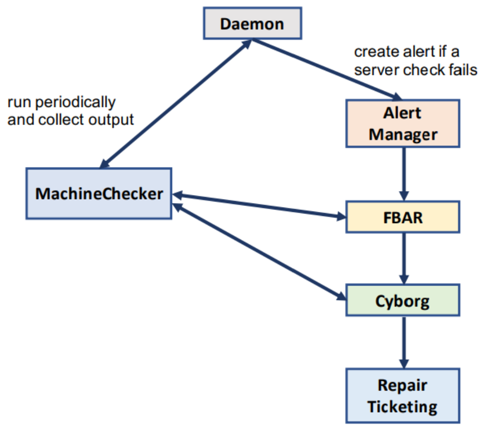

Daemon
create alert if a server check fails
run periodically and collect output
Alert Manager
MachineChecker
FBAR
Cyborg
Repair Ticketing

Figure 4.3 The hardware O&M pipeline used in Facebook datacenters [117].

4.5.1 Hard Drive Monitoring Data

The need to understand HDD failures and to construct uniform failure predictors and detectors has led hard-drive manufacturers to adopt standardized monitoring technologies, such as SMART [331], in their storage products [129].

SMART is a set of over 50 monitoring attributes [331,333] collected from storage drives, describing physical and logical variables related to the drive heath [289]. The SMART values are collected continuously and reported on daily basis. Each SMART attribute is associated with a corresponding integer ID (e.g., 197). The monitoring information reported originates from sensor instrumentation (e.g., temperature) and internal firmware (e.g., reallocated sector count). Although SMART is designed to harmonize the convention of disk monitoring data across vendors, SMART metrics are still subject to cross-vendor variability and lock-in, which makes comparison of prediction models difficult between disks of different manufacturers.

Some SMART attributes were originally left undocumented. SMART attribute interpretation also varies based on the manufacturer, and the utilized serial data interface, such as SAS, SATA, or NVMe. This complicates the comparison across different disk technologies, such as SSDs, and requires to reason in terms of effective information computed (e.g., unnormalized latent count sector in last 24 hours) rather than SMART metric ID (e.g. SMART 5).

Frequently used SMART features may be divided in few categories. Error metrics measure the amount of soft/transient failures experienced by the hard drive, such as read/seek error rate, pending/uncorrectable sector count, reallocated sector count. Usage metrics measure the device aging and wearing due to component utilization. They include power-on hours, power cycle count, spin-up time, start/stop count, load/unload cycle count. Other metrics include temperature, spin retry count, seek time performance. For each metric, both the raw and unnormalized value is typically reported.

Non-SMART metrics used for HDD monitoring include performance data [289], such as disk latency, waiting time, and I/O throughput. Server-level data may also be included (CPU usage, paging, swap, . . .).

Some works [289,290] also utilize spatial information, such as position in rack, server, room, and datacenter. This allows to include the state of neighboring devices, such as other servers or disks, in the failure model. Information from disk drivers may also be complemented [290].

4.5.2 Methods for Hard Drive Failure Prediction

The majority of HDD failure prediction approaches treat the problem as a binary classification task [290, 319, 331], as described in the OFP framework of Section 4.3.

Most common classification methods include tree-based models [11, 23, 219, 220, 319], SVMs [130, 224], LSTM- and sequence-based neural networks [219, 225, 289, 290] (such as RNNs and transformers). Papers frequently apply feature selection techniques, as not all SMART (and other) metrics have the same predictive power. Several papers have conducted a feature analysis to select most important SMART metrics [289, 290, 319]. The selection of important features is frequently dependent on manufacturers, as a consequence of availability of metrics and non-aligned conventions of the metric meaning. Nevertheless, the most frequently selected metrics include pending/reallocated sector count, load/seek cycle count, temperature, and read error rate. The total number of used SMART metrics typically varies between 12 and 14 [289, 319]. Predictions models are evaluated on publicly-available benchmark datasets (such as the Blackbaze dataset [334]).

Typical observation windows of monitoring data vary from 15 to 25 days [290, 319]. The explored failure prediction windows Δt_p range from 2 to 30 days in advance to failures, with 10–16 days showing the most promising results. This allows O&M teams to have sufficient time for replacing components or migrating workload, while balancing prediction coverage with accuracy. The lead time Δt_l is typically set to 0.

4.5.3 Conducted Experiments and Results

Previous works have focused on feature selection from SMART metrics, but have not studied the patterns leading to failures within time-series observations. Moreover, with few notable exceptions [319], the majority of works has been constructed from public HDD benchmarks, which are static collections of disk data and do not reflect the challenges of training, evaluating and deploying OFP algorithms in real-world cloud environments.

To this end, several experiments were conducted to study the dynamics of HDD failures inside a large-scale cloud provider and determine how to effectively evaluate online failure predictors.

SMART metrics were collected for a period of 6 months on tens of thousands of SAS disks. Differently from previous work, SMART time series have been analyzed individually based on their temporal evolution, rather than on their aggregate value or cross-correlation. The analysis of univariate SMART time series reveals the progressive state of degradation which HDDs experience, which leads them to encounter increasing quantity of soft errors before a critical failure occur. These soft failures are manifested as increases of some SMART metrics, such as the grown defect list and the total number of uncorrected errors in read and verify operations.

This enables the construction of single-metric trend predictors, which analyze the increasing trend of important metrics in the last 7 days. Trends are evaluated by means of slope features, such as the incremental ratios between different time-steps within the observation window. These features can be combined to train ML models, such as RFs. However, a large fraction of failures exhibits sudden failure dynamics, which makes them difficult to be anticipated with the current method. Therefore, final HDD failure predictors constructed in this way tend to exhibit high precision (> 90%) but lower recall (40%).

In terms of evaluation, it was observed how the construction of observation and prediction windows plays a significant role in the prediction accuracy of a predictor. If predictors are used for prediction every day in a rolling window fashion, they tend to produce more false positives as they spot early signs of degradation. When the prediction window approaches the actual failure time, the FPR decreases. This motivates the necessity to develop models able to distinguish early symptoms from terminal failure patterns.

A final finding was related to the importance of sub-labels. Although previous works have defined failures on the basis of replacement tickets, in practice HDDs exhibit different types of failures which can be interpreted using failure ticket descriptions or logs. Failing components with same sub-labels tend to have similar failure patterns in SMART metrics, which may prove important to construct effective failure predictors. These sub-labels may also play an important role in determining the root cause of the component failures. For example, some disks experience warning of high I/O usage (close to 100%), which causes them to appear faulty. If these disks are however accessible and fully functioning, these may indicate a problem in the RAID controller.

4.6 Memory Failure Prediction

This section presents a summary of techniques for online prediction of primary memory failures. Memory systems are described in terms of architecture and failure modes and the memory failure prediction problem is introduced. In conclusion, a framework for hierarchical memory failure prediction is summarized [296].

4.6.1 Memory Systems and Failures

Primary memory is typically implemented as a Dynamic Random Access Memory (DRAM) in Dual In-line Memory Modules (DIMMs). A typical memory system has a hierarchical architecture (depicted in Figure 4.4), which comprises a number of Integrated Memory Controllers (IMCs) which enable CPUs to communicate to DIMM chips via *memory channels* [296]. Channels may be shared between different DIMMs.

Each DIMM is composed of several ranks, and each rank is a collection of several bi-dimensional arrays of cells. These 2D arrays may be indexed by row and column, and each cell contains a fixed number of bits determined by the memory data width.

Primary memory failures account for a significant fraction of hardware-related failures in a datacenter, accounting for approximately 5% of all fault events (Section 4.4.2), with annualized failure rates of 0.22% [292].

The primal cause of memory failures are bit errors, which cause one or more memory cells to become unusable because their values are corrupted [296]. Causes of bit errors include strikes of high-energy particles and cosmic rays, leakage current, etc. A memory bit error occurs when the data delivered by the DIMM does not match the value expected by the IMC.

Error Correction Code (ECC) schemes provide a mechanism to partially detect and recover from bit errors. ECC incorporates the use of additional metadata bits to verify the content of memory cells, e.g., by performing checksums or other verification tests. During runtime, the IMC computes the ECC verification test and if the computed outcome differs from the value stored in the ECC bit, an error is raised.

Depending on the ECC scheme employed, different memory components may have different detection and correction capabilities. If a memory error can be detected and corrected, it is called a CE and does not cause any immediate component failure. If a memory error can be detected but not corrected, it is called an UCE and it may cause a component failure when the corresponding error cell is accessed by the kernel. In that case, it is called a critical UCE. Critical UCEs typically lead to service interruption and may trigger high-level recovery actions, such as VM migration and component replacement. If the system has not consumed the UCE error and the system is not yet affected, the UCE is defined as a As-Yet-Unconsumed (AYU) UCE. As AYU UCEs do not cause any visible impact on the system, they are usually not considered failures.

UCEs are not the only causes of failures, as other types of malfunctioning, not related to bit verification, may occur to the component, e.g., manufacturing defects. Therefore, memory components experience both soft and hard errors. Soft errors are represented by CE, which can be handled through ECC. Hard memory errors are present in the form of component faults and UCEs.

4.6.2 Memory Failure Recovery Techniques

To prevent memory errors, several memory recovery mechanisms may be employed in addition to ECC [296].

Sparing consists in the exclusion of a memory region (cell, row/column, or bank) that was previously affected by a fault. Sparing can be both implemented in the software and hardware. It however requires high redundancy and computation overload, which may affect system performance.

Component replacement allows to recover an affected host. It affects the runtime of the host and adds additional economic cost.

VM migration allows to preserve the compute workload by moving OS and software operations from an affected component to a new host, and prevent service disruption. Migration preserves the workload but not the hardware functionality, and it requires time and redundancy allocation to be performed.

4.6.3 Memory Failure Prediction

Memory failure prediction is the OFP of memory failures, to enable failure recovery strategies above-mentioned.

A variety of metrics and static fields may be applied to this end. Monitoring information useful to predict memory failures include (from more granular to higher-level) [296]:

- past CEs [298], measured in absolute count in a time interval or as frequency (errors per second);
- micro-level accesses and fault counts [301, 302], by hierarchy of memory chip (cell, row/column, bank);

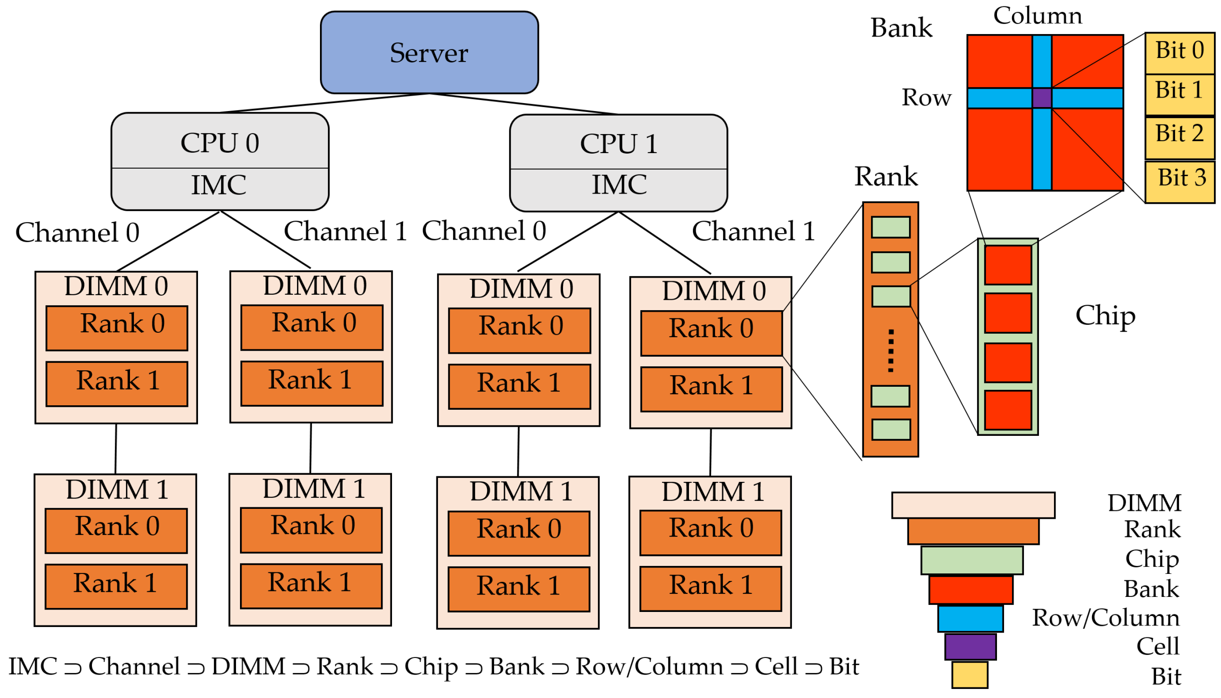


Figure 4.4 Hierarchical overview of a memory system, from server to bits [296]. Each CPU embeds an IMC, which communicates to the DIMMs over serial memory channels. DIMMs are hierarchically composed of 1D subunits down to the bank level, which is a 2D map of memory cells, each containing a fixed quantity of bits (*word size*, e.g., 16 bits).

- spatial and temporal distribution features of error bits [231, 300], e.g., number of adjacent error bits, or minimum time interval between two consecutive error bits, aggregate statistics about error bits over time (minimum, maximum, standard deviation, ...);
- number of AYU UCEs [296];
- static memory features [301], such as manufacturer, data width, memory frequency and capacity;
- memory-related events from kernel logs [231, 297, 303], typically triggered by system-level handling mechanisms in the presence of high quantity of CEs (so-called CE storms);
- workload metrics [302], such as CPU/memory utilization, read/write throughput, memory usage, etc.

While low-level features such as CE counts and distribution of error bits are useful for *micro-level memory failure prediction*, i.e., prediction of memory failures at the row/column, bank, or chip level, micro-level failure prediction is not highly actionable as it does not immediately constitute a reliability risk, thanks to the hardware and OS-level tolerance mechanisms in place.

On the other hand, workload metrics provide high-level information about the state of a server or composite system, but in the past they have not provided statistically relevant results for predicting system-level failures.

4.6.4 Hierarchical Memory Failure Prediction

Different recovery actions operate on different hierarchical planes (bank, chip, server) and thus require failure prediction at the corresponding level. Previous works have focused on either micro-level prediction [299, 301, 302], DIMM prediction [297, 299, 300], or server-level prediction [231, 298, 303]. Integrating all predictions in a unique framework can therefore provide higher value for O&M operators.

The hierarchical memory failure prediction (HiMFP) [296] integrates micro-level (cell, row/column, back, chip, rank), DIMM-level and server-level prediction into a unified context.

To predict micro-level failures, threshold-based detection is applied on handcrafted data wire (DQ) features [296]. These features are constructed from observation of errors in multiple data wires transmissions and may be indication of DQ wire faults or multi-bit error during transmission. Each feature monitors a specific DQ error pattern and whenever a specific threshold is reached, a micro-level failure is reported.

ML models are used to predict DIMM-level failures, using the features described in Section 4.6.3.

To predict server-level failures, DIMM-level failure prediction probabilities are aggregated using probability rules, e.g., maximum DIMM failure probability, or at-least-one DIMM failure probability.

4.7 Optical Transceiver Failure Prediction

This section discusses failure prediction applied to optical transceiver modules, one of the key components in modern datacenter infrastructures. The findings here discussed include the results of a large-scale reliability study conducted on optical transceivers [317], which studies the failure dynamics of these components (Section 4.7.4) and introduced some failure prediction algorithms, as well as other neural-based approaches later discussed (Section 4.7.5).

4.7.1 Introduction

Optical transceivers are essential components of the datacenter infrastructure, as they are responsible for connecting the optical and electronic network infrastructure by converting electrical signals into light, and vice-versa [317]. Because of their compact size and relatively low failure rates, they have been frequently overlooked by previous reliability studies. However, a large quantity of transceivers is necessary to run a datacenter infrastructure and their vital function renders the study of their failures fundamental for network administrators.

DRAM and hard drive failures, discussed in the previous sections, tend to have a localized impact, which only affect the interested host. A faulty optical transceiver, on the other hand, can cause widespread network disruption, which has severe consequences for service quality and can affect a large number of hosts simultaneously. If a transceiver fails, it can compromise the reachability to the switch to which it is connected, and cause unreachability or performance degradation to other devices on a cascading effect. This further prioritizes the necessity to evaluate the reliability of these components, as well as the possible anticipation of transceiver failures to minimize the impact of their failures.

Several anomaly detection [304, 308] and online failure prediction algorithms for optical transceivers have been proposed [305–307, 335]. However, only a few studies have analyzed optical field data in detail [305, 307, 335] and very few small-scale studies have analyzed their failure modes [304, 305], while the majority of large-scale reliability studies have concentrated on other components [292,321,323]. Moreover, the insufficient quantity of open transceiver data has limited the development and comparison of statistical models for the prediction and detection of failures. In addition, all previous works related to optical failure analysis have concentrated on the analysis of the physical variables of transceivers (such as temperature, current, etc.). No previous work has investigated the association of OS-level metrics, such as throughput or error rate, with the appearance of failures, as previously done for other components [289, 302].

In a reliability study previously conducted [317], a large quantity of monitoring information from optical infrastructures in datacenters was leveraged to conduct an in-depth comprehensive study on optical transceiver failures. This section reports the findings related to this previous study. Such findings have been complemented with additional information on how to effectively operate and evaluate optical transceiver failure prediction in real-world production environments.

Contributions

The contributions [317] to the optical transceiver failure prediction problem can be summarized as follows:

- the analysis of transceiver failures (Section 4.7.4), estimating the AFR for optical transceivers in the wild, by monitoring the operational status of over 130 optical datacenter networks for a period of 15 months;

- the discovery and description of optical failure characteristics, in the form correlations, patterns and operating intervals for failures (Section 4.7.4), by associating commonly monitored attributes with the appearance of failures;

- the development of online failure prediction algorithms (Section 4.7.5), based on the application of the statistical insights derived from the previous analytical study.

The next sections present a summary of this study, describing its methodology and results, the main conclusions, and the proposed approach for optical transceiver failure prediction, based on the statistical insights drawn from the study observations.

4.7.2 Optical Transceivers, Networks, and Failures

Optical Transceivers

Optical transceivers (or *optical modules*) are low-voltage devices that convert electrical data signals into optical signals and vice-versa [317]. These signals are transmitted over the optical fiber by means of wavelength-specific lasers.

An optical transceiver can operate one (single-mode fiber) or more data streams (multi-mode fibers). Each data stream is converted to a signal with a unique wavelength, meaning that it is effectively a unique light color (e.g., 850 nm, 1310 nm or 1550 nm). Therefore, the functionality and performance of optical transceivers is deeply reliant on the correct working of the laser.

Optical transceivers adopt various form factors as defined by multi-source agreements. Common standards include Gigabit Interface Converter (GBIC) [336], SFP [337] and its extensions, such as SFP+, Quad SFP (QSFP), Octal SFP (OSFP) [338]. SFP-related standards correspond to the dominant form factors in the market [339, 340]. They are compact network modules that can be inserted in the SFP slot interfaces present in modern optical switches and routers. Compared to fixed interfaces, SFP interfaces provide more flexibility in terms of choice of transceiver and input transmission medium (copper or fiber). All SFP modules are backward-compatible with predecessor form factors, either out-of-the-box or by means of low-cost adapters.

Optical Networks

As described in Section 2.1.2, modern datacenters implement broadband connectivity using optical fiber links to increase bandwidth and throughput. These optical links connect a hierarchy of ToR, aggregation and spine switches to form a tree-like topology (see also Figure 2.1 in Section 2.1.2). Although the physical links are realized by means of optical fiber, these point-to-point interconnects still require electronic switches to implement connectivity across racks, clusters, and the outside world. Light signals traveling in fiber links must be converted to electronic signals for the switch to process it, and back to optical signals for the next hop. These conversions are performed by optical transceivers, low-power devices acting as gateways between electronic communication and optical fiber.

An emerging alternative in datacenter networking is all-optical interconnects [317, 338, 341]. Differently from point-to-point connections requiring transceivers at the fiber ends, all-optical interconnects allow to implement network switching directly at the physical layer, reducing latency and power consumption while increasing bandwidth [338]. However, the adoption of optical interconnects is slowed down by high investment costs and limited buffering capabilities, resulting in hybrid architectures using both optically-switched and electronically-switched interconnects [330, 341]. Moreover, even in the presence of fully-realized, all-optical core infrastructure, devices at the edge of the optical network, such as servers or ToR switches, operate on electronic signals and thus require optical transceivers for each incoming and outgoing link.

Therefore, the necessity of optical transceivers to enable electronic packet switching in datacenters remain high, and the increase in demand of cloud computing services drives to expand their application volume [340].

Optical Transceiver Failures

As all hardware components, optical transceivers are subject to the appearance of failures, which cause the device to malfunction or perform undesired behavior. In this analysis, two types of optical transceiver failures are defined:

- Soft failures (or *transient failures*) occur when the optical module temporarily experiences a temporary degradation of performance, before going back to its fully operational state. This can be due to temporary factors [335], such as increased workload over a short period of time, but it may also be symptom of component aging and wear;
- Hard failures (or *faults*) occur when the optical module stops working permanently and must be replaced. These failures are mostly caused by internal component malfunction, especially the laser [309, 335], but can also be caused by dust or fingerprint contamination [304, 342], as well as Electrostatic Discharge (ESD) due to poor grounding or dry environment [342]. The normal process of aging deterioration of the module components also contributes to the insurgence of these faults.

Cyclic Redundancy Check (CRC) identifies accidental errors occurring during transmission [343]. Transmitted data incorporates check values that are used to verify the integrity of the information. When the interface is disrupted or interrupted, the number of packets with CRC errors increases. Therefore, the degradation of performance is measured using error rate, expressed in CRC packets per second (pps). In normal conditions, an optical transceiver module displays an error rate of zero. While small, sudden spikes in error rate may indicate a soft failure, high persistent error rates are characteristic of hard failures. Hard failures are also empirically associated with high error rates, as the network interface displays high level of error rates before the final breakdown.

4.7.3 Optical Infrastructure and Data Sources

This section describes how large-scale infrastructures composed of optical transceivers can be monitored and how the data sources employed in the study [317], namely the DDM information stored in transceiver modules and additional network metrics from the operating system level (OS-level metrics), may be collected.

Digital Diagnostic Monitoring (DDM)

Most modern optical transceivers implement some standard monitoring capabilities internally. The multi-source agreement SFF-8472 [332] defines an enhanced DDM interface for SFPs, also known as Digital Optical Monitoring (DOM) interface. This interface allows to access real-time device operating information, such as temperature, optical input/output power, and voltage. Such real-time information is accessible via a serial interface at specified addresses in the Electrically Erasable Programmable Read-only Memory (EEPROM) memory of the device.

The DDM specifications also include a system of alarm and warning flags to alert the host system when particular operating parameters are outside normal operating ranges. For example, if the temperature of an optical transceiver is outside factory-defined operating range, the flag bit associated with temperature will be set in the memory map of the device, then the platform supervision system can check if the interface is working correctly by polling the status registers [332].

After the DDM memory-mapped information is stored in the internal EEPROM of transceivers in compliance with SFF-8472, it can be accessed over the network via the application-layer SNMP protocol (see also Section 4.4.3). The DDM interface of SFP modules stores samples of five dynamic fields defined in the standard [332]:

- *voltage*, the internally measured supply voltage of the module (expressed in volts);
- *bias current*, the electric current (expressed in mA) that is fed into the laser diode to keep it in the operational state. Older and less efficient lasers require more bias current to operate, therefore this physical quantity carries information about the deterioration state of the module transmission laser;

- *temperature*, as internally measured and expressed in degrees Celsius (°C);
- receiving/transmitting *(Rx/Tx) optical power*, expressed in dBm;

The DDM also stores static transceiver data, i.e., metadata information of the module that does not change over time. These fields, although they do not provide real-time information about the physical state of the system, allow to understand better the operating characteristic of the transceiver, e.g., what is the expected transmission distance at design, the number of operational channels, the transmission medium. This information can, therefore, provide potential insights into the failure characteristic of some transceiver groups. Static fields stored in the DDM include:

- *wavelength*, nominal transmitter output wavelength (expressed in nm);
- *distance*, the transmission distance (in km). If more than one transmission mode is allowed, all corresponding distances are collected.
- *interface name* (e.g."GigabitEthernet4/0/1"). The first half of the name allows to extract the interface type, while the second half allows to estimate the total number of interfaces of the host.
- *module type*, the form factor of the transceiver, e.g., "SFP" or "SFP+";

OS-level Metric Data

Optical monitoring information may also be combined with other sources of diagnostic information describing the state of the network, e.g., latency, network throughput or transmission error rate. These metrics are collected at the OS level inside the corresponding network device. The association of optical monitoring data to device data can be achieved by associating network interfaces and device IP addresses.

The following OS-level metrics are considered:

- *Error Rate (ER)*, count of CRC errors [343] measured per unit of time (expressed in packets per second, pps);
- *packet loss*, count of lost transmitted packets per unit of time (expressed in pps). Lost packets include all packets that were deliberately not re-transmitted by the interface, for reasons different from CRC errors, e.g., freeing up buffer space;
- interface *Rx/Tx throughput rates*, expressed in bits per second (bps) and log10-scaled;

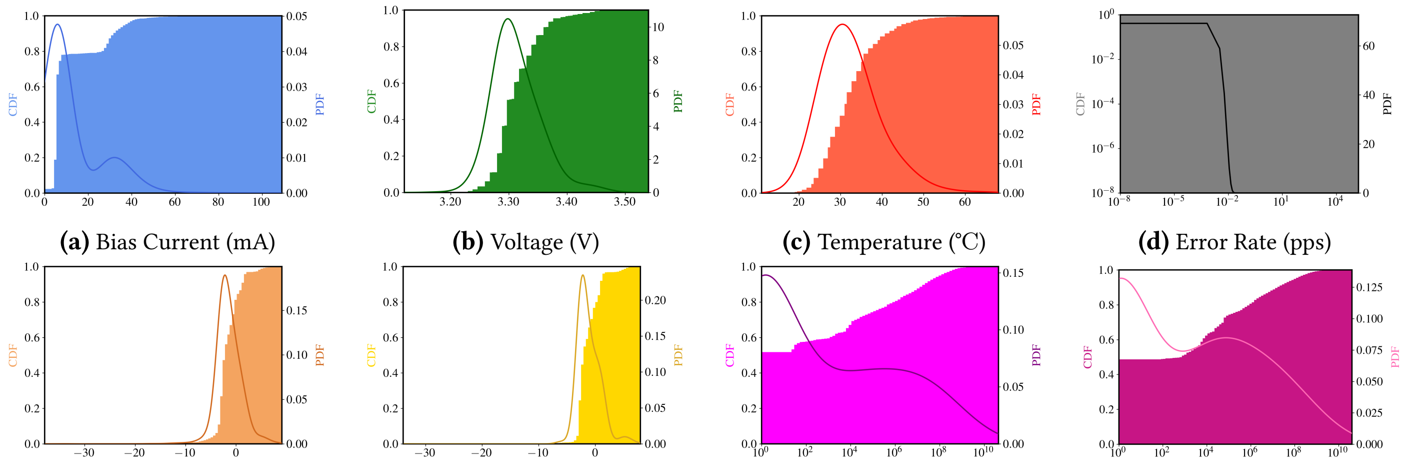
4.7.4 Transceiver Reliability Study

Here are presented the results of the reliability study of optical transceivers [317]. The experiments were conducted using the monitoring data above described, gathered from the optical diagnostic infrastructure and OSs-level tooling of servers. The conclusions drawn are summarized at the end of the section. The experiments conducted and here summarized include a failure rate analysis, a time-series correlation analysis, a study of operating ranges of healthy/faulty components, an analysis of frequent failure patterns and attribute importance for failure prediction.

Dataset and Statistics

For the failure rate analysis, 3.5 million modules deployed in 131 regions were monitored over a period of 15 months from July 2021 to October 2022. The total number of device tracked hours amounts to 4.05 million device years. Modules were tracked with their corresponding error rate to check if they had failed. This *large-scale dataset* is used exclusively to collect insights about baseline failure rate of optical transceivers (Section 4.7.4).

For all other analyses performed, a second dataset (*metrics dataset*) is collected, consisting of the complete set of data sources (DDM and OS metrics) described in Section 4.7.3. This second dataset is the formed by joining three separate data collections:



(a) Bias Current (mA) **(b)** Voltage (V) **(c)** Temperature (°C) **(d)** Error Rate (pps)

(e) Rx Power (dBm) **(f)** Tx Power (dBm) **(g)** Rx Throughput (bps) **(h)** Tx Throughput (bps)

Figure 4.5 Empirical Probability (PDF, as line) and Cumulative (CDF, as histogram) Density Functions for DDM and OS metrics estimated from the collected dataset [317]. Voltage, temperature, and Tx/Rx power follow Gaussian-like distributions, while bias current exhibits a superposition of two Gaussian modes, likely as a result of the deployment of transceivers in separate time windows. Error Rate displays an extremely fast-decaying distribution, while Rx/Tx throughput distributions decay slowly after peaking at intermediate values.

- *Collection 1* of datacenter A consists of 18151 optical modules attached to 223 network switches. The data is collected for the observation period of 1st-31st January 2022.
- *Collection 2* of datacenter B consists of 3042 optical modules attached to 38 network switches. The data is collected for the observation period of 1st-31st January 2022.
- *Collection 3* consists of historical records of 128 faulty modules for a time span between 30 and 45 days before failure, collected from several regions over the course of 2021. To compensate for the class imbalance, it also contains additional healthy modules from the same observation period. 642 modules are observed in total for this collection.

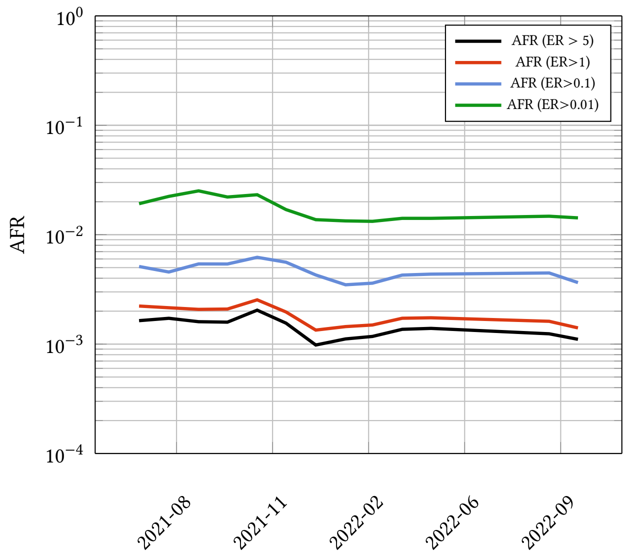
All three dataset collections record optical DDM information (both static and dynamic attributes) and OS metrics at 5-minute resolution, for a minimum consecutive period of 31 days. Figure 4.5 shows the value distributions of the dynamic attributes collected from DDM and the OS. The Probability Density Function (PDF) for different attributes is estimated using Kernel Density Estimation (KDE) (See also Section 2.3.3).

Failure Rate Analysis

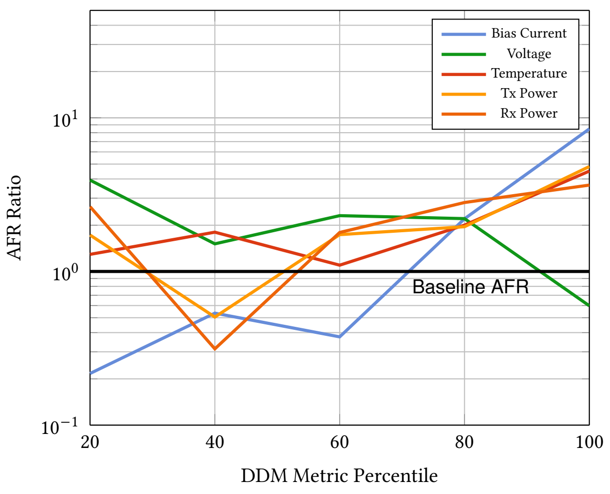
Based on the above-mentioned large-scale module dataset, the average failure rate of optical transceiver modules can be estimated. As for other components described in Section 4.4.2, the AFR is used as measure of failure probability. Given N network interfaces, where transceiver information is monitored for observation windows $[W_1, W_2, \dots, W_i, \dots, W_n]$, AFR is defined as:

$$\mathrm{AFR} = \frac{\sum_i^N \#\text{ component failures in } W_i}{\sum_i^N |W_i|}, \tag{4.2}$$

where $|W_i|$ is the length of observation window in years. Each interface can experience more than one failure if a transceiver fails during the observation window and it is replaced, or if it experiences soft failures that allow to continue operating. The total observation time at the denominator amounts to 1.5 billion component-days. For the total number of failures in observation windows W_i, different degrees of failure severity are tracked, based on the expertise of network administrators. For hard failures, a failure is established if the average ER value exceeds 5 pps in the predefined sampling window of 5 minutes. Similarly, soft failures are monitored based on the following error rate thresholds (in pps): 1, 0.1, and 0.01.



(a) Measured annualized failure rate (AFR) in the observation period (14 months) for different failure severity levels.



(b) Relative increase in annualized failure rate (AFR) as a function of mean DDM metric values, expressed in quintiles (20, 40, 60, 80, 100%).

Figure 4.6 Results of Failure Rate Analysis of optical transceivers [317]. On the left, historical AFR values for the observation period by error severity, on the right the relative increase in AFR in relation to DDM attributes.

Figure 4.6a shows the historical variation of AFR over the monitoring period, by different failure severity. Table 4.3 provides overall statistics of observed failures, and estimated Mean Time Between Failures (MTBF) and average lifespan of modules.

For hard failures (ER > 5), an AFR of 0.1341% is estimated based on 4809 observed failures in optical modules. That corresponds to an average MTBF of 6.537 million hours, or 746 years of average lifespan. This value is comparable in scale to the most accurate failure rate estimates for DIMM chips (0.22%) [292] and significantly lower than AFR of hard drives (1.7–8.6%) [288].

For soft failures, the values increase proportionally to the detection threshold, reaching an absolute maximum of 1.64% AFR for ER > 0.01, i.e., at least 1.64% of optical transceivers experience at least once a small amount of CRC errors annually. The frequency of soft failures with respect to hard failure rate (12.22x) is significant when compared to the soft failure rate of other components. For example, hard drives experience latent sector errors, a type of soft failure, with a rate of 3.45% [327], only 1–2x times the rate of complete HDD faults. DIMMs experience soft errors with an AFR of 8%, or 36x times more their "hard" failure rate [294]. This indicates that optical transceivers are more similar to DIMMs in terms of failure characteristics, as they experience frequent soft failures. Moreover, in both DIMMs and optical transceivers, past and future failures are strongly correlated. Failure rates for intermediate thresholds are progressively increasing (0.1726% for ER > 1, 0.4535% for ER > 0.1) with a significant overlap (77.69%) between ER > 1 failures and ER > 5 failures, indicating saturation over high values of ER.

ER threshold (pps)	# failures	AFR	MTBF, hours	average life, years
5	4809	0.1341%	$6.537 \cdot 10^6$	746
1	6190	0.1726%	$5.079 \cdot 10^6$	579
0.1	16263	0.4535%	$1.933 \cdot 10^6$	221
0.01	59041	1.6463%	$5.325 \cdot 10^5$	61

Table 4.3 Number of observed failures and AFR estimates reported in the reliability study [317] for different ER thresholds, with corresponding MTBF and average component life.

In addition, the relative increase of AFR was measured as a function of prior DDM attribute values. For this analysis, reported in Figure 4.6b, the mean value of dynamic DDM attributes in the full observation window is considered. Modules with anomalous high mean values (> 80th percentile) of bias current, temperature, and Rx/Tx power experience increasing risk of failing up to 8.46x times, measured by the increase in AFR (y-axis). Similarly, low values (< 20th percentile) of all metrics, except bias current, correlate with future failure to a lower extent, with voltage being the attribute with the highest failure increase for low mean values (3x).

Time-series correlation

The correlation of DDM time-series signals over time was analyzed, with the objective of analyzing the possible temporal dependencies between the different monitored metrics. The concept of cross-correlation of time series, here borrowed from signal theory, was developed for the purpose of analyzing correlations of value observation as a function of lag (or distance in time) between samples. Cross-correlation of two discrete, real-valued time series is defined as

$$(f \star g)[n] := \sum_{m=-\infty}^{+\infty} f[m-n]g[m] \tag{4.3}$$

where n is the lag between f and g. The larger the value of $(f \star g)[n]$, the stronger the correlation of the two signals relatively shifted by n time steps. When $f = g$, the cross-correlation is defined as *autocorrelation*.

To measure cross-correlation in the optical transceiver scenario, the module time series were normalized using min-max normalization, based on the aggregate statistics collected at the dataset level. Then, cross-correlation was measured for pairs of different DDM dynamic attributes for each module separately, and summed partial results across different modules to match lags, then the obtained result was linearly scaled so that the central value ($n = 0$) has value 1.

Figure 4.7 depicts the autocorrelation of error rate, compared to the autocorrelation of some baseline signals, namely random white noise (black) and unit ramp signals, growing linearly from 0 to 1 at different rates. For lag $n < -1$ days, the autocorrelation is positive and increasing at a rate comparable to the autocorrelation of a 50-day unit ramp. This entails that from the first appearance of CRC errors, new CRC events tend to repeat with higher magnitude. For $n \to 0^-$, in particular for lags below 24 hours, the autocorrelation grows significantly faster, indicating higher correlation of CRC events to re-occur within a short time frame. This indicates that the probability of observing CRC events is higher within 24 hours from the first error rate manifestation. Because autocorrelation is proportional to the magnitude of signals, this may not only indicate an increase in overall occurrence, but also an increase in quantity of CRC errors when approaching high-error-rate contexts. Overall, the analysis suggests that:

- modules that experience positive error rate tend to experience it again, and more often at short time distances from the first CRC error event;
- modules experience increasing quantity of error rate before failing.

Operating Ranges of Healthy and Faulty Modules

An analysis of the operating values for the dynamic attributes was performed, to understand if specific value intervals are associated with failures [317]. For each of the 8 dynamic attributes monitored, the value distribution is monitored across healthy and soon-to-fail modules.

Table 4.4 reports the 5th-to-95th percentile ranges for aggregate statistics of optical data, measured separately for healthy and faulty modules. Table 4.4(a) reports the distribution of mean values over the module observation window, while Table 4.4(b) reports the observed standard deviation, used as a measure of value variability over time.

For dynamic monitoring attributes, bias current operating range is considerably larger in faulty modules (6 – 84.20 mA) compared to healthy modules (5 – 38.71 mA). Bias current time variation is on average +52%

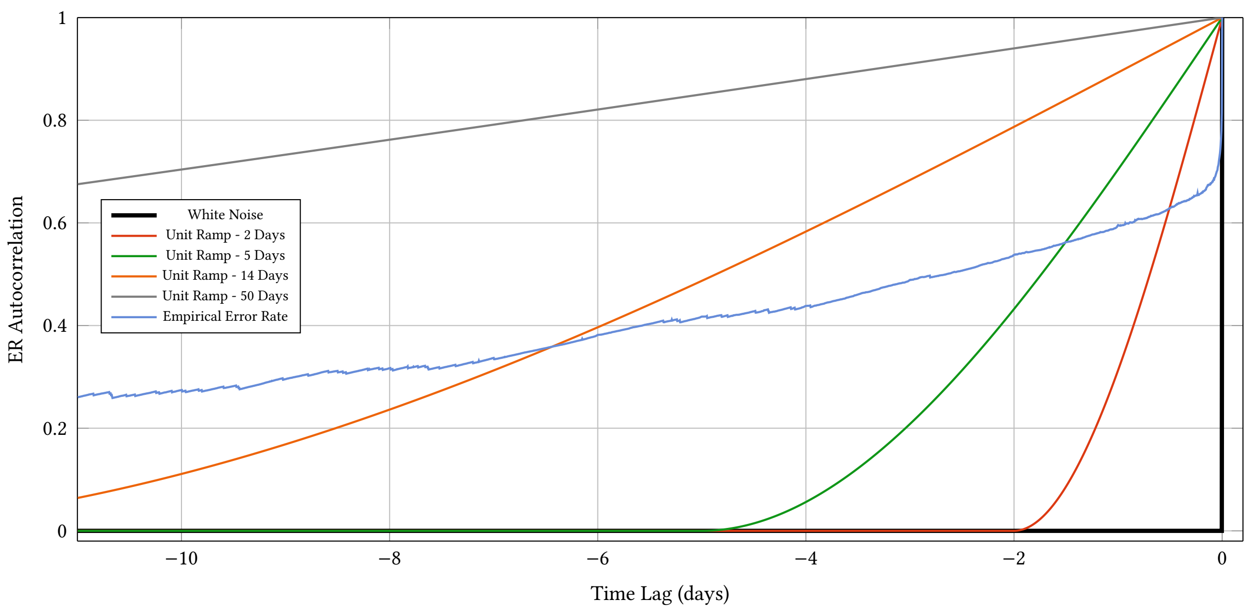


Figure 4.7 Normalized mean autocorrelation of ER signals in optical transceivers [317], compared to the autocorrelation of a white noise signal and unit ramp signals reaching peak value at different rates.

higher in faulty modules (0.5342 mA) compared to healthy ones (0.3508 mA). This indicates that high bias current values may be indicative of future failure, because they are only observed in soon-to-fail modules.

Voltage values are very stable with low standard deviations in both scenarios (high $4.978 \cdot 10^{-3}$ V and $5.495 \cdot 10^{-3}$ V, respectively), with slightly smaller values observed in faulty modules (low −0.5%, high −1.2%). This indicates that voltage carries low to insignificant predictive power for optical failures.

In regard to temperature, a larger range of values is observed in faulty modules (22.49 – 49.16 °C), compared to healthy modules (23.97 – 45.97°C). Inter-module temperature variability is also higher in faulty modules (1.516°C, +75%).

In terms of power, 95% quantiles of Rx and Tx power are considerably larger in faulty modules (7.213 dBm vs. 1.575 dBm for Tx power, and 7.185 dBm vs. 1.575 dBm for Rx power), and the same applies for 5% quantiles to a smaller extent (−3.366 dBm vs. −2.889 dBm for Tx power, −12.27 dBm vs. −4.133 dBm for

(a) Mean

	Attribute	Healthy		Faulty	
		low	*high*	*low*	*high*
DDM Metrics	Bias Current	5.000	38.71	6.000	84.20
	Voltage	3.260	3.380	3.245	3.341
	Temperature	23.97	45.97	22.49	49.16
	Rx Power	-4.133	1.969	-12.27	7.185
	Tx Power	-2.889	1.575	-3.366	7.213
OS Metrics	Error Rate	0.0000	0.0000	0.000	0.1325
	Packet Loss	0.0000	9.900e-5	0.000	0.0000
	Rx Throughput	1.456	8.569	2.206	9.680
	Tx Throughput	2.978	8.581	2.425	9.850

(b) Standard Deviation

Attribute	Healthy		Faulty	
	low	*high*	*low*	*high*
Bias Current	0.0000	0.3508	0.0000	0.5342
Voltage	0.0000	4.878e-3	0.0000	5.495e-3
Temperature	0.1695	0.8674	0.0491	1.516
Rx Power	0.0074	0.0577	0.0047	0.3832
Tx Power	1.060e-4	0.03714	2.071e-3	0.05754
Error Rate	0.0000	0.0000	0.0000	0.7357
Packet Loss	0.0000	0.04425	0.0000	0.0000
Rx Throughput	9.992e-4	0.6122	0.01811	1.159
Tx Throughput	2.780e-3	0.6522	0.02613	1.287

Table 4.4 Typical value range (5th-95th percentile) for optical module aggregate statistics (mean and standard deviation) [317].

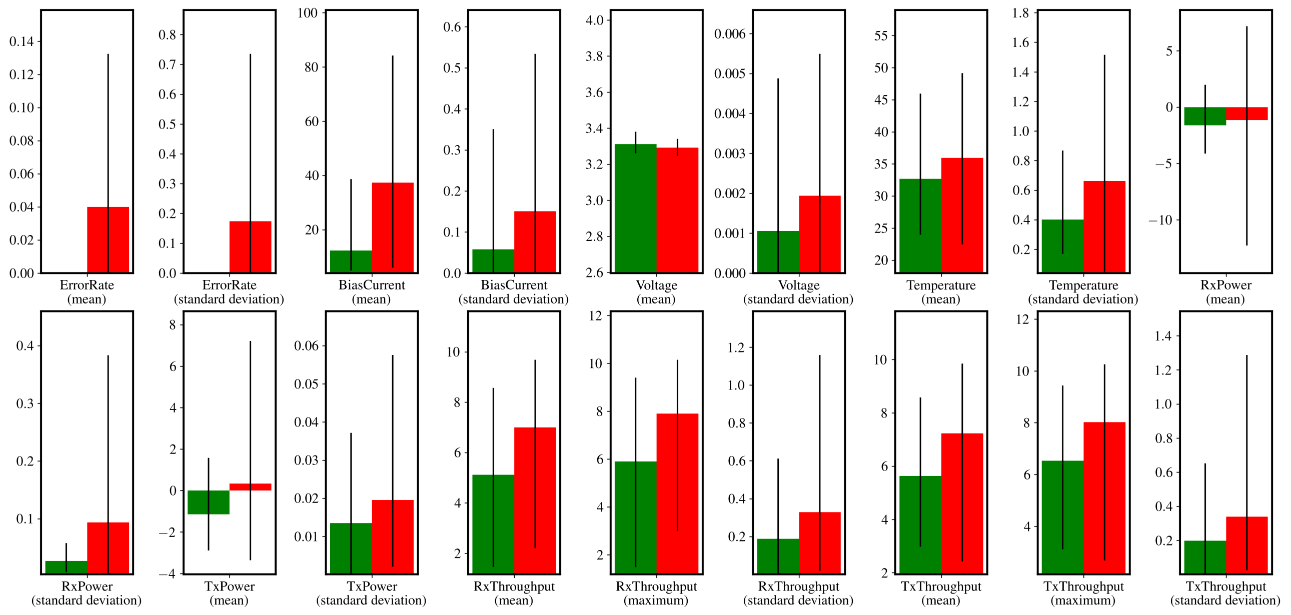


Figure 4.8 Typical value ranges for a selection of aggregate metrics [317], by class of optical modules (faulty in red, healthy in green). Samples belonging to the same module are aggregated using min, max, average and standard deviation functions over the time axis. For each metric, the red and green bars represent the mean value, while the brackets represent the 5th and 95th percentile, all computed over the two module populations. Only the aggregate metrics with the largest inter-class differences are shown.

Rx). This indicates how soon-to-fail modules are more likely exhibit anomalous Tx-Rx power characteristics compared to modules that will not fail.

With respect to OS-level metrics, any positive error rate value is outside of normal operating range for healthy devices (0.00 pps), while error rate ranges up to 0.1325 in faulty ones. Therefore, any observation of non-zero error rates may indicate future failure. Negligible, positive packet loss is only observed in healthy modules (up to 9.90e-5 pps). In terms of interface throughput rates, healthy modules typically exhibit lower throughput rates compared to soon-to-fail modules. Throughput rate standard deviations are one order of magnitude lower in healthy modules.

Failure Patterns

The frequent patterns behind the appearance of optical transceiver failures have also been subject to study.

In this experiment, the likelihood of failures in co-appearance with different metric patterns was studied. For this discussion, a pattern is any valid Boolean test on attribute values that is satisfied in at least one module (see also Section 2.3.3). One example of such patterns may be `BiasCurrent_HIGH`, where `HIGH` indicates that the mean bias current value for such instance is higher than predefined normality threshold (e.g., 95th percentile). Given a specific pattern X observed in faulty modules, the lift [88] of such pattern is measured as described in Section 2.3.3.

Patterns to monitor are constructed based on the following criteria:

- `HIGH` values (e.g., `BiasCurrent_HIGH`), to indicate a mean value > 95th percentile in the observation window.
- `LOW` values (e.g., `voltage_LOW`) to indicate a mean value < 5th percentile in the observation window.
- `HIGH_VARIANCE` (e.g., `TxPower_HIGH_VARIANCE`) to indicate a variance > 95th percentile in the observation window.

For reference, 5th and 95th percentile values are reported in Table 4.4 and as confidence intervals in Figure 4.8. Figure 4.9 lists all the positive-lift associations discovered between the observation window aggregate

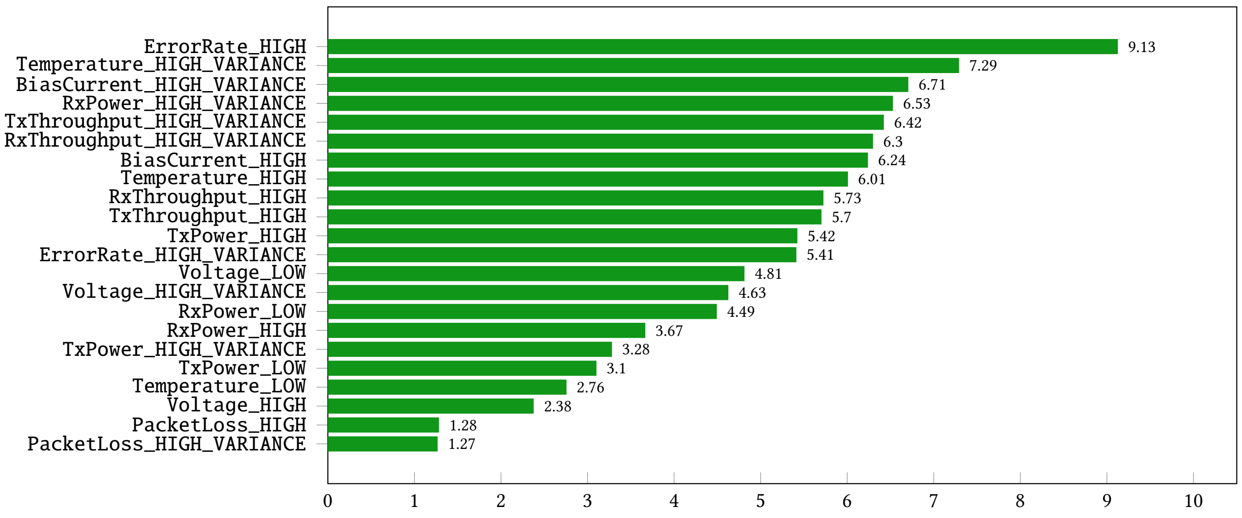


Figure 4.9 Lift [88] of explored DDM and OS metric value patterns [317]. Higher lift indicates stronger association to future failures. Lift equal to 1 indicates variable independence.

data and future failure, by decreasing lift values. The pattern with the highest lift is high error rate, which increases failure probability 9.13 times. In general, high variability in the measured metrics increases the chances of observing failures, ranging from 1.27x for packet loss, up to 7.29x for temperature. Regarding voltage, the increase in failure probability (4.63x) is consistent with what was observed by [309] for voltage variability before failure. High values of bias current correlate with future failures (6.71x). This is consistent with the working principles of SFPs, with manufacturer specifications and previous research studies [306,344]. Similar considerations apply to temperature (6.01x) and Rx/Tx throughput (5.73x and 5.7x, respectively). According to this analysis, both high and low voltages increase chances of failure, although low voltage has a larger impact (4.81x vs. 2.38), as also highlighted in the AFR differential study. Anomalous values of Rx/Tx power also impact failure probability, with lifts ranging from 5.42x to 3.1x. Packet loss-related patterns exhibit significantly lower lifts compared to the other metrics, indicating lower to no influence in the occurrence of transceiver failures.

Attribute Feature Importance

In conclusion, the importance of different monitoring attributes is evaluated using feature selection techniques. Feature importance is measured using two techniques: Mutual Information (MI) and Logistic Regression (LR) coefficients.

For the MI experiment, the class label (1=failure, 0=healthy) is treated as a target discrete variable to correlate with input continuous features.

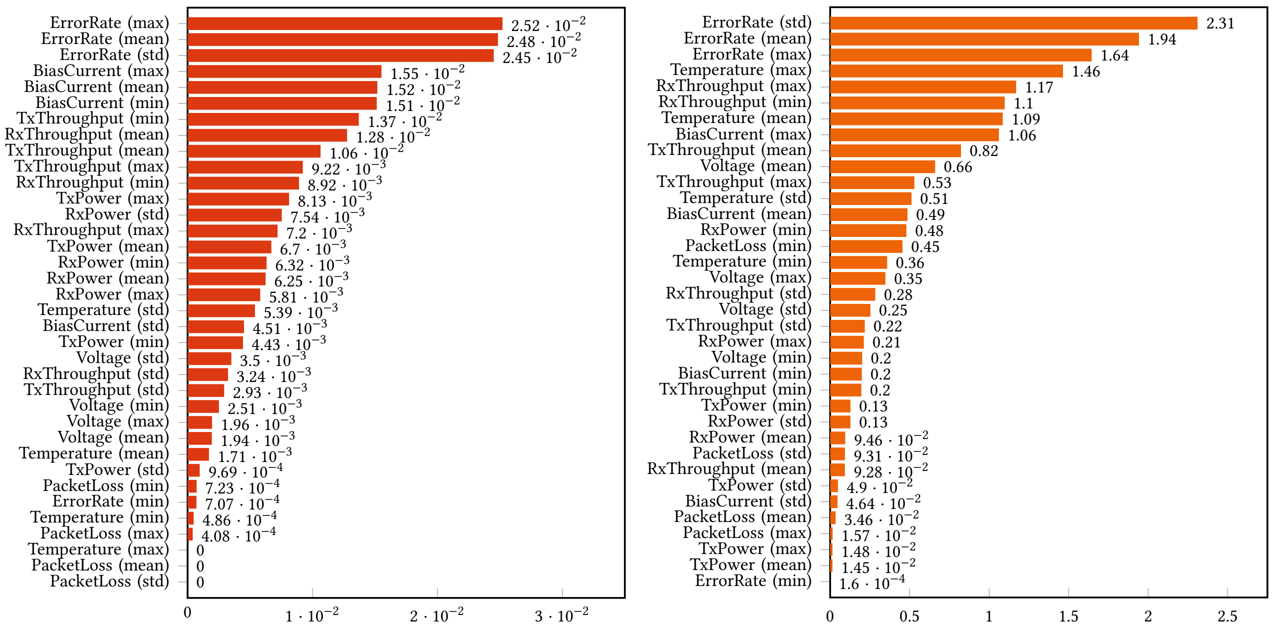
For the LR experiment, a LR classifier is trained using the class labels and the learned weights w for each feature are analyzed. In both cases, a higher factor indicates stronger predictive power towards failures.

Figure 4.10 reports the results of the two experiments. ER is the prevailing feature in both analyses, appearing in the highest rank in terms of mean and standard deviation. Other important features selected are bias current, temperature, Rx throughput and Rx power. These experiments also show that packet loss does not show any significant impact for failure class correlation.

Result Summary

All the empirical observations discussed in the reliability study [38] and described in this section are here summarized.

- *Failure rate of optical transceivers (0.13–1.65%) is lower than other hardware components*, such as HDDs (1.7–8.6%) [288] or DRAMs (0.22%) [292], however, the failure characteristics of optical transceivers is still prone to determine performance degradation because of the high presence of soft failures;



(a) Mutual Information (MI) **(b)** Logistic Regression (LR)

Figure 4.10 Attribute feature importance for several optical DDM and OS aggregate metrics [317], measured using (a) Mutual Information and (b) Logistic Regression weights.

- *Error Rate is the strongest indicator of future failure.* It was observed how non-zero error rate is observed exclusively in soon-to-fail modules, and high values of error rate show the strongest statistical association with future failures. Moreover, in the time cross-correlation study, it was observed how error rate auto-correlates with itself, showing a predisposition to re-occur after previous encounters. Error rate features are also the highest scoring features in the feature importance analysis.

- *High values of bias current increase the probability of future failure.* AFR increases 8.36 times in the presence of high bias current values, and bias current patterns are 6.24 times more frequent in faulty modules compared to healthy modules. It was also observed how 90% operating range of faulty modules covers much larger values of high bias current (< 84.20 mA) compared to the same operating range for healthy modules (< 38.41 mA). This indicates that observing values higher than this second threshold, may predict for an imminent failure.

- *Higher-than-usual variability (95th percentile and above) in all 5 dynamic DDM attributes, as well as error rate, indicates a higher probability of future failure.* This has been observed for error rate, Rx/Tx throughput, Rx power, temperature and bias current in both typical healthy/faulty range analysis and failure pattern analysis.

- *Anomalous high and low values of Rx and Tx power are correlated with an increase in failure probability.* This has been observed in the AFR study and in the operating healthy/faulty range analysis and failure pattern analysis.

- *Low voltage values are correlated with an increase in failure probability.* This is observed in the AFR study (3x) and in the failure pattern study (4.81x).

- Packet loss does not show any correlation with optical transceiver failures.

4.7.5 Optical Transceiver Failure Prediction

Using the statistical insights describe above, OFP models can be constructed to estimate the presence of future transceiver failures.

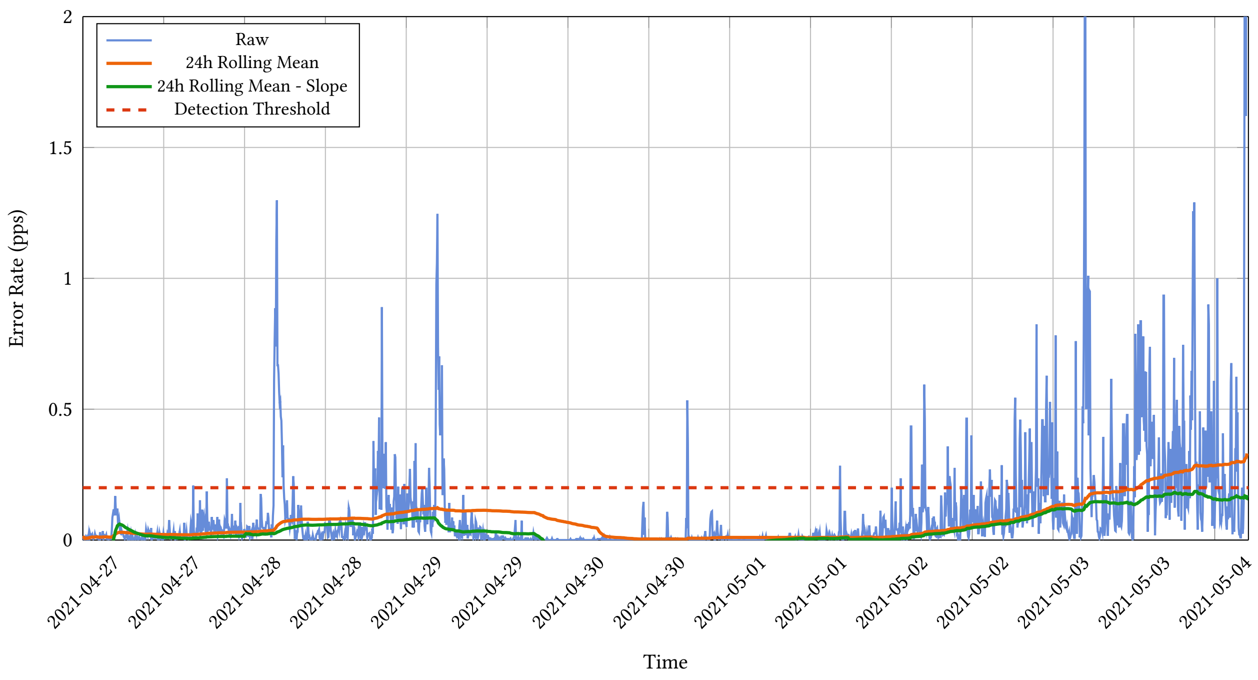


Figure 4.11 Example of OFP in optical transceivers using the ER trend predictor [317]. The blue line represents the error rate measured in a soon-to-fail module. The red line indicates the detection threshold for an upcoming failure.

For model evaluation, the temporal model described in Section 4.3.1 is used. The predictor lead time Δ_{t_l} is set to 1 day, the prediction window size Δ_{t_p} to 7 days, and the observation size Δ_{t_w} to 31 days. These time windows are consistent with the O&M requirements imposed to monitor and operate on faulty transceivers.

For the purpose of the failure prediction study, an optical module is considered faulty if it has experienced any of the two types of failure described in Section 4.7.2. Consequently, the observation windows related to a faulty module prior to the failure are labeled as the positive class, while observation windows of modules that did not fail during operation are labeled as the negative class (healthy). Same applies for faulty modules before sufficiently in advance with respect to their failure period.

After defining all the prediction parameters, and therefore assigning each observation window to one of the two classes, prediction models are evaluated by comparing the assigned class to the prediction class on test samples. To evaluate failure predictors, the metrics defined earlier in Section 2.3.5 (accuracy, precision, recall, and F1-score) are used.

Classical ML predictors

In the reliability study [317], different classical ML algorithms, including random forest, XGBoost and logistic regression were tested on the module dataset described above (Section 4.7.4). These classification algorithms were selected because of their high interpretability and performance, in addition to their ability to model complex variable interdependencies. For these models, the aggregate functions discussed in Section 4.7.4 are applied to all dynamic DDM and OS attributes, to reduce each module observation window to a set of scalar features. The addition of static categorical attributes (wavelength, distance, etc.) did not show any additional benefit in the experiments, therefore they are discarded for this experiment.

In addition, a threshold-based online trend detector is proposed, for fast prediction of error rate degradation failures. This model computes the 24h rolling window of the error rate, and then extracts the rate of increase of this time series. If the slope reaches a predefined threshold (in these experiments, 0.15), the module is expected to fail soon. For this model, only the ER time-series observation is used without applying any reduction function. An example of error rate trend prediction is shown in Figure 4.11.

The module dataset used for studying metric failure patterns is split into train and test sets (80–20% split), the ML models are trained on the training set, and evaluated on the test set. After discarding modules with

Model	Module test dataset (30/1776 faulty)			
	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)
Random Forest	99.4	88.9	47.1	61.5
XGBoost	99.3	91.7	50.0	64.7
Logistic Regression	99.1	72.8	36.4	48.5
ERTrendEstimator	99.6	95.7	21.2	34.6

Table 4.5 Test set evaluation results for different classical ML models for the OFP task [317].

invalid or missing data and after the train-test split, a final test dataset of 1776 modules (30 positive samples) is obtained.

Table 4.5 reports the results of the test set evaluation procedure. The tested classifiers achieve precision values ranging from 72.8% to 91.7%, indicating high specificity of the monitored data for the prediction task. The achieved recalls range between 21.2% and 36.4%. This indicates how the monitoring information considered is not sufficient to predict all failures, but can be used to effectively predict a significant fraction of failures.

Deep Learning Predictors

The possibility to apply neural networks for optical transceiver failure prediction has also been investigated.

RNNs are a natural candidate for time-series analysis problems, as they allow to operate on sequences of variable length and model the long-term dependencies between different time step observations. To this end, a recurrent neural network based on LSTM cells was implemented and evaluated.

Figure 4.12 depicts the neural network architecture employed. Both static and dynamic attributes from the module optical dataset are used, although only the attributes that have proven to have diagnostic values for transceiver failure prediction are given as input. The 8 dynamic attributes (DDM + OS) are min-max normalized and are concatenated to construct a multivariate time series, which is processed sequentially by the LSTM layer. The static attributes are one-hot encoded and fed directly into the fully-connected layer in addition to the LSTM output. This allows to process both sequential and static information without repeating the module metadata in each sample. The final output of the network is a binary class prediction (1=faulty, 0=healthy).

The network is trained in a supervised learning setting using a weighted binary cross-entropy loss function. The class weights of the loss function are determined based on the inverse frequency of the classes. This ensures that the positive class (faulty), which represents a minority of samples, is also learned during training in order to recognize the failure patterns. The network also incorporates ridge (L2) regularization via weight decay, dropout, and early stopping based on minimum validation loss.

Two main experiments are performed to evaluate the validity of the deep neural network approach: a comparison analysis with rule-based approaches, and an analysis of prediction window lengths, to estimate the effective warning time for failures.

For these experiments, the dataset resulting from Collection 2 and 3 (see Section 4.7.4) was used as training set, amounting to a total of 3684 modules (128 faulty). A new test set, composed of 838 new modules (14 faulty), is collected from datacenter B with the purpose of evaluating the classifier accuracy. Each module time series contains 2000 observations, corresponding to approximately 1 week of observation at 5 minute resolution.

In the model comparison analysis, the LSTM network is compared with a rule-based approach based on ER and a hybrid LSTM-rule-based classifier, which combines both predictors into one.

The rule-based system returns a positive prediction according to the rule `any(error_rate_5min>0)`, i.e., it returns a positive prediction if any CRC error rate bucketed in bins of 5 minutes was ever observed for any 5-minute sample in the observation window. This rule has been proposed by O&M experts and it is used for predictive maintenance of optical transceivers. Other threshold values for this rule have been evaluated, but they did not yield more accurate results than the expert threshold of 0. The hybrid approach returns a positive prediction if *any* of the two above-mentioned predictors (LSTM and rule-based) returns a positive prediction.

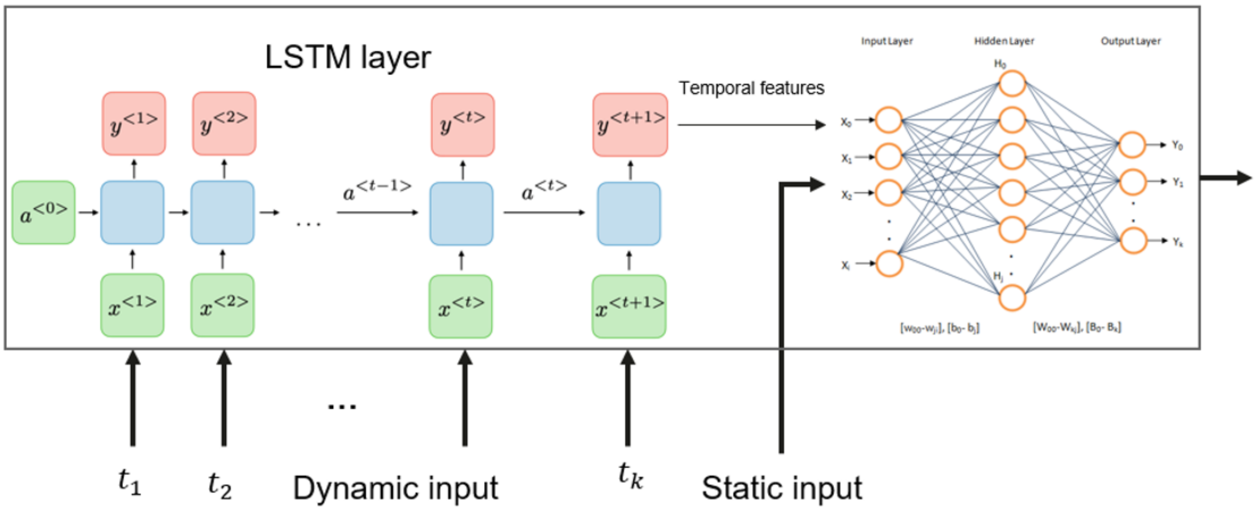


Figure 4.12 LSTM architecture used for optical transceiver failure prediction experiments. The dynamic input attributes observed for a single module are used to construct a multivariate time series x. This series is fed as input to the LSTM layer. The LSTM output of the last time step ($y^{<t+1>}$) is then fed into a fully-connected layer for binary. This intermediate output is also concatenated with the static information about the module described in Section 4.7.4.

.

In this set of experiments, a distinction between two types of failures is made: the ability to detect sudden faults, i.e., optical failures that do not show any precedent sign of CRC, is compared to the detection CRC-anticipated failures, i.e., failures which showed preliminary signs of signal degradation. The former are in general more difficult to detect and predict accurately.

Table 4.6 reports the results of the test set evaluation for the three described models, divided by the two categories of failures. All models can effectively pinpoint CRC-related failures with high precision (100%). In particular the rule-based model, which only analyzes ER, could detect 10 out of 14 failures in advance (71.43% recall). Although the LSTM model can only detect 50% of CRC-anticipated failures, some of the failures detected by the LSTM were not detected by the rule-based model. By incorporating both models into a hybrid model, the highest number of failures may be identified preemptively, by simultaneously exploiting the simple ER test of the rule-based approach and the more complex DDM and OS information using the LSTM.

For the evaluation of all failures, including sudden failures, a 20% of the Collection A is kept aside to construct a test set with both CRC-anticipated failures and sudden failures (with 21 faulty modules). From this 129 modules of Collection A, data augmentation is performed to obtain a set of 1290 random crops of time series of length 2000, i.e., from each module 10 random crops of 2000 consecutive time steps are taken with replacement. Because some crops of faulty modules correspond to time periods far away from the module failure, the 1290 crops are labeled based on the condition $t_{fail} > t_{max} + \Delta_{t_l} + \Delta_{t_p}$, where t_{max} indicates the last available time step of the random crop. In this way, faulty modules that have not yet manifested symptom of failure are effectively treated as healthy. The total number of faulty-labeled crops is 155.

Model	All Failures (155/1290 faulty)				CRC-anticipated Failures (14/838 faulty)			
	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)
Rule-based	93.64(1208)	100.00(155)	47.10(73)	64.04	99.52(834)	100.00(10)	71.43(10)	83.33
LSTM	94.42(1218)	86.73(113)	63.23(98)	73.13	99.16(831)	100.00(7)	50.00(7)	75.00
Hybrid (LSTM + rule)	94.18(1215)	82.26(124)	65.80(102)	73.12	99.64(835)	100.00(11)	78.57(11)	88.00

Table 4.6 Test set prediction results for the LSTM, rule-based and hybrid models on the OFP task, evaluated in terms of accuracy, precision, recall, and F1-score (numbers in parentheses indicate absolute prediction counts). On the right side, the evaluation results for CRC-anticipated failures, on the left side the evaluation results for all types of transceiver failures.

Δ_{t_p} (days)	LSTM - All Failures (155/1290 faulty)			
	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)
3	88.29(1139)	52.17(92)	30.97(48)	38.87
7	94.42(1218)	86.73(113)	63.23(98)	73.13
14	92.87(1198)	60.95(105)	55.65(64)	58.18
30	92.71(1196)	58.27(127)	64.35(74)	61.16

Table 4.7 Test set prediction results for the LSTM model on the OFP task, with different prediction length windows (Δ_{t_p}), evaluated in terms of accuracy, precision, recall, and F1-score (numbers in parentheses indicate absolute prediction counts).

In these results, it can be observed that the rule-based approach retains optimal precision (100%), with recall decreased to 47%, indicating that the rule is not able to detect sudden faults as accurately. The LSTM model again shows improved recall (63%, +16%) at the expense of a 13% decrease in precision. When compared with the results of the other category of failures, the prediction results of the LSTM model are similar (73% vs. 75% F1-score); similar results (F1-score=73%) are also obtained by the hybrid model on all failures, indicating that the contribution of the rule-based approach to hybrid model is negligible (compare with 73% of LSTM alone).

In summary, the results of LSTM-based models are improved over the results with classical ML models (73% vs. 65% best F1-score); yet similar conclusions can be drawn: results indicate that optical transceiver failures can be predicted with high precision for a significant fraction of failures (47–79%), and that incorporating component metrics into the model improves recall (+21–29%).

In a second experiment, the effect of different prediction length windows on prediction accuracy was tested, to estimate the optimal range of predictions to effectively use the LSTM model in real O&M contexts. In this experiment, the LSTM model is trained to predict failures in advance for prediction windows of size $\Delta_{t_p} = 3, 7, 14, 30$ days. The lead time is set to $\Delta_{t_l} = 1$ days. The same training set of 1290 crops is used for training, with corresponding labels updated to reflect the new prediction windows (i.e., if a module fails in x days, it is labeled faulty only if the prediction window extend beyond day x).

Table 4.7 reports the results of the prediction window experiment. The prediction window with the highest prediction performance is 7 days, while both shorter (3 days) and longer windows (14, 30 days) show inferior results. This ambiguous non-monotonic dependency on window length may be the result of two compensating effects: the increasing probability of observing a failure in the prediction window and the decreasing ability of the classifier to recognize failures far away in time. For the first effect, the total probability of observing a failure in the prediction window increases by enlarging the size of the prediction window. This renders the predictor on average more accurate, as there are more possibilities to guess a failure correctly. For the second effect, by extended the acceptable observation window for a failure it also becomes increasingly difficult for the model to understand the patterns leading to the failure, as these patterns manifest earlier in time (with respect to the failure time). This renders prediction more complex and less accurate.

The resulting outcome is that a 7-day prediction window constitutes a sufficiently good compromise between the two effects and provides the optimal results. Future experiments may verify the exact impact of each of these two effects by simultaneously adapting the lead time to prediction window size (e.g., $(\Delta_{t_p} = 7, \Delta_{t_l} = 1), (\Delta_{t_p} = 10, \Delta_{t_l} = 3), (\Delta_{t_p} = 30, \Delta_{t_l} = 23), \ldots)$ so to observe only the impact of the second effect.

4.7.6 Outcomes

The outcomes on the exploration of optical transceiver failure prediction are here summarized.

The annualized failure rates for optical transceivers were estimated, and the appearance of failures was statistically correlated with patterns observable in the monitoring data [317]. The results indicate the presence of strong symptoms of future failure with different degrees of correlation, including past observation of positive error rate, high values of bias current and anomalous (high or low) values or Rx/Tx power, high variability of voltage, temperature and power attributes. Different types of ML predictors have been constructed and evaluated. The results show that effective OFP can be achieved for optical transceivers, which can cover a significant number of future failures (47–79%) with high precision (73–100%). Based on the cost considerations discussed in Section 4.3.3, these results render optical transceiver failure prediction effective and economically viable, due to demonstrated high soft AFR of transceivers, the high prediction accuracy of optical OFP models, and the reduced cost of replacement, compared to the cost of network disruptions.

Considerations regarding accuracy, deployment cost, and ease of implementation may influence which of the described predictors is more suitable to be used in cloud production systems. While rule-based and trend-detectors approaches are simpler to implement and impose a negligible overhead on the monitoring system, more advanced methods (such as classical ML models and LSTM) provides more precise and exhaustive predictions, by leveraging additional information. These methods require additional (and ideally separate) hardware resources to be effectively be used in production.

In the analysis, a potential correlation with device age was not included. It could, however, prove to be a fundamental factor to predict component failure, in addition to the inclusion of additional DDM information, such as attribute warning thresholds. Potential future work may cover these new data sources, as well continue the exploration on the impact of time resolution on results, by employing different observation windows, lead time, and sampling periods. Moreover, an analysis of runtime performance, in terms of inference time and resource utilization, may further improve the estimation of the cost-benefits of executing these OFP algorithms in production.

4.8 Conclusion

In this chapter, infrastructure-level proactive failure management has been discussed. An architectural model of the infrastructure layer was defined, to illustrate how and which infrastructure failures propagate in hierarchical levels of a cloud stack model. Different remediation actions and methods in related work for managing infrastructure failures have been discussed. The specific impact of hardware-related failure was discussed, including estimates of component failures based on existing literature.

A framework for Online Failure Prediction (OFP), composed of a temporal model, a cost model, and a failure model was introduced. Then, different instances of OFP on hardware components were presented. Optical transceiver failure prediction was explored, based on the results of an own reliability study, and by integrating the statistical insights derived into rule-based and ML algorithms for failure prediction. Hard drive and memory failure prediction were equally discussed and presented.

The results presented in these last sections demonstrate the applicability of OFP methods for enhancing cloud reliability at the infrastructure level. In particular, predictors are shown to achieve accuracy results, in terms of precision and recall, which are in accordance with the cost requirements and considerations described in the OFP framework.

The optical transceiver analysis results indicate that high actionable insights may be derived from the observation of DDM metrics. The optical transceiver predictors described here are also the first to consider the impact of workload on the failure characteristic, achieving high-precision results which render them suitable for real-world operation.

5 Platform-level Proactive Failure Management

This chapter discusses the application of proactive techniques for failure management at the platform layer of cloud systems. A background on the platform layer is introduced in Section 5.1. Section 5.2 introduces the problem of Command-Line Interface (CLI) security. Related work on platform-level proactive failure management is summarized in Section 5.3. Section 5.4 discusses Natural Language Processing (NLP)-based methods for command risk classification, presenting the approaches, the evaluation results and the applications to platform-level Failure Management (FM). Section 5.5 summarizes the outcomes of this chapter.

5.1 Introduction

5.1.1 The Platform Layer

While the infrastructure layer is responsible for providing low-level resources, such as hardware, memory, and computing power, the platform layer must provide high-level resources to build and execute applications. These include, but are not limited to, the guest Operating System (OS) and its tools (e.g., the Bash command-line interpreter in Linux), libraries (e.g., a deep learning framework), build systems (such as `gcc` or `javac`), runtime environments (e.g., Java Runtime Environment (JRE), or Python *virtualenvs*) and other middleware requirements, e.g. SQL databases, messaging systems, etc. Platform-as-a-Service (PaaS) offerings may also provide uses with the possibility of installing their own tools. However, the consumer does not have control over the underlying infrastructure. In short, PaaS must provide the ability to develop, test, run, and host application flawlessly [40].

The key requirements of an optimal platform-level offering include:

- *security*, the ability to protect sensitive information (such as login access, customer data, proprietary software, ...) and prevent the manipulation of such data
- *scalability*, the ability to sustain a variable workload;
- *continuous delivery and integration*, the ability to support quick delivery of new features, including automated tools for building, testing and integrating code;
- *reduction of programming effort*, the ability to support development of software, through efficient programming environments and source code (e.g., syntax checking, edge-case detection, ...).

5.1.2 Platform-level Failures

Platform-level failures encompass a variety of issues that impact the operation and availability of the cloud platform. These failures can disrupt services, compromise data integrity, and affect user experiences. Some common types of platform-level failures include:

- *database failures and data corruption.* Failures in database systems can result in data unavailability or corruptions, or inconsistent data states. Data corruption can have severe consequences, especially in critical applications. These failures may be due to hardware faults, software bugs, or improper database design;
- *incorrect service dependencies and compatibility issues.* Cloud platforms rely on various interconnected services. If a dependency service experiences failures, its failures cascade across the platform, impacting the overall functionality of the platform. Moreover, incompatibilities between different software

versions, libraries, or Application Programming Interfaces (APIs) can lead to runtime errors, service disruptions, or application crashes.

- *security issues and vulnerabilities.* Security vulnerabilities or breaches can compromise the confidentiality, integrity, and availability of platform services and data. This can include unauthorized access, data leaks, or Distributed Denial of Service (DDoS) attacks;
- *software bugs.* Software built to serve platform-level offerings may have bugs or flaws that lead to crashes, performance degradation, or incorrect behavior. These bugs might originate from the platform software itself or from third-party applications used for the platform;
- *configuration errors.* Improper configuration of platform components can lead to unexpected behavior or vulnerabilities. Configuration errors might result from human errors or mismanagement of automated deployment systems;
- *deployment failures.* Applying updates or patches to the platform software can sometimes introduce compatibility issues or unexpected behavior, leading to service disruption.

Understanding these different platform-level failures is fundamental for devising an effective proactive failure management strategy. By identifying potential risks and utilizing AIOps techniques, Operations & Maintenance (O&M) teams can anticipate and mitigate these failures before they impact users and services.

5.1.3 Platform O&M

The different types of failures above-mentioned may originate directly at the platform layer, or may originate in the infrastructure layer (as described in Section 4.1.2), where they could not be totally neutralized. In both cases, they may have severe consequences on user experience by affecting on cascade all interested middleware and software systems, and for this reason they must be protected by O&M teams. These systems include:

- the guest OS, which is directly communicating with the infrastructure layer and might be affected by a failure in this sense, or it might itself cause a runtime exception;
- middleware, i.e., interpreters, such as database engines (Big Query), Just-In-Time (JIT) compilers, REST APIs are correctly working and integrated as intended;
- the software runtime, i.e., the instances of execution of a software program, for which it must be ensured that sufficient resources are allocated and the correct runtime environment is used.

When carrying out O&M operations, it is therefore vital to ensure that:

- O&M operations performed satisfy the requirements expected from the platform-level offering. This includes, for instance, verification of correct deployment of new functionalities, verification of service and runtime availability;
- O&M operations performed do not interfere with the correct execution of platform-level customer workloads. In particular, it is important to minimize the emergence of failures during the execution of O&M interventions.

The platform is therefore the environment of building and deployment operations, which must be carried out continuously to ensure services can be run smoothly.

The large majority of high-level tools for O&M above are effectively enabled through CLI operations. O&M operators utilize CLIs to access remote hosts daily and perform large quantity of terminal-based operation to build, deploy, maintain, configure, delete and stop software tools required to operated the platform environment. OS is configured over the CLI, runtime environments are set up over the CLI, OS-level virtualization and orchestration systems are managed through the CLI, and software is compiled and executed over CLI. Therefore, the correct use of CLI plays an important role in ensuring the correct operation of the platform layer.

5.2 Command-line Security

The CLI serves as the gateway through which O&M administrators interact with and control the intricate components of cloud environments. The CLI also encapsulates the core operations required for effective platform-level failure management. It provides the means to execute a variable number of operations to address operational challenges and maintain the robustness of cloud services. Nearly every operational issue that arises within cloud platforms can be traced back to the CLI, and correspondingly, many of these issues can be effectively resolved through the skillful use of CLI commands.

Because of this high actionability, it is fundamental to ensure that CLI operations satisfy security requirements and do not impact negatively on the reliability of deeply interdependent platform systems.

Operator error is the leading cause in two of out of three services [346]. O&M requires operators to access remote systems to configure and repair services via CLI. Providing free access to remote production systems poses two major security challenges, namely:

- the possibility of external attackers gaining unauthorized access to systems. Cybercrime represents the fastest growing cause of data center outages [318];
- the possibility of operators accidentally performing harmful operations and damage service availability (human error).

For this reason, operators typically access remote hosts via a bastion host, which allows to review, approve, and execute commands without establishing direct connection to systems [347].

A bastion host is a gateway computer system that prevents dangerous and malicious operations from being executed. Bastion hosts typically run an interception system, which captures executed commands and analyzes them before granting execution rights. The interception system must be able to accurately recognize and extract CLI commands from the payload of requests, while ignoring other inputs (e.g., logins, yes/no prompts, shell auto-completions).

Bastion hosts also provide access control functionalities, by associating users to privilege policies which may allow a specific operator to execute a privileged operation, which would otherwise not be allowed. However, highly dangerous commands must always be blocked. The recognition and halting of dangerous commands before their execution requires an algorithm to estimate command risk.

The evaluation of command risk is an open and challenging problem, due to large variability in executable programs and their combination of arguments, flags, and environment variables (see Table 5.1). Each command may be composed of a series of multiple *fragments*, each corresponding to an executable program with its flags and arguments, that are concatenated in different ways (pipelining, &&, multi-line commands, …) to obtain a compound result. Evaluating the risk requires to understand the behavior of each fragment, by recognizing program names and associating options and arguments correctly.

Different risk classes may be defined, and each class may be associated with the necessary privilege to execute commands. Command risk classification is the association of incoming commands to one of the predefined privilege and risk classes. Based on the classifier decision and the current user privileges, the

Command	**Risk Class**	**Notes**
`rm 20230421_12:45:67.log`	LOW	deletes temporary log
`rm rf /bin/*`	HIGH	(force) deletes executables dir
`cat $DELETE_LIST | grep *.log`	LOW	prints filtered content
`cat $DELETE_LIST | xargs -0 rm`	HIGH	deletes files from input file
`time ls`	LOW	(timed) `cwd` directory listing
`time kill -9 12345`	HIGH	process kill hidden by `time` call
`echo ‘kill 7890’`	LOW	prints a string
``echo `kill 7890` ``	HIGH	backtick evaluates `kill`

Table 5.1 Examples of commands with varying complexity and risk [345]. It can be easily observed how small changes to parameters, flags, and some uses of the command-line syntax highly influence the risk of executed commands.

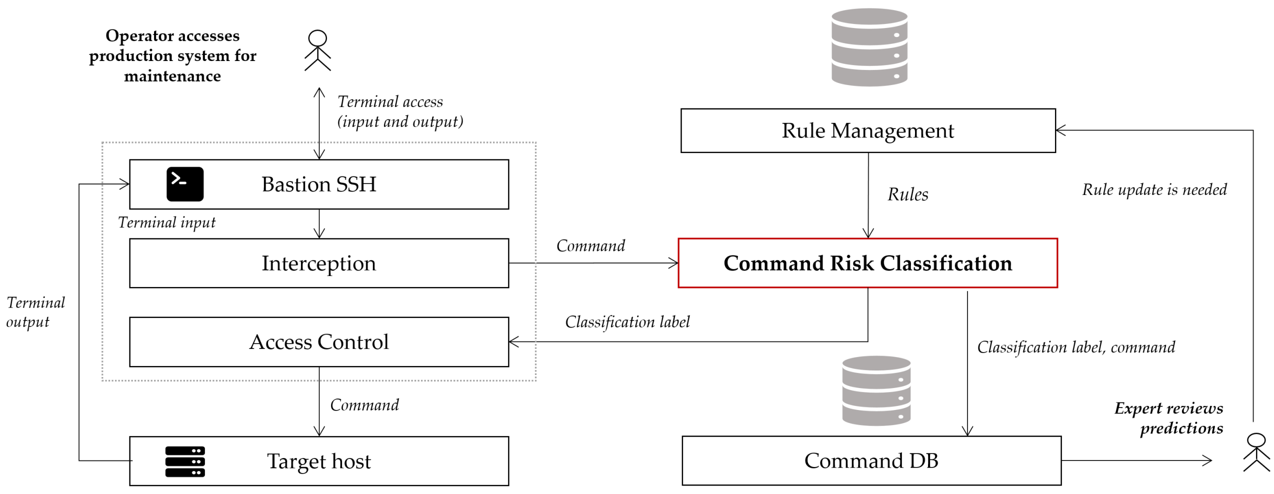

Operator accesses production system for maintenance
Terminal access (input and output)
Bastion SSH
Terminal input
Interception
Access Control
Terminal output
Command
Target host
Rule Management
Rules
Rule update is needed
Command
Command Risk Classification
Classification label
Classification label, command
Expert reviews predictions
Command DB

Figure 5.1 Architecture of a rule-based risk assessment system [345]. Commands executed by operators are intercepted by a bastion host (Bastion SSH) to be evaluated using a set of rules stored in a configuration database (Rule Management). If the command is evaluated safe, it is forwarded to the Target Host, otherwise an error is reported. All risk evaluations are logged and periodically revised by security experts, who may update the rules.

bastion host allows or blocks the current command, and returns output the user. It may additionally record the operation for forensic purposes.

A frequent solution for estimating command risk is a rule-based classifier, where IF-THEN-ELSE rules define which commands are allowed (or *whitelisted*) and which commands are blocked (or *blacklisted*). An example of rule-based risk assessment systems is shown in Figure 5.1. The rules may be defined based on the expertise of O&M operators and the historical records of executed commands, and stored in a configuration database that allows to periodically revise and update them. Rule-based systems may implement a specification syntax, based on regular expressions or quantifiers, to cover a larger set of commands with a single expression. An example of rule-based command risk classification is shown in Figure 5.2.

Rule-based systems are simple, easily configurable and explainable. However, they also have several limitations:

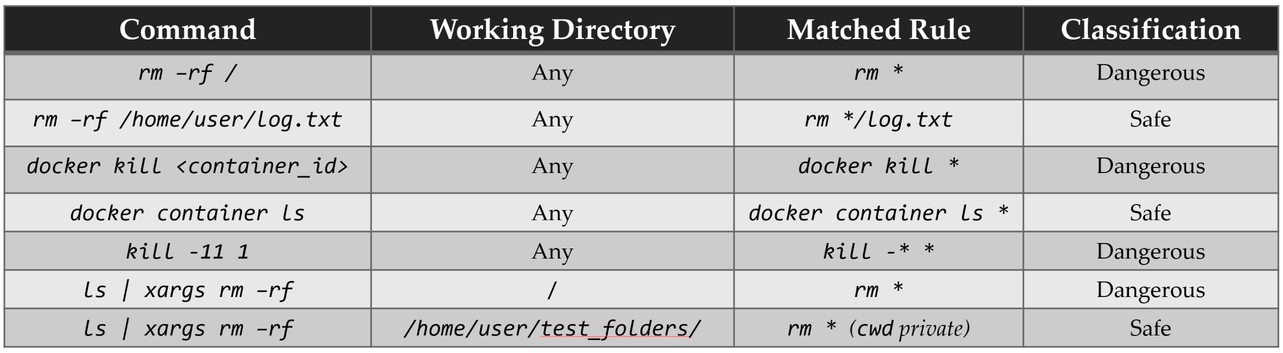
1. new combination of programs and arguments may be executed, for which the existing rules are not suitable;

2. the risk level assigned to rules by operators based on their expertise, may diverge from the true risk of commands, as described in their technical documentation;

3. they require a default handling action when a command does not match any existing rule. This default action is however limiting, as a "default block" strategy may hinder important operations during incident response, while a "default allow" may allow dangerous operations to be executed;

4. dealing with the complexity of command-line syntax is difficult without resorting to a complex pattern language for rules (see Figure 5.1).

Therefore, human supervision is required to make sure the rules reflect the risk of executed commands over time. A revision is performed by comparing commands with their corresponding predicted risk class. Since the majority of commands executed are harmless and dangerous commands are rarely being executed, the manual verification of commands is a tedious and error-prone job. The problem may be further aggravated by an inaccurate interception system, which may capture console content, such as command outputs, password prompts, auto-completions, etc. which does not constitute an executable input and must be ignored.



Command	Working Directory	Matched Rule	Classification
rm -rf /	Any	*rm **	Dangerous
rm -rf /home/user/log.txt	Any	*rm */log.txt*	Safe
docker kill <container_id>	Any	*docker kill **	Dangerous
docker container ls	Any	*docker container ls **	Safe
kill -11 1	Any	*kill -* **	Dangerous
ls | xargs rm -rf	/	*rm **	Dangerous
ls | xargs rm -rf	/home/user/test_folders/	*rm * (cwd private)*	Safe

Figure 5.2 An example of rule-based command classification in 'Dangerous' and 'Safe' classes [345]. In this example, the input command and current working directory are used to estimate risk. Rules allow to capture different commands through the use of the quantifier *.

5.3 Related Work

As most of the techniques to deal with platform failures prevent errors in software, a significant overlap of techniques with software-level FM is present. This section focuses on database failures, job and task failure prediction, and command-line security. Other relevant methods, such as deployment verification and software defect prediction, although applicable at platform level as well, are discussed in Section 6.3.

5.3.1 Database Failure Management

Few works have address failures in data management system using Machine Learning (ML) techniques. Karakurt et al. [348] develop ML models to predict failures during the operation of Oracle databases. Log files are collected directly from the database and labeled as healthy or faulty interaction, for a total of 170 days of operation. From these logs, a collection of 261 text features are extracted and used for training a random forest and other binary classifiers. Their best model achieve a precision of 84.9% and a recall of 75.7%.

Kamra et al. [349] investigate intrusion detection for database systems. Their approach is based on pattern mining analysis of executed queries, historically registered in audit logs. Queries are represented according to a structured 5-feature format. Intrusion detection is treated both as a supervised learning problem, using user profiles as labels, as well as an unsupervised problem, where anomalous operations are identified using anomaly detection techniques based on clustering. The approaches are validated on synthetic and real database traces from a Microsoft SQL server database.

5.3.2 Job/Task Failure Prediction

Several past literature works have applied ML models for failure prediction of jobs and tasks in a cloud environment.

Chen et al. [22] utilize traditional Recurrent Neural Networks (RNNs) for predicting job-level and task-level failures. They leverage static job data, such as priority and user, and historical information, such as job submission history, average resource usage and information about users related to the job, to estimate if a job or task will complete successfully. Their results show how it is possible to save 6–10% of resources by early prediction and stopping of long jobs.

Gao et al. [350] apply a multi-layer bidirectional Long Short-Term Memory (LSTM) to predict tasks and job failures. They utilize workload metrics, such as CPU and memory usage, as input data in a supervised binary classification fashion, to predict whether each job is expected to complete successfully or not. During the evaluation, they achieve 93 and 87% accuracy, respectively.

Hajiaghavi et al. [351] use a similar LSTM-based method to predict application failures by session, leveraging code and telemetry data. They also incorporate Root Cause Analysis (RCA) in their prediction system, by extracting and identify patterns that contribute and/or prevent failures. These event patterns are divided in contributors or blockers, based on their contribution. This enables the interpretation of LSTM model predictions while preserving the highly effective RNN approach (75% precision, 85% recall).

5.3.3 Command-line Security

Some previous works in the context of O&M security [352,353] have applied machine learning to command-line data for security-related classification tasks, such as malicious command detection.

Direct auditing systems [354,355] rely on human expertise to confirm if the risk assignment performed by a rule-based classifier is correct.

A technique for identifying and upholding security rules for commands executed in a O&M context is described in a patent filed by North China Electric Power University [354]. Between clients (O&M operators) and servers, an access gateway is set up so that commands are intercepted, examined, and either forwarded to the destination host or stopped, depending on the outcomes of the rule-based analysis. Data mining techniques that make use of a preliminary learning phase are used in rule-based analysis. The result is a collection of frequently occurring feature patterns seen in typical operation of the O&M system. If the features conform to any of these patterns, the command is considered safe and executed.

A Kaspersky Lab patent [355] proposes a system for auditing existing detection rules for malware based on similarity. Given a set of rules used to recognize malware files, a potentially dangerous file is evaluated against each individual rule. If any of the rules returns a match, the file is deemed a malware. The algorithm introduced in the patent can verify the validity of a classification rule using test (A) and control (B) groups, containing malicious and non-malicious files respectively. Given a rule R that must be evaluated, R is first applied on an unknown set of files C. The rule classifies files in C as either malware or non-malware. Files classified as malware via R are called rule matches in C. The similarity algorithm used for auditing is applied on these rule matches, to test whether they are sufficiently similar to A and dissimilar to B. If the matches are similar to the files in A and different from the files in B, the rule is reliable. If the matches are not similar to the files in A, or too similar to the files in B, the rule is not reliable. In the latter case, the rule would return a lot of False Positive (FP) predictions because B is a set of non-malicious files. Therefore, by applying A/B testing on the two groups, it is possible to discover false positive classifications and improve the rules under test. The files not matched by R in C are ignored. Files not detected by the audited rule are not analyzed, although they might still contain undetected malware files (False Negative (FN)). Therefore, this method can only discover FP predictions and cannot discover FN.

Direct risk classification systems [352, 353, 356] apply machine learning classifiers to predict the risk of commands. By providing a more accurate classification, prediction of the ML classifiers can be used to create and update rules of a traditional pattern-based system.

Hendler et al. [352] evaluate several machine learning models for malicious PowerShell command detection. Both traditional NLP (n-gram, Bag-Of-Words (BoW)) and deep neural network models (Convolutional Neural Network (CNN) [357], LSTM [6]) are considered, including an ensemble combining a 3-gram and a CNN model, which yields the best performance. During the pre-processing phase, characters are one-hot encoded, with case information at character-level provided as an input binary flag. The CNN model applies 1D convolutional layers based on the architecture proposed by [357] on input commands padded to fixed length. However, the use of character-level one-hot encoding with a closed vocabulary does not allow to effectively model the semantics and inter-relationship between input tokens.

Yamin et al. [358] use Naive Bayes and CNN models on command-line arguments to classify PowerShell commands as malicious. They evaluated their classification accuracy on a dataset composed of 14 categories of commands. Their approach focuses on PowerShell and obfuscated command detection. Results are evaluated in terms of accuracy (96%) and the dataset class distribution is not provided. Because dangerous commands are rare, the class distribution is typically highly unbalanced, which allows to construct trivial classifiers that can achieve high accuracy.

To the best of the author knowledge, the only previous work dealing with Unix Command classification is the one proposed by Trizna et al. [356]. They introduce the Shell Language Pre-processing (SLP) library, to support tokenization and encoding of parsing Unix commands. Consequently, they train ML models to distinguish malicious command samples from benign activity, to show that the acquired metrics provide a significant improvement of the F1-score on the malicious classification task.

PyComm [359] is a malicious command detection model for Python scripts. It is based on random forest applied on a hybrid set of static features and Python source code strings. During evaluation, they obtained an accuracy of 95.5% with a recall of 94.3%.

Hussain et al. [353] present a similarity retrieval system to cluster groups of commands that are semantically equivalent. This can be used for classifying commands with unknown risk based on the cluster label. The similarity of commands is estimated by processing the documentation of commands using NLP techniques. Single-program commands are first parsed to recognize the program name and parameters. The program name is then used to retrieve relevant documentation text, to which Term Frequency-Inverse Document Frequency (TF-IDF) [113] is applied to construct a matrix representation for both the documentation and the command parameters. The similarity of two commands is then measured using Cosine Similarity. Classification by similarity is highly sensitive to the choice of the similarity function and to the availability of a large quantity of labeled commands. Moreover, this solution assumes single-program commands, which represent only a fraction of possible commands executed by real-world operators.

5.4 Command-line Risk Classification using Transformer-based Neural Architectures

In this section, two approaches for command risk classification based on Large Language Models (LLMs) are described [345]. A background on NLP and LLMs for command risk classification is provided on Section 5.4.1. The base approach, formalized by a transformer-based architecture, is described in Section 5.4.2. Section 5.4.3 describes the training procedure for the base approach. Section 5.4.4 describes how the classification system is evaluated, while Section 5.4.5 provides the evaluation results.

Section 5.4.6 proposes an advanced approach for command risk classification, based on command documentation. The system architecture is described and final classification results presented. Section 5.4.7 presents a list of cloud-related application for the proposed LLM framework.

5.4.1 Introduction

Several works in NLP have shown the high capabilities provided by ML algorithms, and specifically LLMs on a variety of NLP-related tasks [115]. Their double learning procedure, based on pretraining the model on a large-scale, domain-specific corpus of text, and fine-tuning the model for the specific-task with supervised data, allows to provide higher context understanding and generalization power than previous NLP algorithms.

Similar to other sentimental analysis tasks, LLMs can be applied on documentation text to evaluate the polarity in terms of dangerous/safe behavior, i.e., if the English documentation of a command thoroughly describes the behavior of a command, the LLM may be able to classify if such command can constitute a risk for the host system. A similar argument applies to the remote command itself which, as an instance of a programming language with predefined lemmas, syntax, context and structural dependencies, resembles in any form a natural language, such as English.

No previous work has applied LLMs for command classification, or it has tried to adapt existing LLM-powered solutions for text classification for the CLI security domain. To this end, the next section illustrates how to apply LLMs for the command-line language.

5.4.2 System Architecture

This section presents the system architecture of a LLM-based command risk classifier.

The risk classification system [345] is based on Bidirectional Encoders Representations from Transformers (BERT), originally proposed by Google [111] (see Section 2.3.3). BERT is a LLM for NLP and sequence-based ML applications [115]. BERT has shown to provide more effective results for discriminative tasks, compared to the effectiveness of other approaches (e.g., GPT3 [8]) for generative tasks. For this reason, BERT was chosen as the base LLM approach for the command risk classification task.

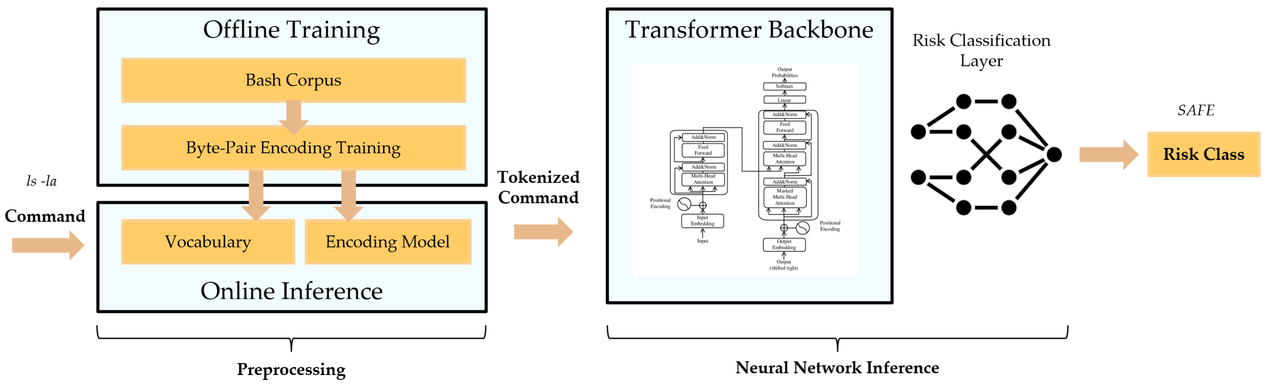


Figure 5.3 LLM Classifier architecture [345]. The input command is pre-processed via Byte-Pair Encoding (BPE) to construct an input sequence of tokens. The sequence is then processed by the BERT backbone to produce a latent representation of the command, which encodes important language-related information learned during pretraining. This latent representation is given to the risk classification layer to estimate the final command risk.

The system architecture is shown in Figure 5.3. It is composed of a pre-processing algorithm and a neural network architecture, which is trained as described in Section 5.4.3.

The only input to the system is the executed command (as a string). In the pre-processing step, the raw command string is split into a sequence of tokens using the BPE algorithm [360, 361]. BPE is an unsupervised tokenization method, in which the most frequently occurring pair of characters is recursively replaced with a character that does not occur in the vocabulary. This allows to estimate the more frequent combination of characters and use it as tokens for the language. BPE is preferred over traditional space-based tokenization for two main reasons [362]:

- BPE will identify the optimal set of tokens, i.e., the set of tokens that are more frequent in the observed language;
- The identified tokens will be composed of one or more characters, but they will be shorter than traditional words, which will make the model more robust towards out-of-vocabulary tokens. This increases the coverage provided by the tokenization system without resorting to a large fixed vocabulary;

The BPE-encoded tokens are fed into a bidirectional transformer architecture [7]. The token indices are mapped to embeddings and fed through the encoder-decoder architecture of the transformer. The transformer architecture is composed of n_A attention heads [7] of size H. To perform classification, an additional classification layer block, with output size C, is appended to the transformer output. The final output is a list of class probabilities associated to the different risk classes. The class with the highest associated probability is selected as the predicted class.

5.4.3 Training

The training procedure of the above-mentioned model [345] is composed of three steps: dataset collection, BPE training, BERT training (pretraining and BERT finetuning). Figure 5.4 summarizes the construction phases of the LLM classifier.

Dataset Collection

For the pretraining phase, requiring large-quantity of raw command data, a corpus of Bash files was programmatically collected from publicly available data. First, GitHub repositories with the Bash language tag were searched. Then, from each of the identified repositories Bash-related files were selected, by matching specific criteria (first line contains a shebang, or the file has a *.sh* extension). This resulted in a final collection of 71164 Bash scripts, amounting to about 500 MB.

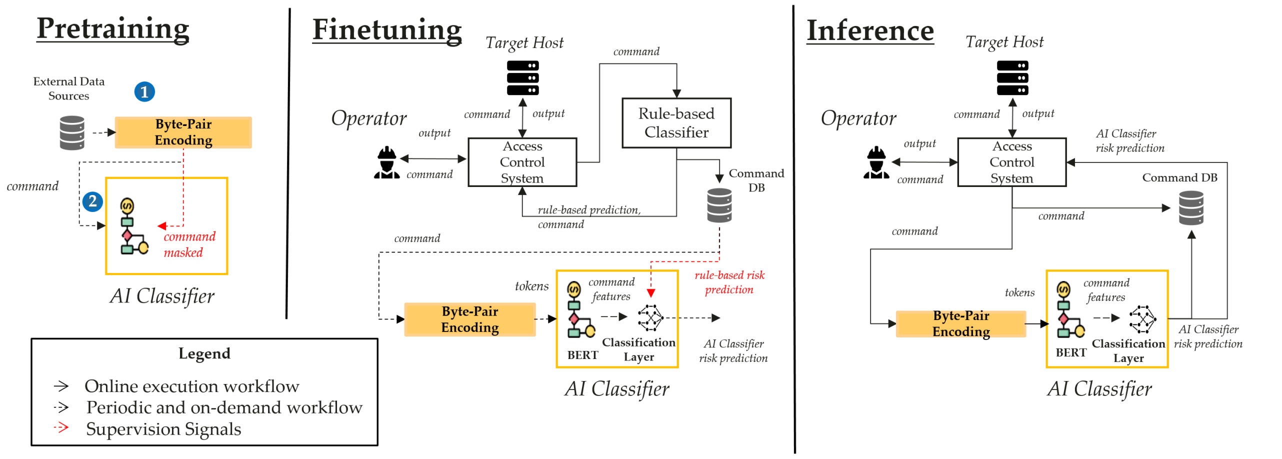


Figure 5.4 System architecture during the three phases of construction of the LLM classifier [345]. During pretraining, a command corpus is used to learn the language tokens and their context relationships. During finetuning, a dataset of labeled commands is used to specialize the AI model for the risk classification task. Both commands and labels are originating from the interception system composed of a rule-based classifier. During inference, the LLM classifier replaces the rule-based classifier providing online risk classification for all commands executed.

BPE Training

During this phase, the corpus of Bash commands is used to learn the tokens and patterns of the scripting language in use, using the BPE algorithm described above (Section 5.4.2). The frequency of different character-level patterns is estimated based on the observed data, to produce a vocabulary of V learned tokens and an encoding algorithm to extract them from raw command strings.

BERT Training

The BERT pretraining step requires to train the transformer network on self-supervised contextual tasks. Therefore, the command corpus is augmented to construct a self-supervised dataset for two contextual tasks, which require sets of adjacent commands as inputs. Because of the many possibilities of sampling subsets of commands from the corpus, after this augmentation step the total dataset size amounts to 15 GB.

During the pretraining phase, the transformer model is pretrained on the same contextual tasks described in the original BERT paper [111]: bidirectional Masked LM, by masking 15% of sequence tokens at random; and next sentence prediction, i.e., by predicting if a group of commands is adjacent to an observed sequence of commands, or is a randomly sampled command set. After pretraining, only the network backbone, which outputs, an h_i-long representation of the input command is retained, while the contextual output layers are dropped.

Finetuning

During the finetuning phase, the pretrained transformer backbone is extended with an additional classification layer block to enable command risk classification. The classification layer block is composed of a fully-connected layer preceded by a dropout layer and followed by a softmax normalization layer. For this training phase, a supervised dataset of commands, annotated with risk classes, is used. The network pipeline is trained end-to-end using the cross-entropy loss function, to maximize the log likelihood of the training dataset. The network weights are updated using gradient descent, with gradients computed via back-propagation.

5.4.4 Experimental Setup

In this section, the experimental setup used during the training experiments and for the evaluation of all prediction models is described.

	parameter	value	description
NN architecture	H	256	hidden size
	Gaussian Error Linear Unit (GELU)	activation function	
	p_d	0.1	dropout probability
	n_A	4	attention heads
	n_h	4	hidden layers
	h_i	1024	intermediate size
	V	20000	vocabulary size
	L	1024	max sequence length
NN training	σ_{init}	0.02	initializer range
	B	128	batch size
	E	16	epochs
	C	3	output classes
	ϵ	$3 \cdot 10^{-4}$	learning rate
ML training	c	10^2	LR inverse regularize factor
	S	50	Word2Vec H
	α	$0.05 \rightarrow 0.0007$	Word2Vec ϵ (start/min)
	E	100	Word2Vec E

Table 5.2 Hyper-parameter configuration used in the command risk classification experiments [345].

The proposed approach and all the evaluated models were implemented in Python. The *4/256 BERT mini* model was selected as the main architecture. All network layers utilize GELU [101] activation functions and are trained using the Adam optimizer. The full list of numerical hyperparameters is shown in Table 5.2. The transformer model was pretrained for 350,000 iterations (or 160 epochs). The total training time required 10 days on a single NVIDIA v100 GPU.

In addition to the pretraining corpus described in Section 5.4.3, a second dataset of commands, annotated with class risk, was collected and labeled. This dataset contains a realistic collection of commands used for O&M and is used for training and evaluating the classification model and all other models, according to the metrics presented in Section 2.3.5.

As the annotated commands originate from an internal cloud production systems, invalid samples resulting from errors in the interception system, such as password prompts, non-Bash commands, and bash terminal output have been automatically filtered out and then manually verified. Misspelled and unknown commands have been preserved, as they still represent executable commands and may cause damage. Multi-program commands (resulting from pipelining, `xargs`, ...) and script file calls have also been preserved, as they are representative of real commands executed in production.

Both commands and corresponding risk labels are retrieved from an internal data store, used for offline analysis of risk predictions as described in Section 5.4.7 and as depicted in Figure 5.4. The risk labels originate from a rule-based system, and have been manually verified by experts and corrected in case of discrepancies with true command risk. In the current scenario, three risk classes are considered:

- SAFE, for read-only commands and commands that do not alter the state of the system significantly;
- RISKY, for commands that may irreversibly alter the state of the system and cause damage, for which privilege escalation is required;
- BLOCKED, for commands that will irreversibly alter the correct state of the system, and must never be executed.

To ensure the annotated dataset is representative of a real command distribution, the distribution of commands in the three classes was studied. After removing invalid commands, an approximate 80%, 20% split between SAFE and RISKY commands was observed, with an additional 0.3% component of (extremely rare) BLOCKED commands.

Therefore, after expert verification of the risk labels, the annotated dataset was down-sampled to replicate the class ratios observed in the class analysis. In the end, a total 47158 high-quality commands, representative of a real O&M workload, are collected. This annotated dataset is divided into *train*, *dev*, and *test* splits with

Dataset Split	SAFE	RISKY	BLOCKED	Total
train	26905	6014	91	33010 (70%)
dev	7637	1765	30	9432 (20%)
test	3858	850	8	4716 (10%)
Total	**38400 (81.43%)**	**8629 (18.30%)**	**129 (0.27%)**	**47158** (100%)

Table 5.3 Composition of the command risk dataset [345], used for finetuning the BERT model and evaluating all models tested.

70%, 20%, 10% ratios, while preserving the class distribution. Table 5.3 summarizes the annotated dataset composition.

5.4.5 Results

The risk classification model was evaluated in terms of accuracy, precision, recall, and F1-score as described in Section 2.3.5. These metrics are measured for the two positive classes of the command dataset, as the main focus is on the detection of dangerous commands, so the SAFE class metrics are not evaluated. The described approach is compared to:

- a baseline model, constructed using Word2Vec embeddings [114] + Random Forest (RF) algorithm;
- the NLP and neural-based approaches presented in [352] for malicious command detection, namely a 3-gram model, a BoW model, two one-dimensional CNNs of 4 and 9 layers (called *4-cnn* and *9-cnn*, respectively), a LSTM-based neural network, and the ensemble model combining the *3-gram* and *4-cnn* models.

The implementations of these algorithms is reproduced to the best of the knowledge available, re-adopting the original hyperparameters when provided.

To evaluate the real-world applicability of the command risk predictors, execution times of all evaluated algorithms were measured in a simulated Infrastructure-as-a-Service (IaaS) cloud deployment with dedicated hardware. Table 5.4 presents the average training and inference time for the different algorithms evaluated. The traditional NLP (3-gram and Word2Vec) models are faster in both training and inference time, with sub-millisecond inference time per sample. They, however, pay a price in terms of accuracy, as illustrated below. Among the deep learning methods, the LSTM shows the best training and inference runtime (3 and 0.9 ms per

	Model	Training		Inference	
		Total	Per sample	Total	Per Sample
	Word2Vec+RF	15.00	0.0004	0.08	$< 10^{-4}$
[352]	3-gram	10.64	0.0003	0.18	$< 10^{-4}$
[352]	BoW	1.399	$< 10^{-4}$	0.08	$< 10^{-4}$
[352]	4-cnn	11281.93	0.0214	101.23	0.0215
[352]	9-cnn	11402.34	0.0216	104.01	0.0221
[352]	LSTM	1581.05	0.0030	4.16	0.0009
[352]	DTEnsemble	11292.50	0.3421	110.02	0.0233
	Bash BERT	3766.33	0.1141	186.43	0.0395

Table 5.4 Runtime performance of the evaluated algorithms [345] in terms of training and test execution time (measured in seconds).

	Model	Precision			Recall			F1-score		
		RISKY	BLOCKED	R+B	RISKY	BLOCKED	R+B	RISKY	BLOCKED	R+B
	Word2Vec	0.9210	1.0000	0.9440	0.6658	0.2857	0.6638	0.7729	0.4444	0.7795
	3-gram	0.9322	0.8571	0.9328	0.8576	0.7500	0.8578	0.8934	0.8000	0.8937
	BoW	0.7018	0.3212	0.7048	0.3212	0.3750	0.3228	0.4407	0.5000	0.4428
[352]	4-CNN	0.9391	**1.0000**	0.9420	0.8894	0.6250	0.8893	0.9136	0.7692	0.9149
	9-CNN	0.9420	0.3077	0.9430	0.8212	0.5000	0.8287	0.8774	0.3810	0.8821
	LSTM	0.9451	–	0.9507	0.7894	0.0000	0.7867	0.8603	–	0.8610
	DTEnsemble	0.9600	0.8333	0.9603	0.9035	0.6250	0.9021	0.9309	0.7143	0.9303
	Bash BERT	***0.9713***	***1.0000***	***0.9716***	***0.9165***	***0.8750***	***0.9161***	***0.9431***	***0.9333***	***0.9430***

Table 5.5 Evaluation Results in terms of precision, recall and F1-score on the positive classes RISKY and BLOCKED, for different algorithms [345]. (R+B) = all dangerous commands (RISKY + BLOCKED). Bold value indicates the best model for each metric. "-" indicates the metric cannot be computed due to no True Positive (TP) predictions.

sample, respectively). The remaining models, including the proposed BERT-based approach, require longer training (in the order of 21 to 114 ms per training set sample).

All models, however, acceptable inference time for real-time production evaluation, ranging up to 39 ms, in the case of the BERT model. This value is considerable acceptable in relation to the other factors playing towards total command evaluation time, which must also include the round-trip latency from/to the SSH endpoint and to/from the target host (for a total of a few hundreds milliseconds), and the average time distance between commands executed in production (around one second).

Table 5.5 presents the comparison of the different models under evaluation, in terms of precision, recall, and F1-score on the two positive classes RISKY and BLOCKED. The proposed BERT-based approach achieves the highest absolute score for 8 of 9 metrics measured. Over the second-best result, the precision in detecting RISKY commands is improved by 1.13%, while the recall is increased by 1.30%. For BLOCKED commands, an increase of 16.7% in precision and 25.0% recall was measured. On average, for all dangerous commands, the LLM approach can improve precision by 1.13% and recall by 1.40%. The increase in F1-score for all dangerous commands is 1.27%. By considering the ratio of dangerous commands (20% from Table 5.3) and by assuming an average number of 3M commands executed per month, which is consistent with a realistic production workload, such increase in recall results in approximately 60k additional dangerous commands intercepted during operations, which may cause an equivalent number of potential incidents.

To show the transfer learning abilities of the proposed model, the accuracy of predictors in recognizing dangerous commands with limited training data was also evaluated. Subsets of 100, 200, 500, 1000, . . . , 20000 commands were randomly sampled from the training set to train the same NLP models. Results are reported in Figure 5.5. It can be observed how the BERT model achieves the best F1 performance on dangerous commands for all reduced dataset sizes, while requiring one order of magnitude less samples to achieve comparable F1-score results. This demonstrates that LLM retains and uses knowledge from the pretraining step to recognize dangerous commands more effectively.

5.4.6 Documentation-based Command Risk Classification

Potential Data Sources

In Section 5.4.2, a NLP classification approach based on LLM was proposed. The prediction of the classifier described uses the command text as input, and its prediction is directly related to the content of the command string, which must encode an indication of risk.

The accuracy of this 'direct' command risk classifier is highly dependent of the quality of the command dataset provided for finetuning. The quality of the command dataset (composed of commands and risk labels) is based on:

- the quality of the commands alone which, as a subset, may or may not be representative of the entirety of operations executed in the real cloud system. Even in the presence of a comprehensive dataset,

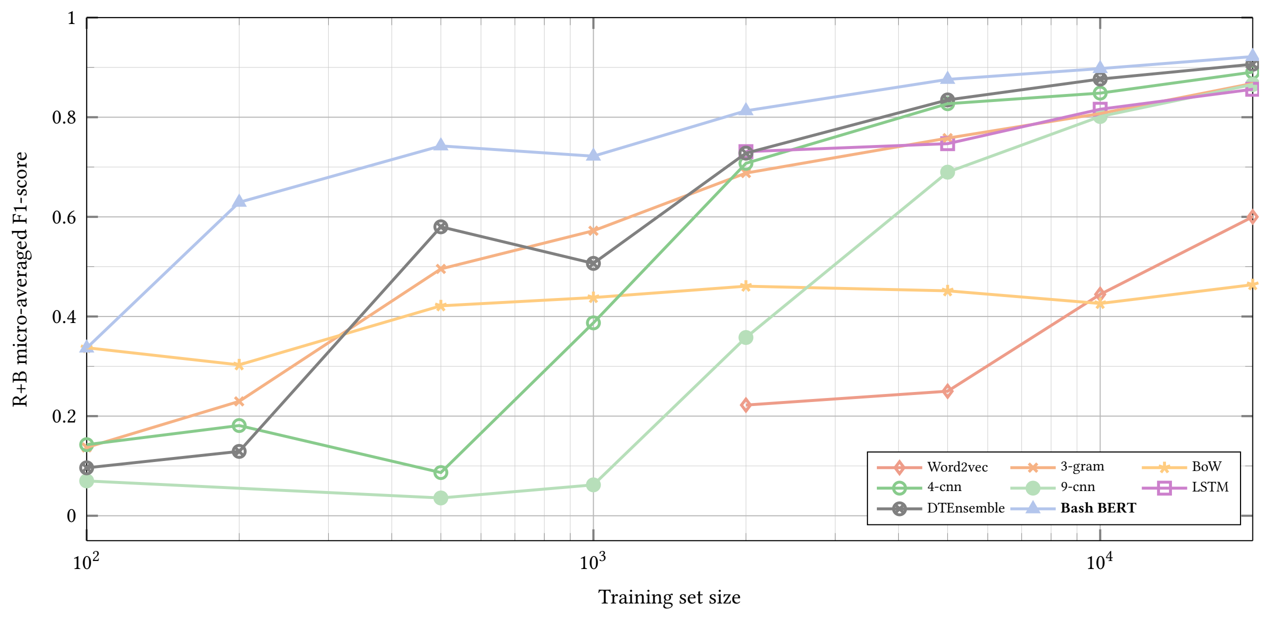


Figure 5.5 F1-score of RISKY and BLOCKED commands on test set for different evaluated algorithms [345], as a function of dataset size used for training. The BERT approach can classify dangerous commands more accurately in the presence of limited training data. Missing points indicate the F1-score could not be computed due to no TP predictions.

the commands must be able to carry sufficient information to represent the risk of executing them. Command strings may be too implicit, but may also be incorrectly extracted, poorly-formatted, or not containing a command at all, when the strings are automatically scraped from networking interception systems. They may also include commands which represent syntax of different CLI tools, such as DB frameworks (MySQL, MongoDB, InfluxDB, etc.) or interactive commands (e.g. Python);

- the quality of the labels assigned to commands. O&M personnel or other field experts are responsible for assigning risk classes to potentially long list of commands, in order to finetune the model for risk classification. This may easily cause labeling error, but it may also lead to miscalculate the risk deliberately, based on the personal experience of the labeler, rather than *inherent* risk of the command.

The first issue may be addressed by improving the collection and revision process of commands. AI-based tools may also be employed to recognize which strings are effective, valid commands and remove garbage.

The second issue cannot be addressed without understanding to true definition of risk and how to obtain it. As long as operators are free to label commands at will, the evaluation of risk is subjective and cannot be fully relied upon.

Without relying on existing classification rules and operator expertise, the source of risk knowledge must then be provided externally. Several data sources can be considered for estimating the risk of commands:

- *OS-level instrumentation.* OS internal tools, such as `strace` or `kprobes` for Linux, or `drstrace` on Windows, allow to collect low-level information about the system calls invoked by the instrumented programs. Audit logs may also be configured to register when specific functions are called. Some Kubernetes config files (*seccomp*) also provide information about allowed system calls inside containerized programs;

- *source code.* Similarly to OS-level instrumentation, via static analysis of source code it is possible to analyze which system calls are invoked and which resources are used during the execution of a command;

- *vulnerability registries and knowledge bases.* They are typically maintained and employed by security programs, such as antivirus software, to store definitions of executables and files which have been associated with cyber-attacks;

- *command documentation.* It provides a description of the syntax, parameters and options of a CLI program. Documentation pages also contain a high-level description of the main uses (and consequences) of using the program. Numerous standards for documentation exist, the most important and comprehensive under Linux is *manpages* [363], which is a collection of program-oriented documentation pages stored in the host system and retrievable through the `man` command.

OS-level instrumentation provides very granular information about the call stack of individual programs. System calls are usually associated to privileged operations and a correlation of risk can reveal potentially damaging operations. However, OS-level instrumentation requires commands to be executed in order to collect tracing information. Moreover, these methods impose an overhead on the target system, i.e., a production host which may host customer workload, that is typically not acceptable for failure prevention purposes. Finally, OS-level instrumentation can only monitor operations run locally on the instrumented host, but cannot produce information if some internal operation is performed remotely (e.g., via API calls or for orchestration commands such as Openstack).

Source code has several drawbacks, as it is available only for open-source software and a sound static analysis of system calls is complex, due to code branching and multiple-language interfacing.

Vulnerability registries are mostly restricted to the cybersecurity domain and do not take into account commands that are dangerous due to human errors. These knowledge bases are also kept confidential for commercial reasons.

Documentation, on the other hand, is freely available (especially under Linux systems) and for undocumented programs it can be easily obtained from the Internet or directly produced. It also contains detailed description of program behavior and usage, in a format that allows to distinguish the outcome based on arguments, parameters, and flags. Documentation also has the advantage of being based on natural language, which makes it suitable to the NLP-based tools used so far for CLI security.

For the above-mentioned reasons, documentation is selected as the additional knowledge source for command risk classification.

Architecture

The documentation-based approach [364] is composed of two main building blocks: the documentation system, and the double-head LLM classifier. The system also features an optional, similarity search system to re-use existing knowledge.

Figure 5.6 depicts the complete system architecture. The documentation system is used to produce short, personalized summaries (or descriptions) of executed commands. The LLM classifier takes as input both the command summary as well as the raw command to estimate the command risk.

At training time, the system relies on the use of expert labels (either provided directly, or retrieved from an existing ruled-based system as in Figure 5.4.6). At inference time, it provides an output risk classification to the access control system, located on the bastion host.

The next sections describe more in detail how to effectively utilize documentation pages for assessing the risk of terminal commands.

Documentation System

The documentation system is responsible to store, manage and process documentation pages [364]. It is composed of a documentation database and a command description module. The documentation system is depicted in Figure 5.7.

The documentation database [364] is a document-oriented database where documentation files of terminal-executable programs are stored in a structured format. A single documentation file provides details on how to use a single program in detail, including syntax usage, program behavior, and available options. The documentation is organized in paragraphs, so that each paragraph either describes the general usage or specific option and/or argument. The documentation is imported in a preliminary setup step using an import script, which takes standardized documentation files as input. Documentation that does not adhere to the standardized format required by the import script must be converted. Internally-defined commands can also be

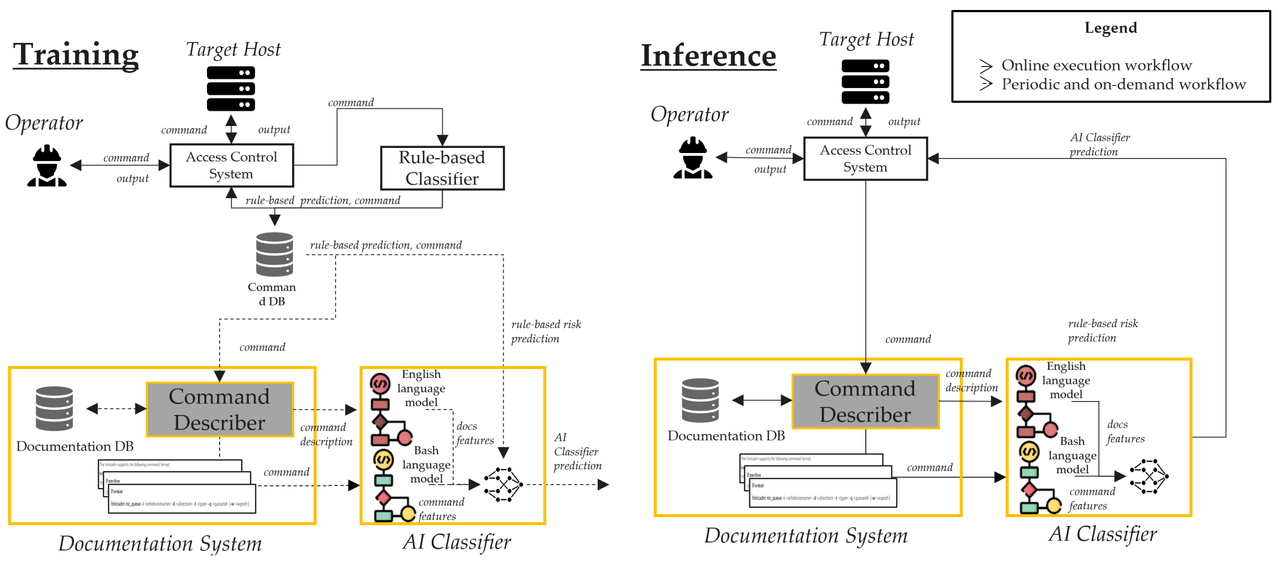


Figure 5.6 Training (left) and inference (right) architecture of the documentation-based approach for command risk classification [364]. During training, the double-head AI model learns from the predictions of the rule-based model stored in a command database. During inference, the AI model replaces the rule-based system for online risk assessment of commands.

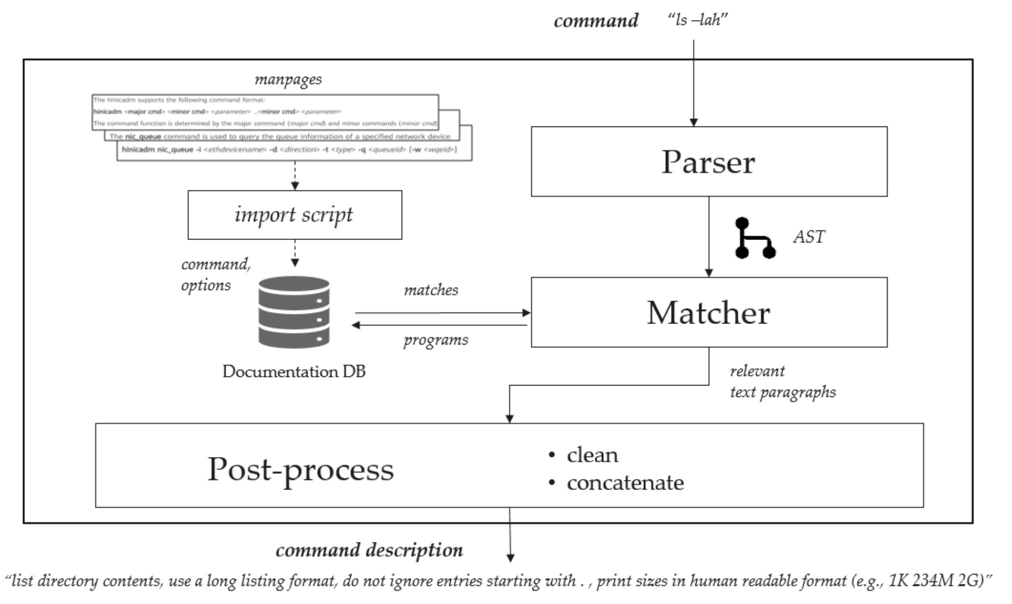
included to improve auditing quality, although they are usually not documented out-of-the-box. In such case, they can be documented directly in the standardized format. The different data sources employed can be summarized as follows:

- documentation pages describing standard operating system commands, files, and system calls;
- internally-defined scripts, programs and aliases;
- additional third-party programs and tools frequently used in the IT environment;
- additional third-party or internal filenames frequently used in the IT environment.

The command description module [364] is a software module that produces explanation summaries of terminal commands. Given an incoming command, the command description module retrieves the relevant documentation pages from the documentation database, extracts information relevant to current command instance, and produces a short text describing the command functionality in English language. It is composed of three submodules:

- a *parser* program, which produces an AST representation of the incoming command, where command tokens are tree nodes with program, option, or argument tags;
- a *matcher* program, which associates elements in the AST with documents in the documentation database, by matching AST element names with documentation file titles and aliases. After the relevant documents are identified, for each document relevant paragraphs are extracted, by matching options and arguments of the AST with the paragraphs of the documentation. General description paragraphs for the program (synopsis, usage, main functionality) are always included in the final selection;
- a *post-processing* program, which concatenates the selected paragraphs and cleans the obtained text, by removing HTML tags, links, author and copyright information; by removing trailing spaces and normalizing text.

The final output is a single string describing the behavior of the input command. The command description is specific to the set of programs, options and arguments specified. The description is sent to the AI model as additional source of information, in addition to the raw command string.



```
The hinicadm supports the following command format:
hinicadm <major cmd> <minor cmd> <parameter> ...<minor cmd> <parameter>
The command function is determined by the major command (major cmd) and minor commands (minor cmd)
The nic_queue command is used to query the queue information of a specified network device.
hinicadm nic_queue -i <ethdevicename> -d <direction> -t <type> -q <queueid> [-w <wqeid>]
```

Figure 5.7 Diagram of the documentation system [364]. Documentation pages are parsed and imported into a documentation database (on the left). At runtime, commands are parsed to an Abstract Syntax Tree (AST) and divided into fragments, each corresponding to a CLI program ('Matcher', on the right). Corresponding program pages are retrieved from the documentation database and only the relevant content of the page, in terms of options and arguments used, is extracted. The result is post-processed and given and output to the LLM classifier.

Double-head AI Classifier

The double-head LLM classifier [364] replaces the previous Bash-trained BERT model for classification. It is composed of:

- A BERT head pretrained on English corpus (English BERT head), for processing the documentation text produced by the documentation system. This head is used in combination with a full tokenizer, combining base tokenization (based on spacing and punctuation) with WordPiece tokenization [365]. The full tokenizer allows to efficiently encode the English text to a latent vector representation.
- A BERT head pretrained on a corpus of Bash commands (Bash BERT head), for processing the command string (as in the base classifier). As before, this head is used in combination with a BPE tokenizer, which can adapt the set of encoding tokens to the input provided during pretraining and to the desired vocabulary size.

The final LLM prediction is computed as:

- the prediction of the double Bert classifier (Bash+Docs), if the documentation is available;
- the prediction of the single Bert classifier (Bash), if the documentation is not available.

Similarity Search System

A similarity search system [364] is devised to overcome potential limitations in the documentation knowledge base, by exploiting the existing system rules to generalize risk predictability in proximal cases. This system covers potential shortcomings such as missing documentation pages for a command, inability to associate a command to its respective pages (e.g., `/home/user/.venv/bin/python3.6` cannot be associated to Python), or command name aliasing. The diagram of the similarity search system is shown in Figure 5.8.

The similarity search system implements a kNN algorithm to classify incoming commands based on the rules stored in a rule database (see Figure 5.8). Each rule is associated with a blocking/allowing policy, which can be used as training label for the classification algorithm. During training, database rules are retrieved and

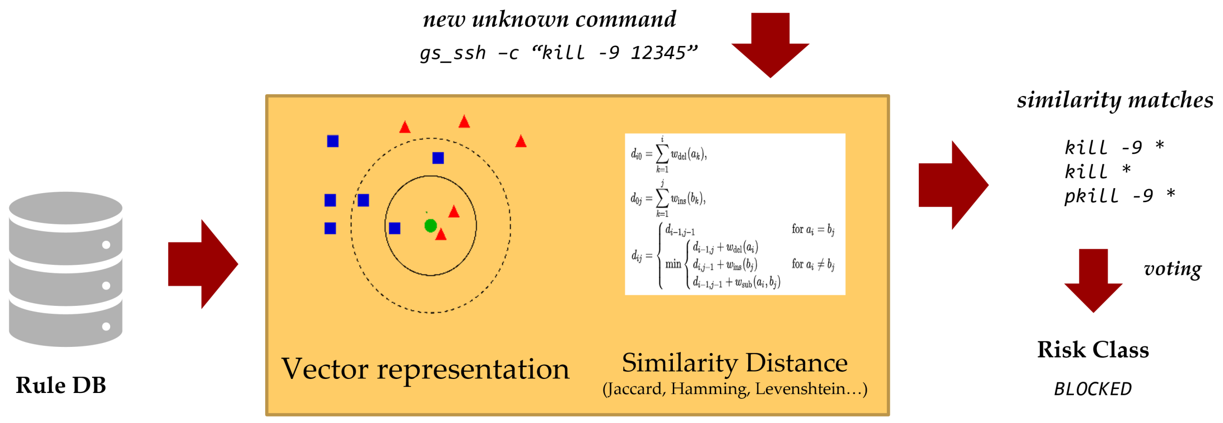


Figure 5.8 Diagram of the similarity search system [364]. The set of rules stored in the rule database is used to retrieve similar commands using a similarity function in a vector space. These similar items are then used to classify the command using a k-Nearest Neighbors (kNN) algorithm.

an efficient tree-like (k-d tree) data structure is fitted for enabling fast neighbor queries during the inference phase. During inference, the incoming command is used as query to return the most similar matching rules. To evaluate similarity, the Levenshtein distance is used, to measure the minimum number of insertion, deletion, or replacement changes necessary to convert the command into a potential match. Once the top k matches are retrieved, their associated labels are used to predict the class of the incoming command, by majority voting.

Response Aggregation

The similarity search model can be enabled or disabled on demand. If it is disabled, the final prediction is the LLM prediction. Otherwise, the final risk prediction incorporates both the LLM prediction and the similarity search prediction.

The aggregation of LLM prediction and similarity search is computed as:

- The similarity search prediction, if the LLM prediction is SAFE and the similarity matching prediction differs;
- The LLM prediction, in all other cases.

This choice is made to emphasize discovery of high-risk commands (i.e., increase recall).

Experimental Setup and Results

To train and evaluate the documentation-based approach, the three necessary data sources (documentation, commands, and rules) were collected as follows:

- *manpages* from the entire Ubuntu repository [363] were collected and imported in the documentation system;
- Documentation for several non-UNIX commands (e.g., Openstack [57]) was collected from publicly available Internet resources and converted to the *manpages* format;
- Several internal-use and other third-party commands were self-documented.

The total number of documentation pages imported is 36.4k, each corresponding to an executable program or file definition. For the commands, the same dataset utilized for the direct command classification approach was used. For the similarity search approach, a set of 5049 production rules, annotated with the three risk classes above described, was imported.

The Bash AI head was pretrained according to the same methodology described in Section 5.4.3. The English BERT head utilizes the same BERT architecture and hyperparameters; however, it was not pretrained since

Model	Precision			Recall			F1-score		
	RISKY	BLOCKED	R+B	RISKY	BLOCKED	R+B	RISKY	BLOCKED	R+B
Bash BERT	*0.9713*	*1.0000*	*0.9716*	*0.9165*	*0.8750*	*0.9161*	*0.9431*	*0.9333*	*0.9430*
Bash+Docs BERT	*0.9712*	*0.8750*	*0.9703*	*0.9141*	*0.8750*	*0.9138*	*0.9418*	*0.8750*	*0.9806*

Table 5.6 Comparison of proposed 'direct' (Bash BERT) and documentation-based (Bash+Docs BERT) classification models in terms of precision, recall and F1-score on the positive classes RISKY and BLOCKED. (R+B) = all dangerous commands (RISKY + BLOCKED).

Scenario Name	Rule-based Prediction	Documentation-based LLM prediction	# Commands	Precision (%)	Recall (%)
whitelisting	RISKY	SAFE	14	64.3	100.0
blacklisting	SAFE	RISKY/BLOCKED	21	9.5	100.0

Table 5.7 Results of the auditing evaluation in the two scenarios described (whitelisting, blacklisting).

the publicly available model [366] is already pretrained on English language corpora. The final finetuning were also carried out according to the same methodology of the direct classification approach.

The risk classification ability of the documentation-based classifier was evaluated in two steps: accuracy evaluation and auditing evaluation.

The accuracy evaluation measured the correctness of predictions directly. Table 5.6 reports the results. The documentation-based approach reaches classification performance similar to the direct classification approach. An analysis of dataset commands reveals that the test set used for evaluation (although composed of commands not seen during training) contains commands that are quite similar to the commands used for training. Both training and test set are in fact taken from the risk predictions performed by the rule-based system. Based on these results, both the direct and documentation-based models are able to classify these unseen commands equally well, as they are fairly similar to the training set commands.

It is however expected that the documentation-based approach must be able to generalize better to new commands outside the training set distribution. To this end, the auditing evaluation is performed. Auditing (see also Sections 5.3.3 and 5.4.7) allows to discover new dangerous and safe commands previously misclassified, in order to improve the classification accuracy of an existing system. As the auditing procedure is only triggered when an unknown command must be classified, it allows to evaluate the generalization ability of the algorithm beyond the commands used for training. For this part of the evaluation, over 500 commands were manually labeled and thoroughly verified by experts. Because commands that are either misclassified or not classified by the rule-based system are in general rare to find, the evaluation is upper-bound limited by the total number of unlabeled and misclassified commands observed (35).

Table 5.7 reports the results of the auditing evaluation. Two main scenarios are considered:

1. *whitelisting*. The audited (rule-based) system does not have an associated rule (defaults to RISKY), the documentation-based system predicts SAFE ;

2. *blacklisting*. The audited system predicts SAFE, the documentation system predicts RISKY or BLOCKED.

For the whitelisting case, the documentation-based AI model is able to detect all SAFE commands with acceptable precision (64.3%). From a real-world use perspective, its use requires human supervision, but it was evaluated overall useful to find commands to whitelist by O&M experts.

For the blacklisting case, the documentation-based AI model is again able to detect all potentially RISKY commands, although it tends to over-report with a consequent drop in precision (9.5%). This entails that the documentation-based approach is able to suggest commands to blacklist, although it requires a necessary revision step by humans as well.

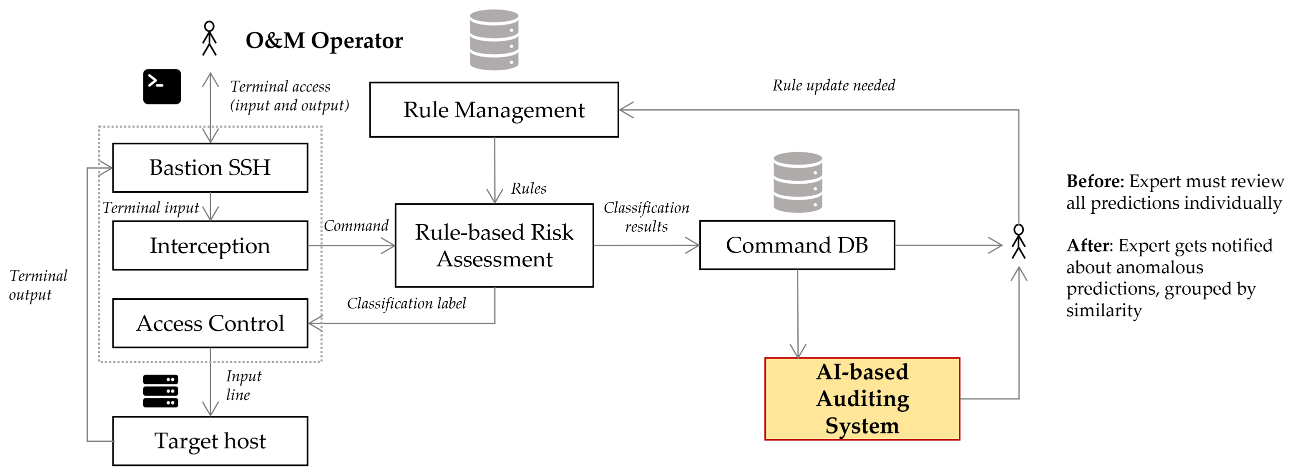


Figure 5.9 Example diagram for auditing applications [345]. A rule-based classifier ('Risk Assessment') evaluates the risk of incoming commands (captured by the interception system in the let-side dashed box) through a knowledge base of rules ('Rule Management'). An additional AI-based system (gray box at the bottom) evaluates the predictions of the rule-based system to recommend corrections to human operators (right side of the picture).

5.4.7 Use Cases of LLM models for the Command Language

The BERT models described above can be applied in many industrial applications related to the command-line, which is the entrance door to the platform layer of the cloud. In this section, the applicability of the approach into different contexts is presented. Several use cases for the model are described, which have been used or are planned to be used in internal cloud production systems in the future.

The first use case is online risk classification. Command risk can be evaluated online to block dangerous commands during interception. A scheme of online risk classification is shown in the last segment of Figure 5.4. First, commands executed over remote terminal are intercepted by an access control system (e.g., a bastion host). Then, the risk of the intercepted command is evaluated using the LLM-based classifier. The classifier can be deployed directly on-site, or accessed via an inference API to take advantage of specialized hardware. Commands and classifier predictions can be stored to permanent storage for offline analysis. Based on the prediction outcome, commands are either blocked or allowed, and the operation output displayed to the operator.

System auditing is the practice of analyzing the quality of an existing system to validate its results and consider potential improvements. In the context of command interception systems, an existing risk classifier can be audited by analyzing if its risk predictions correspond to the true risks of the commands, to support the identification of errors and the creation of new classification rules.

The language model was applied for auditing an existing rule-based system. An example of an auditing pipeline is shown in Figure 5.9. The existing system is composed of a rule-based risk classifier, a rule management system, and database store of rule-based and AI-based predictions. When revision of existing rules is needed, a report of non-matching of predictions from the rule-based and AI models is generated. As for the majority of commands the two predictions will correspond, it is sufficient to report only the commands where the two predictions differ. An human expert can then decide if the discrepancy reported by the AI model is correct and update the corresponding rule in the management system. This comparison speeds up the work of expert reviewers, as they only need a small portion of commands, where the two predictions do not match. Thanks to the high precision of the command classifier, if a command is reported as dangerous, it is likely that an rule update action must be taken. Using the language model, it was possible to discover several new risky commands that were not detected by the rule-based system.

The language model was also considered for command categorization. Commands can be assigned to predefined categories based on their function (e.g., networking, filesystem, scripting, third-party command). For O&M operators it is convenient to know the category of a command for several reasons: to identify similar commands and construct new rules, to clarify the meaning of unknown commands and speed up auditing, to understand which type of commands are currently not identified by the security system. The command categorization problem is comparable to command risk classification, as both can be achieved using the techniques described in this work (provided ground-truth labels are available).

In the conducted experiments, categories from manpages documents [363] were considered as a potential source of command categorization labels. First, commands are parsed to extract program names. Then, program names are matched to manpage files, which store corresponding command categories (e.g., `ls ='OS command'`). This allows to create a labeled dataset associating commands with one (or more) command categories. By applying the finetuning technique described in Section 5.4.3, it is possible to construct a language-specific command classifier for an arbitrary command taxonomy.

The language model approach may also be applied for other NLP-related tasks, such as context extraction, where it can be used to automatically extract code from mixed-language text, e.g., code tutorials and Jupyter Notebooks, and from Standard Operating Procedure (SOP) forms for O&M practices. The extraction of commands from SOPs allows to verify the correct execution of the procedure and potentially recognize SOPs and command sequences that do not conform to the correct security standards.

The approach may also be used for named-entity recognition tasks in a command-line context, e.g., to recognize filenames, API endpoints, or IP addresses. This task can support security auditing by identifying named entities that should not be accessed.

In conclusion, the LLM model may be used for traditional code-language applications, such as code generation and auto-completion. These tasks support development and configuration of software systems in the platform environment. The LLM model may also be used to classify portions of code or scripts to detect potential vulnerabilities and defects (similarly to Software Defect Prediction (SDP), Section 3.4.3).

5.5 Summary

In this chapter, the challenges related to platform-level FM were presented. The platform layer of the cloud provides the operating environment for building and executing software applications.

Platform layer tools are predominantly built, maintained and executed using the command line. Therefore, the security and reliability of the CLI is deemed of high importance to prevent platform-level failures. To this end, a LLM for the command-line language, which is applicable to several NLP-related tasks, was proposed and evaluated via comparison to existing NLP solutions.

First, it was shown how to apply the LLM approach for command risk classification. The language model leverages the contextual knowledge learned during pretraining to achieve higher classification accuracy and pinpoint dangerous CLI operations effectively. The procedure to train the model according to a realistic distribution of production commands was described.

Then, the accuracy of the approach was evaluated and compared to several existing approaches for command risk classification, in terms of performance and accuracy. The results show that an LLM-based solution is feasible for adoption in real-world cloud environments and can improve detection for rare classes of commands, reducing impact and occurrence of CLI-related incidents significantly.

Furthermore, the base LLM approach was extended to process command documentation pages, in order to include external knowledge about the behavior of commands. The documentation-based approach is shown to support efficient auditing of misclassified or unknown commands, for both whitelisting and blacklisting of commands.

In conclusion, the applicability of the LLM models to other CLI-related tasks, including command categorization, SOP verification, and other NLP tasks, was showcased.

5.5.1 Advantages of the Proposed Solution

The LLM approach focuses on universal applicability by supporting all valid commands, including concatenation of multiple commands via pipes and other possibilities allowed by the command-line syntax (`time, `CMD`, $, & &, xargs,`,...).

The LLM model can leverage information learned during pretraining for classification and thus only requires a limited dataset for finetuning to the context-specific tasks, providing more flexibility at a reduced effort. The documentation model, moreover, leverages previous knowledge coming from existing classification rules and external sources of knowledge coming from command documentation.

When applied for auditing [352, 353], the proposed approach operates for both discovery of FPs and FNs predictions during the auditing process, via comparison of all predictions for all samples, not only the positive predictions or for specific rules [355].

The documentation-based approach integrates command documentation to evaluate the inherent risk of commands, so that an existing classification rule can be challenged and revised. Moreover, a fallback solution based on the most similar rules is introduced, so that a risk classification from the auditing system is always available.

5.5.2 Potential Limitations of the Proposed Approach

As the approach is experimental, it is based on several assumptions in order to work effectively.

A first limitation of the approach is the surviving requirement for supervised data. However, the presence of the pretrained BERT backbone highly reduces the quantity of labeled commands needed. IT practitioners interested in this approach may have already implemented a command classification solution, which can aid in the collection and labeling of commands.

A second limitation is the input length. The model described in the paper accepts commands up to token length 512. Although this allows to classify the large majority of the valid commands in production, it may not be applicable to classify longer commands, such as one-liner Bash scripts.

Moreover, the current approach can only evaluate single command instances and cannot, therefore, evaluate the risk associated to a sequence of commands. It is however plausible that such extension can be achieved in the future, due to the flexibility of the model to adapt to different tasks.

Future work may investigate how to overcome such limitations and possibly extend further the applicability of this language model to other O&M tasks.

6 Software-level Proactive Failure Management

In this chapter, proactive failure management at the software level of the cloud stack is investigated.

In Section 6.1, software-level Failure Management (FM) is introduced, by describing the operating environment, common failure modes, and techniques to deal with software-level failures. In Section 6.2, the problem of Root Cause Analysis (RCA) in large-scale services, with its challenges and motivation, is introduced. Section 6.3 presents related work for software-level FM and RCA. Section 6.4 presents a pattern-mining-based algorithm for RCA on structured log data, with evaluation results and an extension to sequential logs. In Section 6.5, the applicability of the proposed approach in the context of large-scale cloud environments are discussed. The chapter is concluded with Section 6.6.

6.1 Introduction

6.1.1 The Software Layer

The software layer is responsible for delivering software applications as a product to end customers. It must ensure application services are executed effectively and data is consolidated.

Because the software layer represents the uppermost layer of the cloud stack model (Section 2.1.3), it is also the layer which provides most complex cloud offerings as a service. The typical Software-as-a-Service (SaaS) offering provides resources ranging the entire stack, from the physical hardware up to the load balancer. The software layer is also the only offering which effectively incorporates the software logic in the provided service. This has the consequence of introducing a large quantity of source-code related failures in the context of failure management.

Software bugs are a particular concern, due to the high variability and complexity of different software offerings [40]. Moreover, under significant load new instances of bugs and undesired behavior tend to appear. Around 40% of all cloud incidents are due to code-related bugs [367], in particular due to unverified code changes or buggy features, invalid values of flags and constants, invalid code dependencies and exception handling. The difficulty in re-creating the real-world production environment for experimental and debugging purposes also complicates traceability and study of software errors.

From an Operations & Maintenance (O&M) perspective, monitoring such a large and complex software stack is complex and time-consuming. While failures in the lower parts of the stack are treated according to the techniques described in the previous chapters.

6.1.2 Techniques for Software-level Proactive Failure Management

Based on the result of the systematic study earlier summarized [75] and additional literature review conducted, AI for IT Operations (AIOps) techniques for software-level proactive FM may be divided into five main categories:

- *software-level failure prevention.* These include numerous techniques already mentioned, relevant when applied at the software level, such as Software Defect Prediction (SDP) [29,199–211], i.e., the automated discovery of software bugs via source code analysis; checkpointing [216–218] and software rejuvenation [133, 137, 194, 214, 215], i.e, techniques to maintain clean operational states of software systems; and fault injection [212, 213], i.e., the study of how simulated injected faults affect the runtime behavior of software, to improve its design or configurations. AI models can be applied estimate the correct checkpointing/rejuvenation policies, or estimate probability of software bugs in source code;

- *software-level Online Failure Prediction (OFP).* It allows to predict whether specific software instances will fail or not. It can refer to predicting failures of software systems [196, 235, 237], e.g., predicting if a specific service is about to experience an error, or it can refer to self-contained instances of software workload [22, 350, 351], such as a job or a task (so-called *job failure prediction*). It relies on service Key Performance Indicator (KPI) or workload data, as well as on the output of OFP of lower levels;
- *deployment verification.* It allows to verify whether specific releases, software updates and rollbacks are safe to deploy for production use [368, 369]. AI models can predict the impact of software updates by analyzing historical update data and the current software environment. The provided insights help in proactively testing updates in isolated environments before deployment. In case an update triggers compatibility issues or failures, AI can also be used to initiate automated rollbacks to a stable version, ensuring uninterrupted service;
- *proactive load testing.* AI-driven load testing tools can simulate various usage scenarios to assess software performance under different conditions. The AI model learns to estimate the real-world conditions of operating a software in scenarios otherwise impractical or unsafe to test directly (without canary deployments). By proactively identifying bottlenecks and stress points, O&M engineers can optimize software performance and responsiveness. This ensures that the software remains stable even during periods of high demand, minimizing the risk of failures caused by unexpected load spikes;
- *runtime verification.* These techniques allow to verify the correct runtime behavior of software, to detect or predict future failures. They rely on internal knowledge of the system, in the form of topology, causality relationships, or other form of dependencies. The internal knowledge allows to draw conclusion on unobservable parts of the system.

6.2 Root Cause Analysis and Software-level Failure Management

RCA is a common AIOps topic which has already been discussed in Section 3.4.3. RCA is the process of inferring the set of elementary faults that caused a failure [370]. RCA is a fundamental reactive step for problem remediation, which is necessary for full recovery from failures and to ensure fulfilling the Service Level Agreeement (SLA) with minimal service interruption [75].

RCA is equally important for proactive failure management [73] as a method for collecting additional diagnostic information, interpret failure predictions and specialize preventive actions against failures to the specific components of subsystem that is expected to fail soon [235, 351]. Moreover, RCA frequently utilizes methodologies such as causality learning, pattern correlation, or graph propagation, which are important to understand the dynamics of a failing system also in a proactive setting. As it will be presented in the next sections, some RCA approaches, such as LogRule (Section 6.4), can be effectively translated into proactive FM methods by exploiting the causality mechanisms learned during post-mortem analysis.

6.2.1 Root Cause Analysis in Large-scale Services

Effective failure remediation is only possible when the root causes of the failure are fully discovered. An efficient troubleshooting depends on the quality and granularity of the logging and monitoring systems deployed in the various components of a large-scale distributed system (e.g., hardware telemetry, event logs, and distributed tracing).

In the available AIOps literature, RCA is often performed through service log analysis, mainly because logs are the most frequent source of diagnostic and monitoring information [38]. Recent trends favor *structured logging*, where information is presented in predefined structures using key-value pairs, so that they may be ingested by monitoring algorithms without requiring any pre-processing.

Manual RCA requires examination of heterogeneous monitoring information, such as system-level metrics, KPIs, and logs, to find one or more explanations for a failure event of interest. Due to the large quantity and complexity of the monitoring information, this task may become intractable for humans [370]. The key

```
Dec 10 06:55:46 LabSZ sshd[24200]: pam_unix(sshd:auth): check pass; user unknown
Dec 10 06:55:48 LabSZ sshd[24200]: Connection closed by 173.234.31.186 [preauth]
Dec 10 07:02:47 LabSZ sshd[24203]: Connection closed by 212.47.254.145 [preauth]
Dec 10 07:07:38 LabSZ sshd[24206]: Invalid user test9 from 52.80.34.196
Dec 10 07:07:38 LabSZ sshd[24206]: input_userauth_request: invalid user test9 [preauth]
Dec 10 07:07:38 LabSZ sshd[24206]: pam_unix(sshd:auth): check pass; user unknown
```

$\Downarrow$

Date	Time	Component	Process ID	Event ID	Event Template	IP Address
Dec 10	06:55:46	LabSZ	24200	E21	pam_unix(sshd:auth): check pass; user unknown	null
Dec 10	06:55:48	LabSZ	24200	E2	Connection closed by <*> [preauth]	173.234.31.186
Dec 10	07:02:47	LabSZ	24203	E2	Connection closed by <*> [preauth]	212.47.254.145
Dec 10	07:07:38	LabSZ	24206	E13	Invalid user <*> from <*>	52.80.34.196
Dec 10	07:07:38	LabSZ	24206	E12	input_userauth_request: invalid user <*> [preauth]	null
Dec 10	07:07:38	LabSZ	24206	E21	pam_unix(sshd:auth): check pass; user unknown	null

Figure 6.1 An example of structured log parsing [371]. The information presented in human-readable logs (top diagram) may be structured using the log message schema. Message fields may then be matched to string templates to extract additional columns, e.g., IP Address in this example. The final result is a tabular collection of data (bottom diagram).

to effective RCA is in the discovery of strong correlations between different system states, so that failure symptoms may be associated to their root causes. In this context, the correlations are often referred to as rules. If indications of a failure are present in individual log records (collection of system states), analyzing the correlations between the states and the failure may reveal that a particular system state is more likely to have caused the failure. For instance, observing a failure in log records corresponding to requests handled by the same IP address may indicate a potential problem in the host associated to that IP address (e.g., `{ip=172.146.XXX.XXX, port=YYYY}` $\Longrightarrow$ `failure`).

The application of automated RCA systems in a large-scale computing environment poses several challenges such as the necessity of large quantities of high-dimensional data, which in turn requires deploying efficient data processing and correlation algorithms. However, in modern systems the majority of existing logging information is stored in traditional, human-readable format and thus requires complex pre-processing steps, such as parsing, template matching, and value extraction. An example of log pre-processing is shown in Figure 6.1.

In contrast, structured logs provide diagnostic information in predefined machine-readable structures, which makes them suitable for automated processing. Therefore, structured logs are great enablers for RCA, due to their native compatibility with AI systems and their readiness for algorithm ingestion.

Association Rule Mining (ARM) [87], as formally introduced in Section 2.3.3, is an established method for automatic discovery of item relations in large databases. Given a formal description of a failure state, ARM employs statistical analysis to generate a list of observable system states, correlated to the observed failure, in the form of rules. ARM may be applied to any set of stateful observations, for example, structured logs.

6.2.2 Association Rule Mining for Root Cause Analysis

ARM [87] provides a statistical method to automatically discover causal relationships $X \Longrightarrow Y$, where X is an observable state and Y is a failure. The information contained in X can then be used to isolate the target problem. The final objective is to obtain a set of high-confidence rules $\mathbb{X}_f = [X_{f1}, X_{f2}, X_{f3}, \ldots]$ able to cover the set of transactions containing failure state Y. Examples of database, item base, and itemset in the context of structured logs are shown in Figure 6.2.

If logs are sufficiently informative, the generated output rules represent the root causes of the investigated failure, herein referred to as *explanation set*. Even though ARM is an effective method, its combinatorial nature limits its scalability to high-dimensional monitoring data collected from large-scale cloud systems. Therefore, in order to employ ARM for RCA in these systems, it is necessary to reduce the complexity and dimensionality of the data without losing important diagnostic information.

```
D = [
 t₁: {timestamp=1628513032, region=europe—east, size=4, fail=True}
 t₂: {timestamp=1628515048, region=europe—east, size=6, fail=False}
 t₃: {timestamp=1628579388, region=north—america, size=12, fail=False}
 t₄: {timestamp=1628582608, region=north—america, size=12, fail=True}
 t₅: {timestamp=1628587572, region=se—asia, size=6, fail=False}]

B = {timestamp=1628513032, timestamp=1628515048, timestamp=1628579388,
timestamp=1628582608, timestamp=1628587572, region=europe—east,
region=north—america, region=se—asia, size=4, size=6, size=12,
fail=True, fail=False}

X = {region=europe—east, size=6}
Y = {fail=True}
```

Figure 6.2 Example of ARM concepts (database, item base and patterns X, Y) in a structured logging context [371].

Previous works in log analysis have shown that logs contain redundant information, both in the row dimension [372,373], and in the column dimension [373,374]. A third type of redundancy is due to key-value-pair co-occurrence patterns, for which one or more key-value pairs always co-appear in the same log lines, and do not appear elsewhere. An example of co-appearing patterns is shown in Figure 6.1, for `EventID=E2` and `EventTemplate=Connection closed by <*> [preauth]`. Because these two elements always appear in the same rows, they can be effectively compressed into one item. The presence of redundancy in logs is vital for applications such as log templating and compression [374] and can be equally exploited for accelerating RCA via redundancy elimination [90]. While row redundancy has been addressed in ARM systems for RCA [90], column and co-occurrence pattern redundancies have not been investigated in the past.

6.3 Related Work

This section describes related work in dealing with failures in software proactively. As the applications and methods of proactive software FM are vast and heterogeneous, a comprehensive overview is not feasible. This section therefore focuses on the most relevant AI-based contributions, divided in failure prevention techniques, failure prediction and RCA.

6.3.1 Software Failure Prevention

Software Defect Prediction (SDP)

SDP discovers software bugs via analysis of source code (See Section 3.4.3). Methods vary in choice of data sources (code metrics vs. code revision history) and target code unit (function, class, module, etc.) [75].

Graves et al. [200] categorize software-quality predictors into two groups: product measures (such as code metrics) and process measures, which quantitatively represent the changing history of a software project. They support the latter group because product measures are deemed inconclusive from a correlation study. To predict the number of defects in the software repository of a telephone switching system, generalized linear models are built from process metrics. Additionally, a weighted-time damp model is presented, where code modifications are down-weighted based on age, which performs better overall.

Moser et al. [205] conducted a comparison analysis of code and change metrics on a repository of Eclipse projects. Three different Machine Learning (ML) methods - namely Naive Bayes, logistic regression, and decision trees - are used to perform a comparison. It has been demonstrated that using individual change metrics makes finding broken source files more effective than using only code metrics. Additionally, a combined approach slightly outperforms or produces comparable outcomes to a change metric strategy.

Convolutional Neural Networks (CNNs) for SDP are investigated by Li et al. [210]. A subset of Abstract Syntax Tree (AST) nodes corresponding to various kinds of semantic operations is extracted during the parsing step. These tokens are converted into numerical features by using embeddings, and they are then fed into a 1D convolutional network to learn intermediate representations of the input code. The final prediction is then made by combining these intermediate representations with manually created features. Results from the PROMISE dataset are compared with those from a traditional logistic regression model and a Bayesian Network approach, showing a significant F-score improvement (60.8%, +8.4% over traditional methods).

Checkpointing and Software Rejuvenation

Checkpointing [216–218] and software rejuvenation [133,137,194,214,215] are connected techniques to maintain clean operational states of software systems.

Checkpointing prevents failures by saving the system state before the occurrence of a failure. Multi-level checkpointing creates persistent-storage checkpoints at different component levels to improve flexibility and resiliency in large-scale systems [216, 218].

To this end, an incremental checkpointing technique for parallel cloud applications is presented by Jangjaimon et al. [218]. To predict turnaround time during program execution, their system uses an Adjusted Markov Model that takes into account both the impact of hardware failures and potential resource revocation events. The model also considers financial factors, such as cost and availability. An adaptive multi-level checkpointing scheme is advantageous for cloud paradigms with resource revocation, as it enables smaller checkpoint sizes, faster execution times (up to -25%), and reduced downtime cost (up to -20%).

Software rejuvenation, on the other hand, is a prevention technique which temporarily suspends a software execution to clean its internal state. Rejuvenation polices may either be periodic or on-demand, which require a scheduling policy. ML models have been used in the past to suggest efficient rejuvenation scheduling policies [137, 194]. Castelli et al. [194] apply stochastic reward networks to model service downtimes and draw conclusions on the efficiency of rejuvenation policies. Both time-based and prediction-based policies are taken into account in their analysis. They use resource exhaustion prediction methods for the latter, which are useful for estimating future failures and examining the expected downtime as a function of the prediction model accuracy and coverage. Both periodic and prediction-based policies are shown to be able to reduce downtime significantly, with prediction-based methods showing a larger overall improvement (-60% downtime at 90% coverage, vs. -25% with optimal time-based), with high-frequency periodic policies better tolerating simultaneous failures (-95%, vs. -85%).

Software Fault Injection

Fault injection is the deliberate introduction of faults into a target working system [134] to evaluate the level of fault tolerance reached [75]. Software faults can be inserted directly or can be simulated to infer the consequences of failures. The set of injected faults is defined faultload and is determinant in evaluating the success of fault injection.

Natella et al. [212] utilize two machine learning algorithms to enhance the faultload set. In particular, they use decision trees to categorize components into higher and lower fault risk. They also apply clustering on code metrics of software components to separate components based on their fan-in and fan-out interaction. Faults are injected into components of the cluster with the lowest fan-in and fan-out. The approach is tested on several commercial software frameworks and can reduce faultload size (up to -69%) and increase fault representativeness (up to +26%).

6.3.2 Software-level Online Failure Prediction

Some methods predict failures of software applications and large-scale services online.

Cohen et al. [235] investigate an approach based on Tree-Augmented Networks (TANs) to associate observed variables with abstract service states, in order to forecast and detect SLA violations and failures in Web services. Their approach is based on the observation of system metrics, such as CPU time, disk reads, swap space; and KPI-related measures, such as the number of served requests. The optimal graph structure composed of the optimal subset of input metrics is selected by means of a greedy algorithm. The approach can also be used for RCA, by exploiting the interpretability properties of TANs. Two fault injection experiments on servers are used for evaluation. The results show high detection rates (83–93%) and False Positive Rates (FPRs) ranging between 9.1% and 24%.

Salfner et al. [237] train Hidden Semi-Markov Models (HSMMs) for online prediction of failures on sequences of categorical events, which are collected from error logs. One HSMM is trained on failing sequences, a second one on non-failing sequences. Then, the sequence likelihood of the two models determines which of the two scenarios is more likely. The approach is general evaluated on logs of a commercial telecommunication system and compared with other prediction methods, achieving an F1-score of 74.19% and a FPR of 1.45%.

HORA [196] predicts service failures holistically, by combining system architectural knowledge and online time-series data to predict SLA violations and failures. Component failures are first predicted from system metrics using autoregressive predictors, then these failures are associated to system-wide problems by using component dependencies and failure propagation models. The approach is tested on a microservice-based application and compared to a monolithic approach (without architectural knowledge) by injecting memory leaks, node crashes, and system overloads during execution. HORA achieves a higher recall (83.3% vs. 69.2%) and Area under the Curve of Receiving Operating Characteristic (AUCROC) (0.920 vs. 0.837, +9.9%) compared to the monolithic approach.

6.3.3 Deployment and Runtime Verification

Velayutham et al. [368] propose a canary testing approach for a radio access network. To do so, they evaluated several sequence-based ML approaches, including Long Short-Term Memory (LSTM) and AutoRegressive Integrated Moving Average (ARIMA) predictors, for forecasting future time-series values and model uncertainty. These predictions are then used to make a final decision on the canary deployment.

Pritchard et al. [369] automate software update rollouts using a Reinforcement Learning (RL) based approach. The software rollout is model as a three-state automaton, on which the RL agent operates. The Q-learning algorithm is used to train the agent in taking optimal rollout decisions, such as preventing a software defect from being shipped.

Magpie [239] and Chen et al. [71] use Probabilistic Context-Free Grammars (PCFGs) to model execution traces from event sequences, which enables online anomaly detection through analysis of deviations from normal components interactions. The approach can also be used for failure prediction by modeling failure and healthy states, for fault localization, failure diagnosis and post-mortem analysis.

6.3.4 Root Cause Analysis

RCA is a complex task, reaching substantially beyond supporting proactive FM (Section 3.4.3). RCA techniques for software vary considerably in terms of goals, data sources, methods, and application scenarios. A large part of existing contributions provide solutions for very specific problems. For example, CloudRaid [375] focuses on discovery of concurrency bugs through log mining in the context of distributed system. Therefore, an exhaustive comparison of all RCA-related sub-problems is infeasible due to the substantial differences in various approaches. Since the RCA approach proposed in Section 6.4 strives for generality and targets a wide application domain, here only general-purpose RCA systems and previous RCA approaches based on ARM and structured logging are presented, as they are comparable in some aspect to the proposed approach.

General-Purpose RCA Systems

Two main groups of past contributions for general-purpose RCA can be identified.

NetMedic [376], G-RCA [377] and Giza [378] apply graph mining and causal inference techniques to uncover spatiotemporal correlations between observable failures and root causes. These approaches are used for diagnosing large-scale IP networks, where a mapping between system components (devices, processes, connections) and a graph model, composed of edges or nodes, is intuitive and easily constructable. These approaches are very effective in network applications, but they fail to show applicability whenever the complete system cannot be easily modeled as a graph.

Approaches in a second category of contribution include Adtributor [379], iDice [380], HotSpot [381], Log3C [382], and Squeeze [383], which perform RCA by means of statistical attribution of low KPI performance to specific system states, described by means of attribute-value pairs. KPI degradation is an important early warning symptom for failure, as well as important evidence signal to start investigating underlying causes in relation to performance issues. Oftentimes, however, the problem is already well-known as it affects only a subset of users. Thus, the analysis does not need to rely on high-level KPI information, but rather on fine-grained information collected on single hosts.

ARM-based RCA Systems

ARM has been frequently used for RCA in combination with structured logs. While various approaches only employ ARM for RCA [90, 94, 384], others apply a combination of supervised learning and ARM for the task [385–387]. There are also approaches available in the literature, which consider architectural enhancements from system perspective rather than algorithmic improvements [388, 389].

Lin et al. [90] propose a RCA framework based on ARM and structured log information. For mining patterns, they consider FP-Growth [89] as a valid replacement for the Apriori algorithm, and prioritize rules based on lift, confidence and support. Furthermore, in order to improve interpretability, they propose a rule refinement technique to remove redundant rules from the final explanation set.

Murali et al. [94] propose Minesweeper, an approach for mining sequential rules from app event traces (e.g., $A \rightarrow B \rightarrow C \implies failure$). Rules are discovered using Prefix-projected Sequential PAttern Mining (PrefixSpan) [92] and evaluated by measuring correlation metrics in faulty and normal groups of traces. Zhang et al. [384] also consider Sequential Pattern Mining (SPM) and propose a pre-processing step based on the item association matrix. They apply this technique for RCA in network alarm logs.

Castelluccio et al. [387] devise the tree-based approach (STUCCO) to classify software crash reports based on contrast set mining. Although supervised classification models like decision trees are usually faster and simpler to implement, they only produce a subset of all candidate explanations and may miss associations, which are too rare in the population to be selected. The results generated by these methods are less interpretable compared to that of pure ARM. Furthermore, the approach is not applicable to composite data types, such as version numbers or URLs/paths.

Arifin et al. [385] propose a supervised Naive Bayes model for SMS spam detection, where FP-Growth [89] is used for extracting discriminative features. Suriadi et al. [386] apply a supervised learning approach based on decision trees using process logs enriched with performance metrics for component attribution.

Bagui et al. [388] propose an efficient Hadoop implementation for ARM in large distributed environments using Apriori [87], while Liu et al. [389] employ ARM to analyze wireless network quality on the Big Data platform Spark. These architectural contributions focus on improving performance and scalability of ARM using Big Data platforms, but do not provide additional advantages in terms of interpretability or usability of the RCA framework.

6.4 Efficient Operator-based Pattern Mining for Root Cause Analysis and Failure Prevention

This section presents a novel pattern mining approach for RCA and failure prevention from structured logs. An introduction on motivation, current challenges and expected contributions is outlined in Section 6.4.1. The proposed LogRule algorithm and its implementation details are presented in Section 6.4.2. Experiments and evaluation results are presented in Section 6.4.3. An extension of the approach to sequential data is presented in Section 6.4.4.

6.4.1 Challenges and Contributions

Both structured and unstructured logs have been previously used for statistical analysis and RCA tasks [90, 94, 371, 376–383, 387–389]. However, the past approaches suffer from several limitations in terms of performance and usability.

Traditional supervised machine learning models [385, 386] do not enable a thorough root cause investigation, because they often provide a single-class as output, which produces a coarse level of detail not suitable for RCA. They also require manual feature engineering and large quantities of labeled data. They must also be retrained frequently to handle new types of failures.

Graph-based approaches [376–378] rely on existing topological information to discover failure correlations, which may not be available in dynamic or partially observable environments. Similarly, KPI attribution approaches [379–383] require numerical KPI information to correlate low performance metrics to individual system states.

ARM-based approaches should provide a natural advantage in terms of interpretability and adaptability to different input types. However, past ARM-based approaches [90, 94, 387, 388] have often suffered from poor result interpretability, because of overlapping and repetitive rules, which render the interpretation of results more difficult. Moreover, in the absence of efficient compression techniques, the exponential complexity of ARM results in prohibitively large execution times. Some approaches apply rule refinement techniques to reduce redundancy and to achieve smaller execution times [90, 387]. This, however, may discard important associations. Some past ARM approaches have also shown limited introspection ability, which hinders accuracy of the prediction results. Their effectiveness highly depends on problem-specific hyperparameter tuning.

To overcome these three main limitations, namely interpretability, performance and accuracy, a novel pattern mining framework for RCA is proposed. LogRule [371] is an efficient and interpretable RCA algorithm based on ARM. LogRule leverages structured logs to generate a list of explanations for events of interest. It may be used for analyzing sets of historical structured logs (composed of key-value pairs), where some entries are marked as failures. LogRule indicates possible root causes (as collectiona of relevant key-value pairs) by analyzing the correlation between the marked failures and other log records.

LogRule improves structured-log RCA in terms of performance, accuracy, and usability. The main contributions, to this end, are:

- the reduction of the problem dimensionality by up to 2000x, resulting in a faster and more efficient mining process to produce candidate explanations. This is achieved through the application of a new rule refinement algorithm and a novel database compression technique called *item-based aggregation*, which reduces the number of key-value pairs to be processed;

- reduction of the redundancies in the generated rules, which results in higher accuracy and, at the same time, improves interpretability of the results. This is achieved by introducing a novel rule evaluation metric called *disjunctive support*, which measures the level of overlap between different rules to produce a succinct set of rules;

- extension of the semantic meaning of extracted patterns and increasing introspection ability of the algorithm. This is achieved by introducing an extended introspective analysis for non-time-critical applications, called *operator-based rule mining*, which extends ARM with typed key-value pairs and operators.

The performance of LogRule is evaluated on a diverse collection of real and synthetic structured log datasets in terms of execution time and prediction accuracy. The results are compared with an existing approach as a baseline [90]. Evaluation results indicate a reduction in total computation time by over 97% (37x faster) compared to the baseline algorithm, while improving average precision and recall of explanation summaries. Moreover, operator-based rule mining further improves accuracy, achieving a final precision and recall of 0.936 (+0.718) and 0.907 (+0.130), respectively. A qualitative analysis of the rules generated by the algorithms is also performed, indicating that LogRule generates compactly-presented root causes, which are more useful for O&M operators to investigate common service problems.

6.4.2 The LogRule Algorithm

The architecture of the proposed LogRule algorithm is shown in Figure 6.3. LogRule is composed of four main steps: pre-processing, candidate mining, selection, and refinement. In the pre-processing step, the database $\mathbb{D}$ is constructed and compressed using two aggregation techniques. In candidate mining, frequent patterns X_c, which constitute the candidate explanations, are discovered from $\mathbb{D}$ using the FP-Growth [89] algorithm.

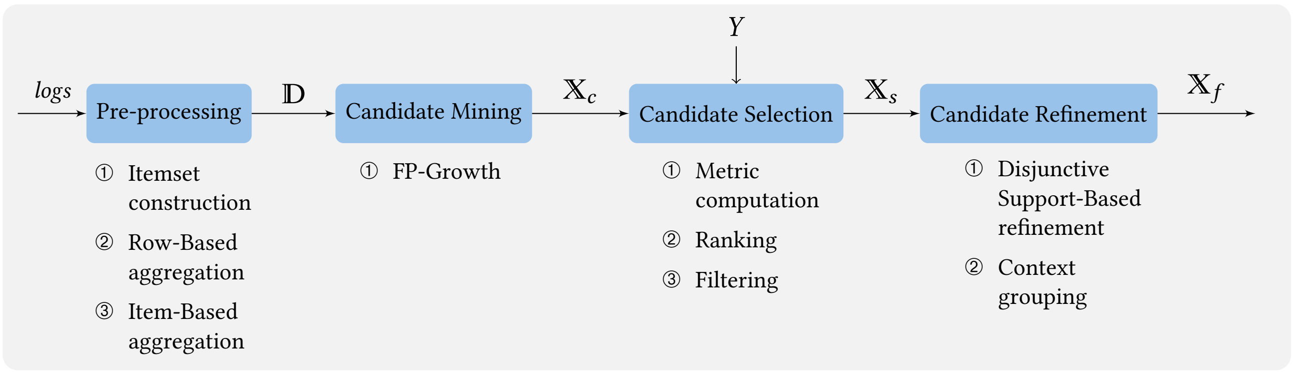


Figure 6.3 Pipeline of the LogRule algorithm [371]. Blue boxes are algorithm phases, numbered items below are the consecutive steps taken in each phase. First, structured logs are pre-processed to construct a transactional database $\mathbb{D}$. FP-Growth is then applied to generate a list of candidate explanations $\mathbb{X}_c$, which are verified and selected via statistical correlation with the input failure pattern Y. Remaining explanations $\mathbb{X}_s$ are then refined and organized to provide a final set of explanations $\mathbb{X}_f$.

In candidate selection, rules are ranked and selected based on support, confidence, and lift, to measure their correlation to a failure state of interest Y. In the refinement step, the remaining candidate rules are grouped and further refined to compute an optimal explanation set $\mathbb{X}_f$.

The pre-processing step is composed of two sub-phases, itemset construction and database compression.

Itemset Construction

The itemset construction step is necessary to transform the original data sources to the standard database format $\mathbb{D}$. During this step, operator-based rule mining is applied to the database to make the subsequent analysis steps more efficient.

The first step is the analysis of logs for optimal column selection. This step is necessary to prevent that the problem dimensionality becomes intractable, while ensuring that only relevant information in the logs is analyzed. LogRule selects optimal columns based on the count of distinct values, and the support of the individual key-value pairs. In fact, columns with a large distinct value count (timestamps, request IDs, ... are unlikely to be informative about large-scale problems and they impose a large toll on performance. Same applies to columns with constant value. In this step, it is possible to override the inclusion or exclusion of a specific column in the analysis. The user also has the possibility to select or exclude columns based on the total number of distinct values in the final database (e.g., $max\_distinct = 100$). Input logs are then loaded and transformed into a database using an itemset constructor, which transforms individual rows into itemsets. It is possible to choose from three different itemset construction algorithms:

- *Pure-itemset Constructor.* This constructor is used for non-tabular datasets, i.e., no key-value pair association (e.g., market basket analysis [390]). In this case, each log line contains a set of distinct elements, which is directly converted into an itemset.

- *Basic Key-value Pair Constructor.* This constructor is used for tabular datasets. Each row is converted into a set of items of the form “key=value”, for each key in the column schema [90]. While each unique value in a given column results in a key-value pair, only the key-value pairs present in a given row appear in the corresponding transaction.

- *Operator Key-value Pair Constructor.* A novel itemset construction technique, which extends the semantic meaning of key-value pairs, is proposed. In the available literature, equality is the only semantic relationship considered for associating key-value pairs. The LogRule algorithm permits other key-value operators during the itemset construction process, which enable more flexible and generalizable RCA compared to standard ARM, which is based only on categorical value with “=” operator.

Operator-based rule mining is implemented during the itemset construction process, where transactions are augmented with additional operator items. A *template set* is first constructed from values based on their semantic type and support for operators. For example, for a given IP address `192.168.10.1`, the template set {`192.168.10.*, 192.168.*.*, 192.*.*.*`} may be constructed. Transactions are then augmented with additional *template items*, which are generated using the template set. For example, all transactions containing `host = 192.168.10.1` are augmented with the following additional items: `host = 192.168.10.*`, `host = 192.168.*.*`, and `host = 192.*.*.*`. A type inference engine is implemented in LogRule, which is necessary to recognize column types in order to enable the application of operators. In the current implementation, the operators are supported for the following types:

- *Numeric values* (e.g., `3, 12.74`), which support the ordinal operators $\leq, <, >, \geq$, for example `latency≥1000ms, cpu_usage<80%`. Since the distinct count of numeric items is usually very large, it is preferable to use a predefined template set with arbitrary thresholds in order to generate template items. To limit computational complexity, LogRule uses as template set only the values observed in failure transactions, as they are more likely to represent an issue;
- *Version numbers* (e.g., `3.4.1, 1.7.0`), which support ordinal operators. If the count of unique version values is small, the generation of additional items can be exhaustive, i.e, the template set is the set of all observed versions. Otherwise, a smaller subset may be selected.
- *IP addresses* (e.g., `192.168.10.1`), which support set operators $\subset$ and $\subseteq$. For example, `192.168.10.1 ⊂ 192.168.10.*`. LogRule supports the CIDR /8, /16, and /24 classes [391] for template set construction.
- *URL and paths* (e.g., `/usr/bin/python3.6`), which support set operators $\subset$ and $\subseteq$, for example `/usr/bin/python3.6` ⊂ `/usr/bin`. In the current implementation, the template set is composed of all the intermediate nodes from the root to the final path, for example `/*`, `/usr/*`, `/usr/bin/*`. For URLs, the domain level hierarchy may additionally be employed to construct the template set, for example `www.example.com` ⊂ `example.com`.

In addition, the operator itemset constructor has the power to augment the database with new metadata columns, which are extracted after analyzing the initial column schema. For example, LogRule is able to extract the transmission protocol and port from the URL, or units of measure for numeric data. These new columns may be integrated into the analysis in order to hint to otherwise undiscoverable root causes.

Database Compression

After itemset construction, two compression techniques are applied to make the subsequent analysis steps faster. First, by *row-based aggregation* [90], identical transactions are pre-aggregated, where only unique rows are preserved and annotated with a corresponding weight (number of occurrences).

Moreover, a novel aggregation technique is proposed to enable ARM on high-dimensional datasets, called item-based aggregation. In contrast with row-based aggregation, this approach reduces the size of the item base by merging items with identical co-appearance pattern. Hence, if the support set of $b_1 \in \mathbb{B}$ is equal to the support set of $b_2 \in \mathbb{B}$, i.e.

$$b_1 \in t \iff b_2 \in t, \forall t \in \mathbb{D}, \tag{6.1}$$

b_1 and b_2 are replaced in the entire database with a new item $b_1 \equiv b_2$, where $\equiv$ indicates that the two items are interchangeable.

The execution time of all subsequent steps of LogRule is strongly dependent on the total item count k. Therefore, item-based aggregation results in smaller average execution time by reducing the total item count. Item-based aggregation still preserves the semantic content of the database during aggregation, because for

Transactional Database		**Association Rule Mining**	
t_i	transaction		
b_j	transaction item	X, Y	itemsets (*patterns*)
$\mathbb{D}$	transactional database	Y	target pattern
$n = \|\mathbb{D}\|$	number of transactions	T_f	ground-truth explanation set
$\mathbb{B}$	item base	$\mathcal{S}(X)$	support of pattern X
$k = \|\mathbb{B}\|$	number of distinct items in $\mathbb{D}$	$\mathcal{S}(X\|Y)$	conditional support of pattern X on Y
k'	k when using item-based aggregation	$\mathcal{C}(X, Y)$	Confidence of $X \implies Y$
m	number of columns	$\mathcal{L}(X, Y)$	Lift of $X \implies Y$

LogRule	
λ_m	item reduction factor
$\mathbb{X}_c, \mathbb{X}_s, \mathbb{X}_f$	set of mined, selected and refined patterns
$p = \|\mathbb{X}_c\|$	number of mined patterns
p'	number of mined patterns (with item-based aggregation)
λ_p	pattern reduction factor
β	confidence-support weight factor
$\mathcal{S}_\vee(X_1, \ldots, X_k \mid Y)$	disjunctive support of $X_1, X_2, \ldots$ conditional on Y
C	minimum-$\mathcal{S}_\vee$ stop threshold

Table 6.1 Notation table for LogRule [371]

each pair of aggregated items b_1, b_2 the information is preserved in the union item $b_1 \equiv b_2$. Therefore, employing item-based aggregation results in fewer patterns, which are, however, semantically equivalent to those generated in its absence. Evaluation results indicate that this approach reduces data dimensionality up to 60x for items and 2000x for patterns. Furthermore, fewer patterns result in shorter execution times in the selection and refinement steps.

Candidate Mining

Rule candidates are generated via frequent pattern mining after dataset construction and aggregation.

The Apriori algorithm [87] is one of the earliest algorithms used for frequent pattern mining [87]. Apriori is based on a candidate generation process, which constructs new item combinations recursively by selecting a new item from the item base $\mathbb{B}$ at each iteration. It then verifies the combinations that are effectively present in the database and measures their support. Patterns which satisfy a minimum support threshold are included in the final results. If a pattern does not reach the support threshold, all its supersets may also be excluded, because of the downward-closedness of support and, therefore, the recursion may stop. However, in the initial step a large number of patterns, which are not present in the database, may be generated. These, however, still require time for calculation of their support. The time complexity of the Apriori algorithm is exponential in the size of the item base: $O(2^{|\mathbb{B}|})$.

FP-Growth [89] improves Apriori by skipping the initial generation process. It constructs a dedicated data structure, called FP-Tree, to store item dependencies and mine only the patterns present inside the database. The frequent patterns are mined by using a divide-and-conquer strategy. Differently from the Apriori algorithm, FP-Growth does not require an internal candidate generation step and only examines existing item-to-item connections in the database once, resulting in a more efficient runtime $O(k^2)$. Algorithm 1 summarizes the algorithmic procedure of FP-Growth.

The slow candidate generation process that Apriori employs renders it impractical for mining patterns in high-dimensional datasets, which are often encountered in large-scale computing environments. Therefore, LogRule employs FP-Growth as a more efficient alternative.

LogRule further reduces the execution time of the mining step by exploiting log redundancy. This is possible to the size reduction of the item base during item-base aggregation, so that the final runtime complexity of the mining step is $O(k'^2)$, where $k' = \lambda_m \cdot k$ with k' the number of items resulting from item-based aggregation

Algorithm 1: FP-Growth algorithm [89, 371]

```
Data: FP-tree $Tree$, pattern base $\alpha$
Result: List of patterns $L$
$L \longleftarrow \emptyset$
if Tree contains a single path $P$ then
    for each combination $\beta$ of the nodes in $P$ do
        append to $L$ pattern $\beta \cup \alpha$ with support $\min(\mathcal{S}(X)|x \in \beta)$
    end
else
    for each $a_i$ in $Tree.header$ do
        append to $L$ pattern $\beta = a_i \cup \alpha$ with support $a_i.support$;
        construct $\beta$'s conditional pattern base;
        construct $beta$'s conditional tree $Tree_\beta$;
        if $Tree_\beta \neq \emptyset$ then
            FP-Growth ($Tree_\beta$, $\alpha$);
        end
    end
end
```

and $0 < \lambda_m = \frac{k'}{k} < 1$ defined as the *item reduction factor*. The value of λ_m varies across different inputs and depends on the degree of item co-occurrence in the database, but it is always less or equal than 1. By substituting k' in the equation, it can be shown that the mining step of LogRule is effectively $1/\lambda_m^2$ faster than FP-growth without a item-based aggregation step. The reduced number of items has also the effect of generating fewer candidates, by not repeating redundant patterns composed of co-occurring elements.

Another method used by LogRule to further accelerate the mining process, is to mine patterns exclusively on faulty-state transactions (containing Y, unlike other available approaches [94]. This is a vital dimensionality reduction step, as the ratio of faulty transactions, i.e., $\mathcal{S}(Y)$ is usually a very small fraction of the entire dataset. A detailed proof of the soundness of this step is shown in Appendix B.

Prior to candidate mining, item-based aggregation is executed again to further reduce the dimensionality of the selected transactions which contain the faulty state Y, because there might be new item-to-item equivalences that apply only to the transactions containing Y and that were previously unfit for compression. This enables selection of a very low minimum support threshold, making the analysis more precise.

The FP-Growth algorithm is then executed for pattern mining. After all candidate explanations have been generated, the output is then de-compressed and re-aggregated according to the initial decompressed format.

Candidate Selection

In this step, candidate rules generated during the mining step are evaluated and selected based on the results of metric computation, filtering, and ranking steps.

During the metric computation step, the conditional support, confidence, and lift metrics are computed for each candidate pattern. The computed metrics enable the identification of patterns that correlate with the failure state.

For filtering, LogRule discards rules with lift smaller than 1, which indicates that the rule and the failure state are not positively correlated. This threshold is adjustable depending on the specific use case and dataset. Since this process is easily parallelizable, the execution may be distributed among several processes in order to speed up filtering. The runtime complexity of metric computation and filtering is linear in the number of discovered patterns $O(p)$. However, because of item-based aggregation, the total number of patterns to analyze is effectively reduced to $O(p') < O(p)$, because of the item base compression described above. The pattern reduction ratio $\lambda_p = p'/p$ depends on λ_m as well as on the pattern characteristic of the input data. If the execution time for this step becomes noticeable, multi-processing is employed with *num_processes* = 12, else single-process computation is used.

Algorithm 2: Rule refinement based on disjunctive support [371]

```
Data: Candidate Rules $\mathbb{X}_s$, min disj. support threshold $C$
Result: List of explanations $\mathbb{X}_f$
$supp \longleftarrow 0$;
$\mathbb{X}_f \longleftarrow \emptyset$;
for each $X_c$ in $\mathbb{X}_c$ do
    $supp_{new} = \mathcal{S}_\vee(\mathbb{X}_f \cup \{X_c\})$;
    if $supp_{new} > supp$ then
        $\mathbb{X}_f \longleftarrow \mathbb{X}_f \cup \{X_c\}$;
        $supp \longleftarrow supp_{new}$;
        if $supp_{new} \geq C$ then
            break;
        end
    end
end
return $\mathbb{X}_f$
```

In the ranking step, patterns are sorted based on the values of the computed metrics. LogRule employs the Weighted Harmonic Mean (WHM) defined as

$$WHM_\beta = (1 + \beta^2)\frac{C \cdot \mathcal{S}}{\beta^2 \cdot C + \mathcal{S}}, \tag{6.2}$$

where C and $\mathcal{S}$ denote confidence and conditional support, respectively, and $\beta > 0$ is the support weight factor. Rules are sorted in decreasing order based on this measure. For small β, this prioritizes rules with higher confidence, which cover a large proportion of failure cases, as measured by conditional support. If two rules have equal WHM, LogRule prioritizes rules with more items because they are more informative.

Candidate Refinement

In this step, rules are refined and grouped to produce a final explanation set $\mathbb{X}_f$. In order to prevent similar rules from appearing in the final explanation set, a new metric is introduced, called *disjunctive support*, to evaluate the fitness of rules when grouped together, rather than individually evaluating rules based on their statistical significance. Disjunctive support is the support of the union of a given set of rules $\mathbb{X} = \{X_1, X_2, X_3, \ldots, X_k\}$ conditioned on a given pattern Y, or equivalently, the conditional probability of observing any of the patterns in $\mathbb{X}$ in the presence of Y. Hence,

$$\mathcal{S}_\vee(X_1, X_2, \ldots, X_k \mid Y) = P(X_1 \vee X_2 \vee \ldots X_k \mid Y). \tag{6.3}$$

Intuitively, disjunctive support measures the ratio of failure cases explained by a given set of rules $\mathbb{X}$.

The novel rule refinement Algorithm 2 is proposed [371]. By this algorithm, new rules are added to the explanation set only if they increase disjunctive support. Rules are examined in the order obtained from the ranking step of candidate selection. This ordering guarantees that more applicable rules with higher confidence are examined and added to the explanation set earlier than less applicable rules with smaller confidence. When a predefined minimum support threshold is reached, the algorithm terminates and returns $\mathbb{X}_f$. Hence, LogRule selects the highest-confidence rules that can also explain the largest fraction of failure cases. While rules that are supersets of one of the rules in $\mathbb{X}_f$ do not increase disjunctive support, because of downward-closedness property of support, their subsets may increase disjunctive support. This property prevents LogRule from selecting high-confidence rules that only apply in very specific contexts, while favoring aggregation of similar explanations into more general root causes. Because disjunctive support measures the fitness of the entire explanation set, instead of the fitness of individual explanations, it ensures that the explanation set is comprehensive and concise, since equivalent rules do not increase disjunctive support. The preliminary ranking step ensures that most confident rules are examined and selected before less confident rules.

Explanation set (after refinement)

```
[request_url=/login.html,http_method=POST,backend_version=1.9.0]
[request_url=/login.html,http_method=POST,backend_version=1.9.1]
[request_url=/login.html,http_method=POST,backend_version=2.0.0]
[request_url=/login.html,http_method=POST,backend_version=2.0.1]
[request_url=/index.html,http_method=GET,backend_version=2.0.0]
[request_url=/index.html,http_method=GET,backend_version=2.0.1]
            ...
```

$\mathcal{S}_\vee > 0.99$

$\Downarrow$

LogRule output

```
──────{request_url=/login.html,http_method=POST}──
|               [backend_version=1.9.0]
|               [backend_version=1.9.1]
|               [backend_version=2.0.0]
|               [backend_version=2.0.1]
──────────────────────────────────────────────────
```

Figure 6.4 Example of context grouping for similar rules [371]. Ranked rules are selected until the disjunctive support reaches a predefined threshold (e.g., 0.99). The selected rules are then grouped into contexts if the share one or more itemsets (in the example, request URL and HTTP method).

LogRule employs Algorithm 2 to jointly analyze the fitness of multiple explanations. Since the ranking process depends on the β parameter of the WHM (Equation (6.2)), β may be used to control the level of aggregation of root causes. This provides a means to the users to tweak the level of details in the produced root cause summary. Based on empirical testing, a choice of $\beta = 0.1$ and a minimum disjunctive support threshold of 0.99 produce well-interpretable results in most applications.

The final set of explanations $\mathbb{X}_f$ is then presented graphically by grouping similar explanations together. The rule grouping approach is shown in Figure 6.4, where the discovered rules are listed. If two or more patterns share a common sub-pattern, they are presented next to each other nested inside a common node/group called *context*. The goal is to reduce redundant information in the final explanation report in order to provide a more accurate and clear result. Although not shown in the figure for clarity, the output result of LogRule also reports the confidence and support for each rule in the final explanation set, to allow users to evaluate the quality and comprehensiveness of the discovered associations.

The proposed refinement approach only analyzes metrics for the small set of rules selected by Algorithm 2, and not all the rules produced by the rule selection step. This enables a faster and more complex refinement analysis.

6.4.3 Evaluation

The LogRule algorithm [371] has been implemented as a Command-Line Interface (CLI) tool in Python. End users may specify a list of log files to analyze and the target failure pattern Y. The tool also enables users to specify optional parameters such as minimum support and lift thresholds. The user may also enable or disable individual functionalities such as aggregation or refinement algorithms through the CLI. The tool takes as input the logs and the itemset Y describing the target failure and produces a set of itemsets $\mathbb{X}_f = \{X_{f1}, X_{f2}, \ldots\}$ hinting at potential root causes of Y.

The performance of the LogRule algorithm is evaluated in terms of execution time, accuracy and interpretability. In order to measure the impact of the proposed algorithms, experiments are performed on three incremental versions of LogRule, each introducing one of the proposed improvements (item-based aggregation, disjunctive support-based grouping, and operator-based rule mining).

As a baseline for comparison, the Fast Dimensional Analysis algorithm [90] is considered, which combines FP-growth, the most established efficient algorithm for frequent pattern mining [392], with support- and lift-based rule selection, which have been found to be the most important metrics to discover important associations [393].

Algorithm	**Features**			
	Aggregation		Refinement Method	Operators Support
	Row-Based	Item-Based		
Baseline [90]	✓		lift-support refinement	
LogRule-lite	✓	✓	lift-support refinement	
LogRule	✓	✓	disjunctive support grouping	
LogRule-op	✓	✓	disjunctive support grouping	✓

Table 6.2 Feature summary of the four algorithms evaluated [371].

As discussed in Section 6.3 and to the best of the author knowledge, other available works in the same problem domain consider architectural improvements from system perspective rather than algorithmic improvements. The incremental algorithm versions and their features are summarized in Table 6.2. For the baseline algorithm (and LogRule with item-based aggregation), the refinement step corresponds to the "interpretability optimization" step proposed in the original paper [90].

Evaluation Datasets

The proposed algorithms are evaluated on a heterogeneous collection of structured-log datasets. The datasets and experiment compositions are summarized in Table 6.3, where an experiment is the execution of a specific algorithm on a given dataset to analyze root causes of a given failure state Y.

In order to ensure a comprehensive and diverse set of experiments, which would represent real-world scenarios, several experiments have been performed on the same dataset by changing the target failure state Y. In each experiment, predefined values are set for algorithm hyperparameters, such as the minimum support threshold and the maximum input size, equal for all algorithms. All other configuration parameters of algorithms are equal across all experiments. This resulted in a set of 49 experiments. All the evaluated algorithms undergo the complete set of experiments, for a total of 49*4 algorithm executions. An algorithm execution is considered to be successful if it is completed within 6 hours.

The evaluation datasets may be divided into three categories: synthetic data, data collected from test environments, and production data. In order to evaluate accuracy, each experiment is annotated with a set of ground truth explanations $\mathbb{T}_f = \{T_{f,1}, T_{f,2}, T_{f,3}, \ldots\}$, which contains key-value pairs corresponding to the explanations experts would expect to see. The next sections provide additional details about the content of structured logs and the selection of the ground truth set $\mathbb{T}_f$.

For synthetic data, synthetic log files were directly constructed with predefined failure patterns that are difficult to detect using "vanilla" ARM. The synthetic structured log files contain records of HTTP-like requests, storing information such as request status, user agent, call method, protocol version, location, request time and URL. For each of these files, random key-value pairs are assigned using predefined value sets. The request status indicates whether the request was completed successfully. As the dataset is synthetic, the ground truth

Dataset	**Experiments**	**Rows**	**Columns**	**Y Example**	**Remarks**
Synthetic	8	100000	12	`{success=False}`	Synthetically generated scenarios
OpenStack Access [57]	10	30	*590*	`{status_code=500}`	Simulated network faults
SockShop [394]	12	466335	162	`{loglevel=ERROR}`	payment error and others (log-level based)
Storage Service Access	5	810799	230	`{http_status=500}`	company internal
Apache Access [395,396]	7	*147846*	12	`{status_code=403}`	Access logs of a live Apache server
LogPAI [397]	7	*18750*	*15*	`{Level=error}`	misc. log collection (Hadoop, Android, …)
Total	**49**				

Table 6.3 Composition of the set of experiments conducted [371], by dataset. *Italicized* values denote that the number of rows/columns vary across experiments. Reported values are the maximum dimensions.

$\mathbb{T}_f$ is predefined as part of the experiment creation. Records containing any of the key-value-pair sets of $\mathbb{T}_f$ are marked as unsuccessful under the request status column.

Test environment data originates from two different environments, SockShop [394] and OpenStack [57]. SockShop is a demo microservice environment simulating an e-commerce website. A synthetic load was applied to the front-end interface and instrumented each microservice to collect logs and pre-process them when necessary. Various experiments are constructed by changing the target failure state Y. For example, in one experiment the algorithm must diagnose a failure reported during the payment process. The failure appears whenever the shopping cart value exceeds a specific amount ($100). Because the cart state is observable in the logs in the form of an error message, a ground-truth set of key-value pair explanations $\mathbb{T}_f = \{[\texttt{message} = \text{“}\texttt{Total exceeds } 100\$\text{”}]\}$ is constructed. The same instrumentation and load process was applied on an OpenStack testbed [57] composed of various microservices. To obtain a realistic fault, the communication between two microservices is inhibited by disabling communication ports. Both the port numbers and microservice name are included in the structured log schema, so the combination of these two key-value pairs constitutes the ground truth.

For real production experiments, publicly available log datasets [395–397] are used, as well as company-internal access logs collected from production environments. For the public log datasets, two categories are selected: 1) files from the LogPAI repository [397], an established benchmark for log analysis, generated by different software applications (Apache, Hadoop, Android, BlueGeneL, OpenSSH) and 2) a collection of publicly accessible Apache access logs [395, 396]. For the company-internal logs, an analysis is performed to identify the root causes of some sporadic server-side errors occurred in a specific cloud availability region. An access log dataset is collected from the bucket storage service of a large cloud provider in a collection frame of 7 minutes, during which some anomalous service behavior was detected. The collection monitors the requests received in the impacted region (composed of 10456 hosts) and contains $\sim$ 800k rows (corresponding to individual requests), out of which approximately 7000 lead to a server error (50X). Each request is annotated with 230 request fields, including client/server versions, involved hosts, request method, operation type, completion time statistics, and other configuration properties. From this dataset, several experiments are constructed by varying the input dataset size (using 3, 10, 30, 100% of the entire row collection and by varying column size) and by running the algorithms with different target failure Y (e.g., $[\texttt{status\_code} = 500]$, $[\texttt{status\_code} = 503]$, $\ldots$). The ground-truth explanation set for this group of experiments was identified through manual analysis and comparison of the failing requests by humans.

Evaluation Setup

To ensure fairness, the complete set of experiments was executed for all the evaluated algorithms and the hyperparameters are set to equal values for the same experiment (see Table 6.4).

All the experiments are run individually on the same server machine, a Virtual Machine (VM) hosted in an internal elastic-compute cloud service with 8 CPU cores (Intel Xeon Gold 6278C @2.60GHz) and 64 GB of memory. For each experiment, the execution time of the loading, mining, selection, and refinement steps of the algorithms were measured using the built-in Python library `time`. Furthermore, the accuracy of the output explanation set $\mathbb{X}_f$ is evaluated by measuring *precision* $\mathcal{P}$ and *recall* $\mathcal{R}$ as

$$\mathcal{P} = \frac{|\mathbb{T}_f \cap \mathbb{X}_f|}{|\mathbb{X}_f|} \quad \text{and} \quad \mathcal{R} = \frac{|\mathbb{T}_f \cap \mathbb{X}_f|}{|\mathbb{T}_f|}, \tag{6.4}$$

respectively.

A high precision $\mathcal{P}$ indicates that most of the rules reported in the explanation set $\mathbb{X}_f$ are actual root causes present in the ground truth set $\mathbb{T}_f$, while a high recall $\mathcal{R}$ indicates that most of the actual root causes in the ground truth $\mathbb{T}_f$ are included in the output explanation set $\mathbb{X}_f$. Based on this definition, these metrics functionally correspond to the homonymous metrics defined in Section 2.3.5. The number of items and patterns before and after item-based aggregation is also recorded.

parameter	value	experiments
minimum support	0.05	synthetic-1
	0.4	openstack15XX-all
	0.6	openstack15XX-2, apache-access-1
	0	others
minimum distinct values	2	all
minimum lift threshold	3	all
β	0.1	all
C	0.99	all

Table 6.4 Hyper-parameter configuration used in the experiments [371].

Execution Time Evaluation

Execution time of the algorithms is shown in Table 6.5 and Figure 6.5. The comparison of the execution time between the baseline approach and LogRule-lite, which highlights the effect of item-based aggregation on execution time, is shown in Figure 6.5. In these experiments, the proposed aggregation method reduces the number of generated patterns (x-axis) up to a factor of 2000x (average 315x), while maintaining the semantic properties of candidates. Moreover, item-based aggregation helps reduce the execution time of the mining, selection, and refinement steps of the algorithm by at most 800x, 2100x, and 45000x, respectively (see lowermost point of each series in Figure 6.5).

Average execution time by algorithm step is shown in Figure 6.6. LogRule increases the execution time of the loading step on average because of the additional item-based aggregation. This, however, results in reduction of execution time in the consequent steps, with time ratios ranging from 1 (equal execution time) to a maximum of $3 \cdot 10^{-3}$ (320x speedup). Furthermore, the disjunctive support approach reduces the refinement time from 9.541s to 0.175s (see column 2 and 3). Overall, LogRule reduces average computation time from 212.818s to 5.816s (37x, -97.3%). In the case of LogRule with operator-based rule mining enabled (LogRule-op), the creation of operator-based itemsets imposes a small overhead (1.878s) and increases the average mining and selection times to 4.694s and 818.980s, respectively. However, the average refinement time is not affected (0.177s vs. 0.175s).

Accuracy Evaluation

The number of experiments that terminated prematurely due to out-of-memory exception or a timeout (set to 1 hour), as well as precision, recall, and F1-score of the algorithms are listed in Table 6.6. The base versions of LogRule (i.e., without the extended operator mode) are more stable than the other algorithms, resulting in a success rate of 100% in the performed experiments.

LogRule-lite increases average precision by 0.364 (+167%) and recall by 0.15 (+19%) compared to the Baseline algorithm. The precision improvement is attributed to the item-based aggregation because it removes redundant patterns. The introduction of disjunctive support grouping (LogRule) further improves precision (+0.332, +57%). Disjunctive support prevents similar rules from being simultaneously selected, hence, reducing over-reporting, which increases the precision. A slight negative effect on recall (-0.034, -3.7%) is also

Algorithm	Execution Time (s)					Dimensionality Reduction	
	Loading	Mining	Selection	Refinement	Total	Items	Patterns
Baseline [90]	0.594	0.141	25.442	186.641	212.818	-	-
LogRule-lite	0.818	0.034	4.805	9.562	15.218	-89.79%	-99.68%
LogRule	**0.824**	**0.032**	**4.786**	**0.175**	**5.816**	-89.79%	-99.68%
LogRule-op	1.878	4.694	818.980	0.177	825.728	-89.05%	-99.10%

Table 6.5 Average execution times (seconds) and dimensionality reduction factor of LogRule [371], compared to the baseline algorithm [90]. The introduction of item-based aggregation (compare row 2 vs. 1) drastically reduces mining, selection and refinement time with a negligible loading overhead. The introduction of the new refinement algorithm (row 3 vs. 2) further reduces refinement time.

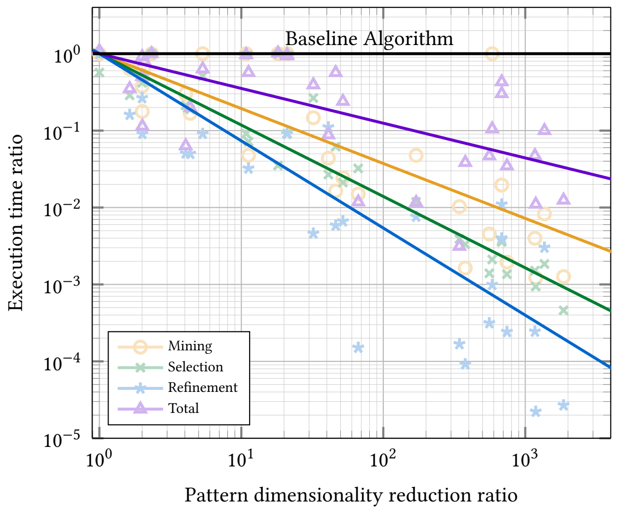


Figure 6.5 Execution time ratio as a function of pattern reduction ratio for LogRule-lite [371]. Item-based aggregation reduces the number of patterns (x- axis) to be processed, consequently resulting in smaller execution times (y-axis) in all steps of the algorithm. Single dots represent individual experiments, for which linear regression lines are constructed.

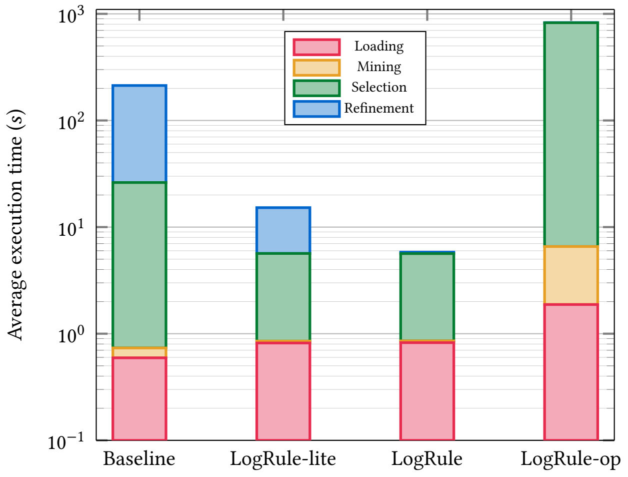


Figure 6.6 Comparison of the execution time of the algorithms evaluated [371], in seconds. The contributions of various steps of the algorithms are shown as a fraction of the total execution time. The introduction of item-based aggregation in LogRule-lite and of disjunctive-support grouping in LogRule both speed up the total execution time of the algorithm.

observed. The improvements compared to the Baseline algorithm are 0.696 (+319%) and 0.116 (+15%) in terms of precision and recall, respectively. LogRule-op reports more accurate results than LogRule. The operators are beneficial in the experiments, in which it is possible to aggregate root causes based on a version, IP class or URL pattern. The F1-score of LogRule-op is 0.921, which is +2.0% and +17.1% better compared to LogRule and the Baseline algorithm, respectively.

Qualitative Considerations

In this section, considerations regarding the quality and interpretability of results are presented, by comparing the outputs of LogRule and Baseline [90] algorithms.

One of the synthetic examples consists of a collection of virtual service requests annotated with HTTP method, request URL, server backend version and user agent (e.g., Chrome). These requests are manually constructed, such that a 500 HTTP request status code is generated for POST login requests handled by server versions $\geq$ 1.9.0. Hence, `Y=[http_status=500]` and the true explanation is `X=[backend_version=[1.9.*|2.0.*], http_method=POST, request_url=/login.html]`. The explanation set generated by the Baseline algorithm is shown in Figure 6.7. The first four rules (lines 1–4) are correctly identified. However, rules in lines 5–9 are redundant, because they describe specializations of the real root cause scenario, and the remaining rules (lines 10–14) are incorrect. An in-depth analysis reveals that the first four rules have a higher lift (31.7) compared to the rules in lines 5–9 (23.8 or less). Therefore, the Baseline algorithm requires tuning

Algorithm	Incomplete Executions	Precision	Recall	F1-score
Baseline [90]	17 (35%)	0.218	0.777	0.340
LogRule-lite	0 (0%)	0.582	0.927	0.715
LogRule	0 (0%)	0.914	0.893	0.903
LogRule-op	5 (10%)	**0.936**	**0.907**	**0.921**

Table 6.6 Average precision, recall, and F1-score of experiments conducted [371] by algorithm, computed across the 49 successful experiments (failed executions reported in the second column).

of the minimum lift threshold in order to generate a more accurate set of explanations. Nevertheless, all rules in lines 5–9 would still appear in final results because they are correct, even though, redundant.

The output of LogRule in this experiment is shown in Figure 6.8. Only the top four rules (by confidence) are selected, as they already cover 100% of the failure cases. Moreover, all rules share a common context, therefore explanations are grouped and only the different items of each pattern are reported. LogRule-op produces the output shown in Figure 6.9. A single explanation is produced as all failures satisfy this pattern. This example demonstrates the benefit of operator-based analysis compared to a categorical mining approach in generating more concise results. Differently from some previous methods [379, 380], which only return single-attribute explanations, the proposed approach discovers multi-attribute root causes.

In another synthetic experiment, requests for the same page were analyzed. Some of these requests will fail when the user posts an HTTP request with the ‘search’ parameter embedded in the URL. It is difficult for ARM-based RCA algorithms to detect this situation, because there is no unique key-value pair associated to the failure. The outputs of LogRule and LogRule-op for this example are shown in Figure 6.10 and 6.11. While both algorithms correctly identify the root cause with 100% disjunctive support, LogRule reports one rule for each URL correlated with the failures. LogRule-op, to the contrary, is able to combine these sub-cases into a single, more general rule, because of its ability to introspect URLs and extract new attributes.

In summary, LogRule-op is able to conduct more detailed and introspective analysis compared to LogRule and the baseline, which is indicated for use scenarios where a time-sensitive analysis is not a requirement.

```
1   ['backend_version=1.9.0', 'http_method=POST', 'request_url=/login.html']
2   ['backend_version=1.9.1', 'http_method=POST', 'request_url=/login.html']
3   ['backend_version=2.0.0', 'http_method=POST', 'request_url=/login.html']
4   ['backend_version=2.0.1', 'http_method=POST', 'request_url=/login.html']
5   ['backend_version=2.0.1', 'http_version=2.0', 'location=africa', 'request_url=/login.html']
6   ['backend_version=2.0.1', 'location=europe-west', 'request_url=/login.html',
        'user_agent=Opera']
7   ['backend_version=2.0.0', 'http_version=1.1', 'location=oceania', 'request_url=/login.html']
8   ['backend_version=2.0.0', 'location=oceania', 'request_url=/login.html',
        'user_agent=Firefox']
9                                  ....
10  ['http_method=POST', 'http_version=1.1', 'location=africa', 'request_url=/login.html']
11  ['http_method=POST', 'http_version=1.0', 'location=asia-rest', 'request_url=/login.html']
12  ['http_method=POST', 'http_version=2.0', 'location=se-asia', 'request_url=/login.html']
13  ['http_method=POST', 'request_url=/login.html']
14                                 ....
```

Figure 6.7 Output of baseline algorithm [90] for the version problem. Rules 1–4 are correct, while remaining explanations appear because of high lift but are redundant [371].

```
------{request_url=/login.html,http_method=POST}--
|               [backend_version=1.9.0]
|               [backend_version=1.9.1]
|               [backend_version=2.0.0]
|               [backend_version=2.0.1]
--------------------------------------------------
```

Figure 6.8 Output of LogRule for the version problem [371]. Only a minimal set of relevant explanations is shown, as these rules cover the totality of failure cases. Rules are moreover grouped by similar context.

```
['http_method=POST', 'request_url=/login.html', 'backend_version>=1.9.0',
    'backend_version<=2.0.1']
```

Figure 6.9 Output of LogRule-op for the version problem [371]. The four correct rules of Figure 6.8 are aggregated into one general rule thanks to operators.

```
──────{'http_method=POST', 'user_agent=Edge'}──────
|    ['request_url=/products.html?search=coffee+table']
|    ['request_url=/products.html?search=sofa']
|    ['request_url=/products.html?search=&page=1']
|    ['request_url=/products.html?search=outdoor+table']
|    ['request_url=/products.html?search=sofa&page=2']
|    ['request_url=/products.html?search=outdoor+table&page=4']
|    ['request_url=/products.html?search=rocking+chair']
────────────────────────────────────────────────────────────
```

Figure 6.10 Output of LogRule for the search URL problem. The algorithm identifies a complete and correct set of rules. In addition, LogRule-op can summarize the complete set in one rule.

```
['http_method=POST', 'location=*', 'request_url/params/search=*', 'request_url=/*',
    'user_agent=Edge']
```

Figure 6.11 Output LogRule-op for the search URL problem. As above, the algorithm identifies a complete and correct set of rules. In addition, LogRule-op can summarize the complete set in one rule.

6.4.4 Extending LogRule for Sequential Pattern Mining

An assumption of the pattern mining framework introduced so far is the mutual independence of log entries, which describe individual requests in the use cases; however, logs frequently encode a sequence of states of the same program, which does not translate into independent lines (or observations) across time.

The problem is generalizable based on single and sequential state characterizations. Single state problems contain the observable system state within one itemset, and itemsets correspond to transactions. Sequential state problems contain the observable system state in a series of itemsets, each containing one or more items, which together encode the state as a sequence. Therefore, it is desirable to extend the proposed algorithm to a SPM framework, which is able to analyze and model temporal dependencies present inside sequential logs.

The new database transaction then becomes an *ordered* sequence of itemsets of the form $\mathbf{t_i} = [t_{i1}, t_{i2}, \ldots, t_{ij}, \ldots, t_{iL}]$, where $t_{ij} \subseteq \mathbb{B}$ is an itemset as defined previously. These sequences compose a transactional database $\mathbb{D}$ (see also Section 2.3.3). Then, SPM [92, 94] discovers rules of the form $\mathbf{X} = \{X_1 \rightarrow X_2 \rightarrow \ldots \rightarrow X_j \rightarrow \ldots \rightarrow X_t\} \implies Y$, where X_j, Y are itemsets.

For the pre-processing step, the concepts previously introduced still apply. Instead of operating item-based aggregation for each transaction (i.e., sequence), it is applied at each time-step t_{ij} of each transaction $\mathbf{t_i}$. This is because item-based aggregation is applicable to itemsets, and sequences are ordered collections of itemsets; therefore the compression must be applied at each individual time-step. Row-based aggregation, on the other hand, is still applied on the transaction level, i.e., identical sequences are replaced by one instance with a corresponding weight factor equal to the occurrence count. All operations on itemsets and all itemset constructors are preserved.

For the mining step, SPM algorithms, such as PrefixSpan [92], are employed. They discover frequent sequential patterns $\mathbf{X}$ as described above, which are used as candidate rules.

ARM Concept		Itemset Mining		Sequential Pattern Mining	
Symbol	Interpretation	Definition	Type	Definition	Type
t	transaction	$t \subseteq \mathbb{B}, \mathbb{B} = \{b_1, b_2, \ldots, b_k\}$	itemset	$\mathbf{t} = \left[t_1, \ldots, t_j, \ldots, t_L\right], t_j \subseteq \mathbb{B}$	ordered sequence of itemsets
X	pattern	$X \subseteq t$, for at least one t	itemset	$\mathbf{X} = \{X_1 \rightarrow \ldots \rightarrow X_j \rightarrow \ldots \rightarrow X_t\} \subseteq \mathbf{t}, X_j \subseteq \mathbb{B}$	ordered sequence of itemsets
Y	failure state	$Y \subseteq \mathbb{B}$	itemset	$Y \subseteq \mathbb{B}$	itemset
$\mathcal{S}(X)$	$P(X)$	$\lvert\{X \subseteq t \mid \forall t \in \mathbb{D}\}\rvert/\lvert\mathbb{D}\rvert$	real, $0 \leq \mathcal{S}(X) \leq 1$	$\lvert\{\mathbf{X} \subseteq \mathbf{t} \mid \forall t \in \mathbb{D}\}\rvert/\lvert\mathbb{D}\rvert$	real, $0 \leq \mathcal{S}(\mathbf{X}) \leq 1$
$\mathcal{S}(X \mid Y)$	$P(X \mid Y)$	$\mathcal{S}(X \cup Y)/\mathcal{S}(Y)$	real, $0 \leq \mathcal{S}(X \mid Y) \leq 1$	$\mathcal{S}(\mathbf{X}^\frown Y)/\mathcal{S}(Y)$	real, $0 \leq \mathcal{S}(\mathbf{X} \mid Y) \leq 1$
$\mathcal{C}(X, Y)$	$P(Y \mid X)$	$\mathcal{S}(X \cup Y)/\mathcal{S}(X)$	real, $0 \leq \mathcal{C}(X, Y) \leq 1$	$\mathcal{S}(\mathbf{X}^\frown Y)/\mathcal{S}(\mathbf{X})$	real, $0 \leq \mathcal{C}(\mathbf{X}, Y) \leq 1$
$\mathcal{L}(X, Y)$	odds increase	$\mathcal{C}(X, Y)/\mathcal{S}(Y) = \mathcal{S}(X \mid Y)/\mathcal{S}(X)$	real, non-negative	$\mathcal{C}(\mathbf{X}, Y)/\mathcal{S}(Y) = \mathcal{S}(\mathbf{X} \mid Y)/\mathcal{S}(\mathbf{X})$	real, non-negative

Table 6.7 Comparison of metrics for traditional ARM and SPM. For a sequence X to be part of a transaction $\mathbf{t}$ (i.e., $\mathbf{X} \subseteq \mathbf{t}$), the itemsets of X must all appear in $\mathbf{t}$ in the same order as in X. The $\frown$ symbol indicates concatenation.

For the rule candidate selection step, the notions of confidence, support, and lift can be extended [94]. Table 6.7 compares the definition of metrics in ARM and SPM, to clarify the extension.

Based on these generalized definitions, the same filtering and refinement techniques or the single-state case may be applied.

With respect to the grouping and result presentation, the same itemset grouping techniques are again applicable. Because the presented rules are now sequential, additional grouping may be applied, on the basis of shared sub-pattern sequences across selected rule patterns. However, this was not considered for this extension.

The extension was implemented in Python in the form a CLI command, as for the base program. It was tested on several single-state and sequential-state structured logs to evaluate its applicability and usefulness. The next sections describe the potential applications of both the base and the sequential algorithm for pattern mining.

6.5 Applications of LogRule for RCA and Software-level Failure Management

The ARM framework utilized in LogRule [371] is general, so that it can be extensively applied to numerous O&M scenarios. In this section, a list of real-world applications for the LogRule framework is presented, based on the past experience in using LogRule, as well as on theoretical speculation on where the LogRule may bring the highest benefits.

LogRule draws its introspection power from the ability to observe multiple execution instances, and to perform a differential analysis of symptoms between healthy and faulty states of a system. These multiple instances either arise from multiple parallel executions of the same software (e.g., function invocations, or HTTP request callbacks), or actual parallel systems (e.g., multiple VMs, hosts, containers, ...). The former can be defined as a *dynamic invocation* of a system, and the latter as a *static invocation* of a system, as they system state is assumed to be static for the duration of the analysis.

Moreover, LogRule was originally designed to model single state problems, i.e., problems where the state of a system is completely manifested within one observation (or transaction) of the observable input data. Based on the discussion in Section 6.4.4, LogRule can be extended to work on sequential state data, where the state of the system is manifested on multiple, temporally-separated observations of the same system.

Each problem can then be categorized as either single or sequential state, as well as either static or dynamic in invocation. Figure 6.12 summarizes the different combinations of system state modalities and invocation types with some examples.

6.5.1 Applicability to Single State Problems

The discussion on applicability of LogRule starts by focusing on single-state problems. In addition to the experiments presented earlier in Section 6.4.3, LogRule was considered for several other application scenarios.

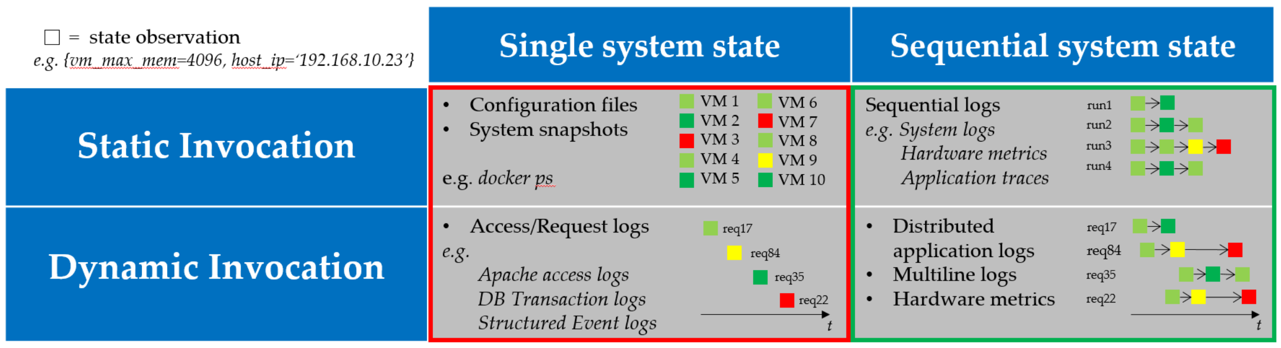


Figure 6.12 Diagram of applicability contexts of LogRule [371]. The red box indicates contexts where base LogRule is directly applicable. The green box indicates contexts where the LogRule extension to SPM is applicable. The bottom-right scenario (sequential system state, dynamic invocation) requires to first introduce a de-interleaving system to separate independent invocations (e.g., requests from different users) inside logs.

Use Case	Data Sources	Ground-truth (Y)	Objective
Access Log Analysis	Access logs	Request Status	Identify request patterns causing failures / DDoS
Hardware Failure Prediction	Hardware Metrics	Failures	Feature selection, correlation, failure prediction
Service-level RCA	Host/region state	KPIs, alarms	Discover causes of local performance degradation
Function Verification	Parameter values	Return status	Troubleshooting / Debugging software functions
Configuration Analysis	CMDB	Server Tickets	Identify erroneous server configurations
DB Association Analysis	DB Table	WHERE clause items	Discover hidden key-value relationships

Table 6.8 Use cases of single-state scenarios for LogRule [371].

Here are evaluated other possibilities to apply LogRule to different situations, and potential use cases in large-scale computing environments are explored. Table 6.8 presents a list of the all the single-state use cases identified for LogRule. Below the most important use cases are discussed in detail.

Access logs are collections of resource requests submitted to servers by end users. They are frequently used in various network and service monitoring applications. These logs are commonly used for RCA, because they contain request information in a predefined structured format with clear failure indications, for example HTTP request status. Furthermore, each log entry contains sufficient diagnostic information about the state of a request, contrary to conventional logs, where the request state must be reconstructed from a sequence of log lines. LogRule was tested on a variety of access logs, including logs from company internal production systems, where the root causes of server-side errors were correctly identified even in challenging scenarios, in which only a unique combination of request parameters would result in an error.

Three application scenarios are identified for access log analysis: load balancer analysis, Application Programming Interface (API) gateway monitoring, and SSH access analysis. Load balancers ensure reliability and efficiency of services by distributing application traffic to different servers. The traffic distribution is performed based on request information such as HTTP headers, cookies, or internal application data. Load balancers may also record which servers handled one or a group of similar requests. Therefore, load balancer access logs create fertile ground for automated structured logging RCA. Similarly, API gateway monitoring may record request features and their failure behavior to discover potential patterns in specific URL, methods or data formats. SSH access logs can be collected from bastion hosts to identify root causes of common issues encountered by users during connection. SSH logs are also vital to identify and protect from cyber-attacks. For example, LogRule can be applied to identify the common origin of requests during a Distributed Denial of Service (DDoS) attack. In the Apache use case, LogRule correctly identified as root cause of failing Web server requests the incorrect format of the HTTP method (‘GET’ $\neq$ ‘get’) in the request header. In the OpenStack deployment experiments, LogRule reported IP addresses, IP ports, service names which all linked back to the origin of the problem.

RCA for hardware failures in datacenters is an important yet challenging task. LogRule may be employed for feature selection by finding correlations between failures and metrics collected from hardware. LogRule was used to discover correlations between various memory-bank metrics and the number of UnCorrectable Errors (UCEs). The reported patterns were used to select an important set of features for memory failure prediction, which resulted in models with more accurate predictions. LogRule may also be employed in other metric-based RCA scenarios, for example to correlate poor KPI behavior (e.g., low latency) to specific state patterns (such as geographical region, host machine, and subnet) as frequently done with similar approaches [379–383].

Because all explored metrics are numeric, past approaches would have to rely on manual bucketing to discretize values, or run into a combinatorial explosion risk. To prevent this, LogRule with operators was used to handle numeric types efficiently. LogRule-op was also able to discover a single rule covering 40% failure cases thanks to the operator-based rule mining. LogRule was also applied to optical switch failure data, where it was able to associate the interface status (`up` or `down`) to specific module IDs and modes. Is it plausible to assume LogRule may identify similar relations in other hardware telemetry data, e.g., hard drive Self-Monitoring Analysis and Reporting Technology (SMART) features.

Software functions are often a target of failure analysis, where a minimal set of input parameters leading to failures shall be identified. LogRule may be used to identify such combinations of interest by analyzing input parameters of the function. In this context, the possibility to use LogRule to analyze jobs executed by an internal auto-remediation system was considered. LogRule can be applied to understand why some specific auto-remediation jobs do not terminate correctly or do not resolve the underlying issue. Each job is annotated with a completion status Y, while the job parameters and logs are used as input evidence $\mathbb{D}$.

Furthermore, LogRule may be applied to associate specific hardware/software configurations to failures. In large-scale service environments, this information is stored in a Configuration Management Database (CMDB). The CMDB information augmented with dynamic failure information enables diagnostics of various configuration failures. For example, LogRule may identify a combination of firmware version and memory chip models that are frequently associated with sudden crashes, unresponsive servers, or unsuccessful reboots [90]. Operator mining parameters and values can be correlated to the faulty behavior of a function. The operator extension of LogRule allows to represent combinations of parameters as sets, to allow robust and efficient correlation analysis.

To conclude, LogRule can discover associations between key-value pairs, therefore it may be used in databases to detect patterns such as column or key-value-pair equivalencies. These equivalencies may then be aggregated in order to compress or remove redundant data. The input parameter Y may be used as a query parameter, which enables discovery of fuzzy 'group-by' relationships in failure data.

6.5.2 Applicability to Sequential State Problems

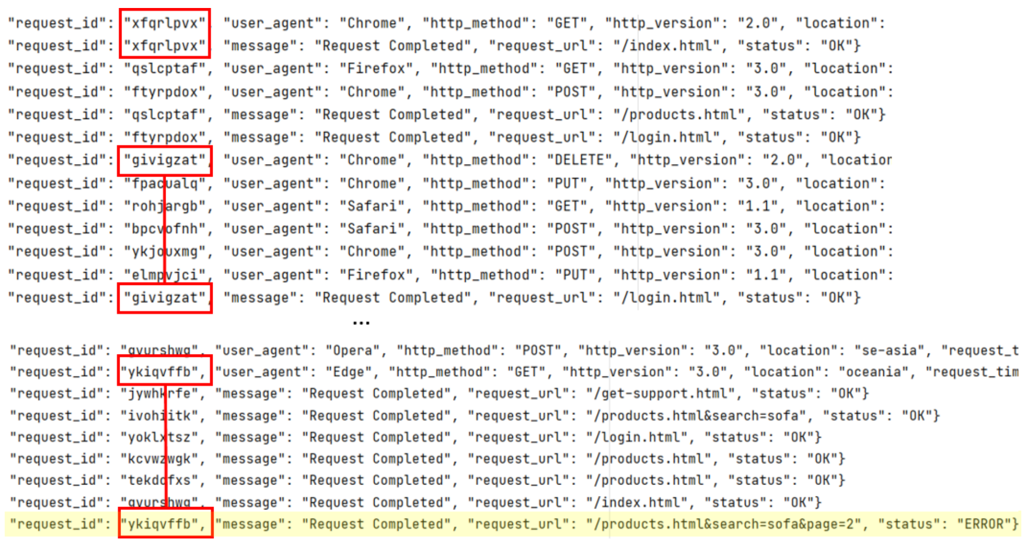
The extension of LogRule to sequential-state problems enables the application of ARM to a variety of use cases for both RCA and proactive FM. The totality of use cases discussed for single-state problems can be reformulated for sequential state problems as well, including access log analysis, OFP, function verification, etc. This subsection, therefore, focuses on additional aspects and use cases related specifically to the applicability of sequential structured logs.

Traditional system and application logs are in large majority sequential. The state of the program is represented by a series of print statements which describe the value of variables or action taken during the program execution. As such logs typically follow a log template, several fields can be extracted, such as timestamp, log severity, service identifiers, user or transaction identifiers, host name or IP address, source code file or method, and other fields present in the free text of the log line. All these fields provide information to both 1) mark specific lines as failures (e.g., using line severity) which constitutes the necessary input Y and 2) diagnose these future failures by observing log lines that preceded them. To this end, applications include Operating System (OS) level, service, and application troubleshooting [94]. The sequential approach may also be used for API or microservice debugging, for tracking the state of different handlers inside an interconnected system, which are related to each other temporally.

LogRule was tested on several sequential log files to verify its effectiveness fro troubleshooting services. Figure 6.13 shows an example of a synthetic sequential log for a request-based system. Lines associated with the same requests are correlated using request IDs and are annotated by using the request status as failure variable Y. LogRule could effectively mine common request patterns across states and recognize the combination of items leading to the terminal state Y in the last request line.

A second potential use case is runtime verification of sequential state programs. It is composed of two steps: training and inference. During training, LogRule is used as in a traditional setting to learn the causality patterns of failures. The output of the system is a set $\mathbb{X}_f$ of sequential rules $\mathbf{X}_{f,i} = \{X_{f,i1} \rightarrow X_{f,i2} \rightarrow \dots X_{f,iL}\} \rightarrow Y$. During inference, these rules are used to verify whether the existing state observations from logs may predict for a future failure. If one of the learned failure sequences starts to occur, warnings can be raised to operators, so that they can take remediation or mitigation action. Because the rules are static at verification time, efficient verification methods (e.g., based on state automata) may be applied to speed up pattern recognition and alerting. The sensitivity of the algorithm may be adjusted my controlling the level of sequence overlap (e.g., 80% or $n - 2$ steps) to raise a specific warning, in consideration of the failure lead time (Section 4.3.1).

The ability to analyze sequences of features also enables the analysis of time-series data. This may be applied for anomaly detection, OFP, and software defect prediction. The sequential LogRule approach may also be used to evaluate the failure risk of a series of CLI commands (see Section 5.4.7).



```
"request_id": "xfqrlpvx", "user_agent": "Chrome", "http_method": "GET", "http_version": "2.0", "location":
"request_id": "xfqrlpvx", "message": "Request Completed", "request_url": "/index.html", "status": "OK"}
"request_id": "qslcptaf", "user_agent": "Firefox", "http_method": "GET", "http_version": "3.0", "location":
"request_id": "ftyrpdox", "user_agent": "Chrome", "http_method": "POST", "http_version": "3.0", "location":
"request_id": "qslcptaf", "message": "Request Completed", "request_url": "/products.html", "status": "OK"}
"request_id": "ftyrpdox", "message": "Request Completed", "request_url": "/login.html", "status": "OK"}
"request_id": "givigzat", "user_agent": "Chrome", "http_method": "DELETE", "http_version": "2.0", "locatior
"request_id": "fpacualq", "user_agent": "Chrome", "http_method": "PUT", "http_version": "3.0", "location":
"request_id": "rohjargb", "user_agent": "Safari", "http_method": "GET", "http_version": "1.1", "location":
"request_id": "bpcvofnh", "user_agent": "Safari", "http_method": "POST", "http_version": "3.0", "location":
"request_id": "ykjouxmg", "user_agent": "Chrome", "http_method": "POST", "http_version": "3.0", "location":
"request_id": "elmpvjci", "user_agent": "Firefox", "http_method": "PUT", "http_version": "1.1", "location":
"request_id": "givigzat", "message": "Request Completed", "request_url": "/login.html", "status": "OK"}
                                        ...
"request_id": "gvurshwg", "user_agent": "Opera", "http_method": "POST", "http_version": "3.0", "location": "se-asia", "request_t
"request_id": "ykiqvffb", "user_agent": "Edge", "http_method": "GET", "http_version": "3.0", "location": "oceania", "request_tim
"request_id": "jywhkrfe", "message": "Request Completed", "request_url": "/get-support.html", "status": "OK"}
"request_id": "ivohiitk", "message": "Request Completed", "request_url": "/products.html&search=sofa", "status": "OK"}
"request_id": "yoklxtsz", "message": "Request Completed", "request_url": "/login.html", "status": "OK"}
"request_id": "kcvwzwgk", "message": "Request Completed", "request_url": "/products.html", "status": "OK"}
"request_id": "tekddfxs", "message": "Request Completed", "request_url": "/products.html", "status": "OK"}
"request_id": "gvurshwg", "message": "Request Completed", "request_url": "/index.html", "status": "OK"}
"request_id": "ykiqvffb", "message": "Request Completed", "request_url": "/products.html&search=sofa&page=2", "status": "ERROR"}
```

Figure 6.13 Example of synthetic sequential log for a request-based system (e.g., HTTP server). The state of a request is encoded over several lines (connected by red lines), which make it sequential state system. Moreover, multiple requests are reported within the same file over time, which makes it a dynamic invocation system. The different request steps can be associated by primary keys, such as request IDs (encircled in red boxes).

6.6 Conclusion

In this chapter, the application of AI for proactive FM at the software level was discussed.

Service monitoring and causes of software failure are introduced and techniques for handling failures at the software level were described. The problem of RCA and log correlation was introduced as a motivation for applying causal discovery techniques for proactive failure management.

In the last sections LogRule, an RCA algorithm based on ARM using structured data, was presented. LogRule introduces three novel contributions, namely item-based aggregation, disjunctive support-based grouping and operator-based rule mining, to enable timely and accurate association rule mining in high-dimensional datasets. Various experiments were conducted to quantify the effect of the proposed improvements in terms of execution time, accuracy, and result interpretability and compared the results with a state-of-the-art algorithm.

The evaluation results indicate that LogRule reduces average execution time by over 97%, while achieving a 0.921 F1-score. The results and use case studies confirm that LogRule is a valid tool for RCA on structured data, with base LogRule as the ideal solution for efficient and time-sensitive analyses and LogRule-op being most indicated for long-term and deferrable RCA tasks.

While LogRule considers only cases where single log entries are mutually independent, this assumption does not always hold. Some software logs must be sequentially processed, because the sequence represents the state of the program rather than the individual log entry. To this end, an extension of LogRule to sequential state analysis has been proposed. Different use cases for sequential state analysis, such as application troubleshooting and runtime verification, were presented. This extension provides applicability of LogRule for proactive failure management, by observing early states of a failure sequence and promptly reporting it to operators for proactive remediation.

As the current framework only concentrated on a specific subset of operators and data types, further research may extend the range of available operations and attributes, e.g., to include set operations or modulo operator for cyclic data. Future research may also investigate other O&M use scenarios for the sequential framework of the algorithm.

7 Conclusion and Future Outlook

This chapter draws the conclusions from the discussions presented in the rest of this dissertation. Section 7.1 summarizes the contributions of the previous chapters. Section 7.2 explores potential extensions to this work, in the context of proactive Failure Management (FM).

7.1 Summary

This dissertation presented a comprehensive study of techniques for proactive FM in large-scale computing environments, through the use of AI tools and especially Machine Learning (ML) methods. Through the use of systematic literature view, past AI for IT Operations (AIOps) are collected and categorized to identify blind spots and shortcomings of the current FM landscape. The problem of proactive FM is divided by cloud abstraction layers and tackled separately at each layer.

For the infrastructure layer, where hardware faults are predominantly responsible for the appearance of failures, Online Failure Prediction (OFP) techniques for handling failures of most affected components were discussed, including hard drives and DRAM memories. Novel OFP algorithms for optical transceivers have been introduced. These algorithms have been developed on the basis on in-depth statistical analysis of failure patterns, which show the importance of CRC-related metrics and physical variables collected from the Digital Diagnostic Monitoring (DDM) infrastructure. The consequent feature extraction performed to construct online failure predictors, based on traditional and neural-based ML, verifies the importance of such features to classify components as healthy or faulty. The evaluation results show the ability of such predictors to accurately ($A > 99\%, P > 73\%$) anticipate future transceivers faults for a considerable fraction ($> 50\%$) of verified future failures.

For the platform layer, where complex interdependent OS, middleware and software systems must be deployed while ensuring protection from malicious attacks and operator incidents, the importance of evaluating command-line operations and prevent undesired behavior was motivated. To this end, a Large Language Model (LLM) for the command line, based on the BERT architecture, was proposed, which is effectively applied for command risk classification, through the application of a pretraining step for command-line language and a finetuning step using a supervised command dataset. Results show how the LLM approach is the most effective method for classification ($P > 97\%, R > 91\%$) of dangerous commands and requires less supervised data (down by 10x) than its predecessors. The approach is extended for improved generalization using a documentation-based method, which extracts command description from available documentation and allows to identify previously misclassified commands. Documentation is shown to enable effective detection of misclassified commands for auditing applications. A series of potential use cases for the proposed LLM model are showcased, to highlight the high degree of transfer learning achievable through command-line pretraining.

For the software layer, where for intricate and complex service architectures sufficient reliability must be guaranteed to satisfy customer needs directly, a general framework for Root Cause Analysis (RCA) and runtime rule verification based on Association Rule Mining (ARM) is presented. It extends existing contributions in terms of performance, interpretability, and correctness, to allow diagnosis and failure prediction for a variety of Operations & Maintenance (O&M) applications on single-observation state systems, such as access logs and server configurations. Results indicate that the proposed ARM framework is efficient (up to 37x faster), accurate ($P > 93\%, R > 90\%$), and concise in producing root-cause explanations, and its rules are actionable in the context of proactive FM. The ARM framework is extended to enable analysis of sequential state programs, which further increases the applicability of the ARM framework for RCA and proactive FM.

In consideration of the importance of handling failures proactively and according to the motivations described in the introduction section, all these contributions offer techniques to deal with failures before any damage has manifested and in relation to preventive actions applicable to the corresponding scenario.

This allows to perform O&M with more efficient planning, reduced response times and limited failure impact. Because of these desirable benefits, these contributions have been implemented inside a large-scale cloud provider to reduce the workload of O&M operators.

7.2 Future Outlook

Although this dissertation presents a framework for proactive FM at all layers of a cloud infrastructure, several open problems remain active in the domain areas of AIOps and proactive FM. Future solutions may be presented to support these problems and potentially be integrated in the scheme of FM methods here proposed. The current section describes a list of potential extensions and open problems which require further contributions.

7.2.1 Virtualization-level Proactive Failure Management

The contributions to infrastructure-level proactive FM in past literature and in this dissertation (Chapter 4) have mostly focused on hardware failure prediction, because it is responsible for the large majority of failures at the infrastructure layer.

However, hypervisor and virtualization failures are still frequent and mitigation strategies for this abstraction layer are under-represented due to proximity to the hardware layer.

To this end, a Virtual Machine (VM) failure prediction algorithm may be integrated into the existing framework for online forecasting of future VM failures, which may be caused by underlying hardware faults, software exceptions, resource exhaustion, or configuration errors. Potential data sources for this task may include physical host telemetry, such as resource utilization and system call tracing logs; or guest Operating System (OS) telemetry, such as application tracing, resource usage, and system logs. The concept may be applied for predicting failures in containers, an established alternative for executing single-application workload in a resource- and time-efficient paradigm.

As the hypervisors may also be the target of failures, hypervisor OFP may be supported by additional monitoring information of the hypervisor itself, or may be supported by VM information gathered from OFP predictors, as described above. The imminent prediction of a hypervisor failure may also constitute important knowledge to estimate residing VM failures, so that the state of the hypervisor may also support VM failure prediction.

Other virtualization-level failure prevention techniques may include VM checkpointing, to restore correct and most recent VM states following hardware repairs or live migration; or a fault injection study to determine the degree of reliability of existing virtualization systems. To this end, Reinforcement Learning (RL) models may act as intelligent agents in a simulated environment, while attempting to maximize damage to the infrastructure. The results may report significant information to reduce failures and external agent attacks.

7.2.2 Additional Techniques for Failure Prevention

As discussed in the results of the systematic literature review (Chapter 3), a significant number of existing contributions for proactive FM have focused on OFP. Failure prevention offers great potential to reduce the impact of failures in areas that are not previously covered by existing work.

Canary release verification has been previously applied to predict the impact of software updates. LogRule has been applied to runtime verification to forecast future failure events thanks to causality learning. Additional verification methods may be proposed, e.g., to predict the effect of new infrastructure deployments on total workload, or estimate the impact of network re-routing via Software Defined Networking (SDN).

Integrating RCA systems into AI-based failure prevention system may endow them of additional knowledge (causality, topology, cross-feature dependencies) which may support rational design choices for new IT systems. To this end, graph-based model descriptions may support efficient fault injection, to understand how failures of directly interconnected components affect neighbors. Providing topology information, such rack and vertical position or proximity to other servers, has already shown to improve accuracy for hardware failure prediction [289]. This impact may also be translated into other O&M domains.

Proactive load testing, described in Section 6.1.2, provides methods to assess software performance under variable operating conditions, by simulating different workload scenarios. Future work may show how to effectively apply generative AI to model tail-distribution scenarios, difficult to represent or consider. AI agents may also directly operate to control scaling under such conditions, in order to reduce the impact of failures and performance degradation under stress. Finally, ML models, in particular the most recent LLMs, may be applied on source code to identify performance bottlenecks and sub-optimal code regions, based on raw source code, profiling information and runtime-collected telemetry.

7.2.3 Application of LLMs to other O&M Problems

Chapter 5 has showcased the large applicability of LLMs for different platform-level O&M tasks. These applications expand beyond the platform level and may bring large benefit to other several other tasks, e.g., solution recommendation via prompts, intelligent log analysis, anomaly detection, or automatic remediation.

Following the example of existing LLM prompt services [9], LLMs may be used to provide a prompt to O&M operators. This prompt may help operators describe their problem, investigate solutions and consider potential remediation actions, e.g., a reboot or a CLI operation. The LLM may be finetuned on such O&M topic prompts or trained via reinforcement learning.

LLMs may also perform RCA and anomaly detection from logs automatically. Given the sequential and natural language foundation of logs, they are directly applicable to LLMs as inputs. The LLM may help extract relevant log lines to debug a performance issue, or produce a short summary of the encountered problem. Such log analysis system would be highly beneficial for RCA purpose and reduce Mean Time to Repair (MTTR). Similarly, LLMs may analyze metrics, traces, or other monitoring data to detect abnormalities or undesired deviations from normal behavior. These indications may be reported to O&M operations via alarms, or may directly be connected to action-taking systems.

7.2.4 Hardware Failure Prediction for Other Components

The majority of existing contributions for hardware failure prediction have focused on storage media (hard drives, SSDs), memory and few network components (such as optical transceivers). However, a large fraction of remaining components has not seen equal research interest (see Table 4.2).

The infrastructure layer would benefit from the introduction of failure prediction algorithms for other server components, such as CPU, power supply units, or motherboards, as well as for networking equipment, such as routers, switches, load balancers, firewalls, etc. Moreover, the rapid emergence of hardware acceleration services for ML imposes an important requirement for reliability on these acceleration systems. To this end, failure prediction for GPUs and other related hardware technologies (TPU, FPGA, ...) may prove fundamental to maintain Service Level Agreeement (SLA) requirements.

A List of Authored and Co-Authored Publications Associated with this Dissertation

Journal Papers

- Paolo Notaro, Jorge Cardoso, and Michael Gerndt. 2021. A Survey of AIOps Methods for Failure Management. ACM Transactions on Intelligent Systems and Technologies (TIST) Vol. 12, Issue 6, Article 81 (November 2021). Available at: `https://doi.org/10.1145/3483424`
- Paolo Notaro, Soroush Haeri, Jorge Cardoso and Michael Gerndt. 2023. LogRule: Efficient Structured Log Mining for Root Cause Analysis. In IEEE Transactions on Network and Service Management. Available at: `https://doi.org/10.1109/TNSM.2023.3282270`

Conference Papers

- Paolo Notaro, Qiao Yu, Soroush Haeri, Jorge Cardoso and Michael Gerndt. 2023. An Optical Transceiver Reliability Study based on SFP Monitoring and Operating System (OS)-level Metric Data. In Proceedings of 2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid), Bangalore, India, 2023. Available at: `https://doi.org/10.1109/CCGrid57682.2023.00011`
- Qiao Yu, Wengui Zhang, Soroush Haeri, Paolo Notaro, Jorge Cardoso, Odej Kao. 2023. HiMFP: Hierarchical Intelligent Memory Failure Prediction for Cloud Service Reliability. In Proceedings of 53nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2023). Available at: `https://doi.org/10.1109/DSN58367.2023.00031`

Workshop Papers

- Paolo Notaro, Jorge Cardoso, and Michael Gerndt. 2020. A Systematic Mapping Study in AIOps. In AIOps 2020 Workshop, located in the International Conference of Service-Oriented Computing (ICSOC 2020), December 14–17, 2020, Part of the Lecture Notes in Computer Science book series, volume 12632, 110–123, Springer-Verlag. Available at: `https://doi.org/10.1007/978-3-030-76352-7_15`

Pre-prints and Papers under Review

- Paolo Notaro, Soroush Haeri, Jorge Cardoso, Michael Gerndt. 2023. Command-line Risk Classification using Transformer-based Neural Architectures. Submitted to the 2023 Conference on Empirical Methods in Natural Language Processing. Available at: `tiny.cc/nlp-command-line`

Others

- Paolo Notaro, Soroush Haeri, Jorge Cardoso. Apparatus and Method for Auditing Rule-based Command Risk Assessment Systems, Patent filed on 2023-04-28 at European Patent Office (EPO), pending approval.
- Paolo Notaro, Jorge Cardoso, Michael Gerndt. 2020. Systematic Mapping Study in AIOps - Technical Report. Available at: `tiny.cc/aiops-technical-report`

B Mathematical Proofs and Derivations

Expected Cost of Proactive vs. Reactive Repair

- p_{fail} is the probability of observing an infrastructure failure for one component within a predefined time frame T.
- $\mathbb{P}$ is the space of events e (tuples of failures and predictions) occurring in T. These events belong to one of the four categories TP, FP, FN, TN with their associated costs described in Section 4.1.4.
- $C_{predictor}$ is the cost resulting from repairing a device proactively, i.e., in the presence of an online failure predictor. It depends on the predictor accuracy, as well as on the occurrence and cost of failures. The accuracy of the predictor is measured by true positive rate TPR (also known as *recall*) and false positive rate (also known as *fall-out*) FPR.
- $C_{nothing}$ is the cost resulting from repairing a device reactively, i.e., after the device failure has occurred. It depends exclusively on the occurrence and cost of failures.

Computing the expected cost for the predictor scenario and by using $TPR = 1 - FNR$, $TNR = 1 - FPR$, we obtain

$$\begin{aligned}
\mathbb{E}_{\mathbb{P}}[C_{predictor}] &= \sum_{e\in\mathbb{P}} C_{predictor}P(e) \\
&= c_r \cdot P(TP) + c_r \cdot P(FP) + c_f \cdot P(FN) + \cancel{0 \cdot P(TN)} \\
&= c_r \cdot P(+|F)P(F) + c_r \cdot P(+|\neg F)P(\neg F) + c_f \cdot P(-|F)P(F) \\
&= c_r \cdot TPR \cdot p_{fail} + c_r \cdot FPR \cdot (1 - p_{fail}) + c_f \cdot FNR \cdot p_{fail} \\
&= c_r \cdot R \cdot p_{fail} + c_r \cdot FPR\dot{(}1 - p_{fail}) + c_f \cdot (1 - TPR) \cdot p_{fail}
\end{aligned}$$

Computing the expected cost for the traditional scenario, we obtain

$$\begin{aligned}
\mathbb{E}_{\mathbb{P}}[C_{nothing}] &= \sum_{e\in\mathbb{P}} C_{nothing}P(e) \\
&= c_f \cdot P(TP) + \cancel{0 \cdot P(FP)} + c_f \cdot P(FN) + \cancel{0 \cdot P(TN)} \\
&= c_f \cdot P(+|F)P(F) + c_f \cdot P(-|F)P(F) \\
&= c_f \cdot P(F) \\
&= c_f \cdot p_{fail}
\end{aligned}$$

By comparing the two:

$$
\begin{aligned}
\mathbb{E}_{\mathbb{P}}[C_{predictor}] &< \mathbb{E}_{\mathbb{P}}[C_{nothing}] \\
c_r \cdot TPR \cdot p_{fail} + c_r \cdot FPR\dot{(}1 - p_{fail}) + c_f \cdot (1 - TPR) \cdot p_{fail} &< c_f \cdot p_{fail} \\
c_r \cdot TPR \cdot p_{fail} + c_r \cdot FPR\dot{(}1 - p_{fail}) &< c_f \cdot p_{fail} - c_f \cdot (1 - TPR) \cdot p_{fail} \\
c_r \cdot TPR \cdot p_{fail} + c_r \cdot FPR\dot{(}1 - p_{fail}) &< c_f \cdot p_{fail} \cdot (\cancel{1 - 1} + TPR) \\
\cancel{TPR \cdot p_{fail}} \cdot c_r \left(1 + \frac{(1 - p_{fail}) \cdot FPR}{p_{fail} \cdot TPR}\right) &< c_f \cdot \cancel{p_{fail} \cdot TPR} \\
c_r \cdot \left(1 + \frac{(1 - p_{fail}) \cdot FPR}{p_{fail} \cdot TPR}\right) &< c_f
\end{aligned}
$$

Proof of Correctness of Mining only Y Transactions

In Section 6.4.2, it was claimed that it is not necessary to mine patterns on all transactions t_i of the database $\mathbb{D}$. It is in fact sufficient to apply the pattern mining step only on the transactions where the itemset Y is present, which we will define as $\mathbb{D}_Y = \{Y \subseteq t \mid \forall t \in \mathbb{D}\}$. This does not mean all mined patterns should contain Y (which were actually excluded as they are trivial solutions to the problem), but rather that these mined patterns should only be composed of items seen alongside Y. Patterns that do not meet this condition can be ignored, as they never appear with Y and cannot positive correlate with its appearance. In particular, all these patterns will have confidence, conditional support, and lift equal to zero, which renders them unapplicable for Root Cause Analysis (RCA).

This result is shown as follows. We want to show $X \nsubseteq \mathbb{D}_Y \implies \mathcal{S}(X \mid Y) = \mathcal{C}(X, Y) = \mathcal{L}(X, Y) = 0$.

Conditional support, confidence, and lift are all proportional to $\mathcal{S}(X \cup Y)$:

$$
\begin{aligned}
\mathcal{S}(X \mid Y) &= \frac{\mathcal{S}(X \cup Y)}{\mathcal{S}(Y)} \propto \mathcal{S}(X \cup Y) \\
\mathcal{C}(X, Y) &= \frac{\mathcal{S}(X \cup Y)}{\mathcal{S}(X)} \propto \mathcal{S}(X \cup Y) \\
\mathcal{L}(X, Y) &= \frac{\mathcal{C}(X, Y)}{\mathcal{S}(Y)} = \frac{\mathcal{S}(X \cup Y)}{\mathcal{S}(X)\mathcal{S}(Y)} \propto \mathcal{S}(X \cup Y)
\end{aligned}
$$

and, by definition, $\mathcal{S}(X \cup Y)$ is the ratio of transactions containing $(X \cup Y)$:

$$
\mathcal{S}(X \cup Y) = \frac{|\{X, Y \subseteq t \mid \forall t \in \mathbb{D}\}|}{|\mathbb{D}|}
$$

Therefore, we can reduce the problem to proving $X \nsubseteq \mathbb{D}_Y \implies |\{X, Y \subseteq t \mid \forall t \in \mathbb{D}\}| = 0$. This is however a trivial result, given $|\emptyset| = 0$:

$$
\begin{aligned}
X \nsubseteq \mathbb{D}_Y &= X \nsubseteq \{Y \subseteq t \mid \forall t \in \mathbb{D}\} \\
\implies \{X, Y \subseteq t \mid \forall t \in \mathbb{D}\} &= \emptyset \\
\implies |\{X, Y \subseteq t \mid \forall t \in \mathbb{D}\}| &= 0 \\
\implies \mathcal{S}(X \cup Y) &= 0 \\
\implies \mathcal{S}(X \mid Y) = \mathcal{C}(X, Y) = \mathcal{L}(X, Y) &= 0
\end{aligned}
$$

List of Figures

List of Tables

List of Algorithms

Index

Bold indices indicate definition of a concept, normal-text indices indicate a reference.

Acronyms

AFR Annualized Failure Rate

AIOps AI for IT Operations

API Application Programming Interface

ARIMA AutoRegressive Integrated Moving Average

ARM Association Rule Mining

AST Abstract Syntax Tree

AUC Area under the Curve

AUCROC Area under the Curve of Receiving Operating Characteristic

AWS Amazon Web Services

AYU As-Yet-Unconsumed

BERT Bidirectional Encoders Representations from Transformers

BI Business Intelligence

BoW Bag-Of-Words

BPE Byte-Pair Encoding

CE Correctable Error

CLI Command-Line Interface

CMDB Configuration Management Database

CNN Convolutional Neural Network

CRC Cyclic Redundancy Check

CV Computer Vision

DDM Digital Diagnostic Monitoring

DDoS Distributed Denial of Service

DES Double Exponential Smoothing

DIMM Dual In-line Memory Module

DRAM Dynamic Random Access Memory

ECC Error Correction Code

EDAC Error Detection And Correction

EEPROM Electrically Erasable Programmable Read-only Memory

ER Error Rate

ESD Electrostatic Discharge

FaaS Function-as-a-Service

FM Failure Management

FN False Negative

FNR False Negative Rate

FP False Positive

FPR False Positive Rate

GBDT Gradient Boosted Decision Tree

GELU Gaussian Error Linear Unit

GPT	Generative Pre-training Transformer
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
GSP	Generalized Sequence Patterns
HDD	Hard Disk Drive
HMM	Hidden Markov Model
HPC	High Performance Computing
HSMM	Hidden Semi-Markov Model
IaaS	Infrastructure-as-a-Service
IMC	Integrated Memory Controller
IoT	Internet of Things
ITC	Internet Traffic Classification
JIT	Just-In-Time
JRE	Java Runtime Environment
KDE	Kernel Density Estimation
kNN	*k*-Nearest Neighbors
KPI	Key Performance Indicator
LAN	Local Area Network
LLM	Large Language Model
LR	Logistic Regression
LSTM	Long Short-Term Memory
MCE	Machine Check Exception
MI	Mutual Information
MIB	Management Information Base
ML	Machine Learning
MLP	MultiLayer Perceptron
MSE	Mean Squared Error
MTBF	Mean Time Between Failures
MTTD	Mean Time to Detect
MTTR	Mean Time to Repair
NER	Named-Entity Recognition
NIST	National Institute of Standards and Technology
NLP	Natural Language Processing
NLTK	Natural Language ToolKit
O&M	Operations & Maintenance
OAM	Operations, Administration, and Maintenance
OFP	Online Failure Prediction
OID	Object Identifier
OPEX	Operating Expense
OS	Operating System
PaaS	Platform-as-a-Service
PCFG	Probabilistic Context-Free Grammar
PDF	Probability Density Function

PDF Cumulative Density Function
PICO Population, Intervention, Comparison Outcome
PoS Part-of-Speech
PPV Positive Predictive Value
PrefixSpan Prefix-projected Sequential PAttern Mining

QoS Quality of Service

RAID Redundant Array of Inexpensive Disks
RCA Root Cause Analysis
ReLU Rectified Linear Unit
RF Random Forest
RL Reinforcement Learning
RLHF Reinforcement Learning from Human Feedback
RNN Recurrent Neural Network
ROC Receiving Operating Characteristic
RUL Remaining Useful Life

SaaS Software-as-a-Service
SAS Serial Attached SCSI
SATA Serial AT Attachment
SCSI Small Computer System Interface
SDN Software Defined Networking
SDP Software Defect Prediction
SFL Software Fault Localization
SFP Small-Form Pluggable
SGD Stochastic Gradient Descent
SLA Service Level Agreeement
SMART Self-Monitoring Analysis and Reporting Technology
SMS Systematic Mapping Study
SNMP Simple Networking Management Protocol
SOP Standard Operating Procedure
SPADE Sequential PAttern Discovery using Equivalence classes
SPM Sequential Pattern Mining
SQL Structured Query Language
SSD Solid State Drive
SVM Support Vector Machine

TAN Tree-Augmented Network
TF-IDF Term Frequency-Inverse Document Frequency
TN True Negative
TNR True Negative Rate
ToR Top-of-Rack
TP True Positive
TPR True Positive Rate
TPU Tensor Processing Unit

UCE UnCorrectable Error
URL Uniform Resource Locator

VM Virtual Machine
VMM Virtual Machine Manager
VPN Virtual Private Network

WAN Wide Area Network
WHM Weighted Harmonic Mean

Bibliography

[1] Eurostat, "Cloud computing - statistics on the use by enterprises." [Online]. Available: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Cloud_computing_-_statistics_on_the_use_by_enterprises

[2] Forrester, "The State Of Cloud In North America, 2022: Modernization And Cloud Native Will Be The New Normal." [Online]. Available: https://www.forrester.com/press-newsroom/the-state-of-cloud-in-north-america-2022/

[3] S. Russell and P. Norvig, "Artificial Intelligence: a Modern Approach," *Prentice Hall Upper Saddle River, NJ, USA: Rani, M., Nayak, R., & Vyas, OP (2015). An ontology-based adaptive personalized e-learning system, assisted by software agents on cloud storage. Knowledge-Based Systems*, vol. 90, 2002.

[4] A. M. Turing, "I.—COMPUTING MACHINERY AND INTELLIGENCE," *Mind*, vol. LIX, no. 236, pp. 433–460, 10 1950. [Online]. Available: https://doi.org/10.1093/mind/LIX.236.433

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[6] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[8] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[9] OpenAI, "GPT-4 Technical Report," 2023. [Online]. Available: https://arxiv.org/abs/2303.08774

[10] P. P. Shinde and S. Shah, "A review of machine learning and deep learning applications," in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, 2018.

[11] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 193–204. [Online]. Available: http://dx.doi.org/10.1145/1807128.1807161

[12] L. A. Barroso, J. Clidaras, and U. Hölzle, *The datacenter as a computer: An introduction to the design of warehouse-scale machines.* Morgan & Claypool Publishers, 2013, vol. 8, no. 3.

[13] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, “Recovery oriented computing (roc): Motivation, definition, techniques, and case studies,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-02-1175, Mar 2002. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2002/5574.html

[14] A. Lerner, “AIOps Platforms - Gartner,” 8 2017. [Online]. Available: https://blogs.gartner.com/andrew-lerner/2017/08/09/aiops-platforms/

[15] 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).* USA: IEEE, 5 2019, pp. 4–5. [Online]. Available: http://dx.doi.org/10.1109/ICSE-Companion.2019.00023

[16] A. Levin, S. Garion, E. K. Kolodner, D. H. Lorenz, K. Barabash, M. Kugler, and N. McShane, “AIOps for a Cloud Object Storage Service,” in *2019 IEEE International Congress on Big Data (BigDataCongress).* USA: IEEE, 7 2019, pp. 165–169. [Online]. Available: http://dx.doi.org/10.1109/bigdatacongress.2019.00036

[17] Moogsoft, “An Everything Guide to AIOps,” 2020, accessed on: 2023-8-16. [Online]. Available: https://www.moogsoft.com/resources/aiops/guide/everything-aiops/

[18] Broadcom, “What is AIOps?” Broadcom, 2020, accessed on: 2023-08-16. [Online]. Available: https://www.broadcom.com/products/software/aiops

[19] OpsRamp, “AIOps (AI for IT Operations) - OpsRamp,” 2020. [Online]. Available: https://www.opsramp.com/solutions/service-centric-aiops/

[20] BMC Software, “AIOps - Artificial Intelligence for IT Operations,” 2020, accessed on: 2023-08-16. [Online]. Available: https://www.bmc.com/it-solutions/aiops.html

[21] Resolve Systems, “What is AIOps? - Resolve,” 2020, accessed on July 15, 2021. [Online]. Available: https://resolve.io/what-is-aiops

[22] X. Chen, C.-D. Lu, and K. Pattabiraman, “Failure analysis of jobs in compute clouds: A google cluster case study,” in *2014 IEEE 25th International Symposium on Software Reliability Engineering.* USA: IEEE, 11 2014, pp. 167–177. [Online]. Available: http://dx.doi.org/10.1109/issre.2014.34

[23] J. Xiao, Z. Xiong, S. Wu, Y. Yi, H. Jin, and K. Hu, “Disk failure prediction in data centers via online learning,” in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA: Association for Computing Machinery, 8 2018. [Online]. Available: http://dx.doi.org/10.1145/3225058.3225106

[24] Y. Li, Z. M. J. 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*, vol. 29, no. 2, pp. 13:1–13:24, 4 2020.

[25] S. Nedelkoski, J. Cardoso, and O. Kao, “Anomaly detection and classification using distributed tracing and deep learning,” in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID).* USA: IEEE, 5 2019, pp. 241–250. [Online]. Available: http://dx.doi.org/10.1109/ccgrid.2019.00038

[26] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, “Detecting large-scale system problems by mining console logs,” in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles - SOSP ’09*, ser. SOSP ’09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 117–132. [Online]. Available: http://dx.doi.org/10.1145/1629575.1629587

[27] 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*. Dallas, Texas, USA: ACM, 10 2017, pp. 1285–1298. [Online]. Available: http://dx.doi.org/10.1145/3133956.3134015

[28] A. Brown, A. Tuor, B. Hutchinson, and N. Nichols, "Recurrent neural network attention mechanisms for interpretable system log anomaly detection," in *Proceedings of the First Workshop on Machine Learning for Computing Systems*, ser. MLCS'18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3217871.3217872

[29] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, 1 2007.

[30] J. Li, X. Ji, Y. Jia, B. Zhu, G. Wang, Z. Li, and X. Liu, "Hard drive failure prediction using classification and regression trees," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. USA: IEEE, 6 2014, pp. 383–394. [Online]. Available: http://dx.doi.org/10.1109/dsn.2014.44

[31] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: problem determination in large, dynamic internet services," in *Proceedings International Conference on Dependable Systems and Networks*, 2002, pp. 595–604.

[32] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: Automated classification of performance crises," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 111–124. [Online]. Available: https://doi.org/10.1145/1755913.1755926

[33] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "SherLog: error diagnosis by connecting clues from run-time logs," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 143–154, Mar. 2010. [Online]. Available: http://dx.doi.org/10.1145/1735970.1736038

[34] N. Bobroff, A. Kochut, and K. Beaty, "Dynamic Placement of Virtual Machines for Managing SLA Violations," in *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*. IEEE, May 2007. [Online]. Available: http://dx.doi.org/10.1109/inm.2007.374776

[35] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant, "Automated control of multiple virtualized resources," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 13–26. [Online]. Available: https://doi.org/10.1145/1519065.1519068

[36] E. Barrett, E. Howley, and J. Duggan, "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 12, pp. 1656–1674, 2013. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.2864

[37] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," *SIGPLAN Not.*, vol. 48, no. 4, p. 77–88, 2013. [Online]. Available: https://doi.org/10.1145/2499368.2451125

[38] P. Notaro, J. Cardoso, and M. Gerndt, "A Systematic Mapping Study in AIOps," in *International Conference on Service-Oriented Computing - Workshop on Artificial Intelligence for IT Operations (AIOps)*. Springer, 2020, pp. 110–123.

[39] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," National Institute of Standards and Technology, Tech. Rep. NIST Special Publication (SP) 800-145, Sep. 2011. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-145/final

[40] R. Hill, L. Hirsch, P. Lake, and S. Moshiri, *Introducing Cloud Computing*. London: Springer London, 2013, pp. 3–19. [Online]. Available: https://doi.org/10.1007/978-1-4471-4603-2_1

[41] X. Zhou, H. Liu, and R. Urata, "Datacenter Optics: Requirements, Technologies, and Trends," *Chinese Optics Letters*, vol. 15, no. 5, May 2017. [Online]. Available: https://research.google/pubs/pub45992

[42] K. Bilal, S. U. R. Malik, S. U. Khan, and A. Y. Zomaya, "Trends and challenges in cloud datacenters," *IEEE Cloud Computing*, vol. 1, no. 1, pp. 10–20, 2014.

[43] C. Kachris and I. Tomkos, "Optical interconnection networks for data centers," in *2013 17th International Conference on Optical Networking Design and Modeling (ONDM)*, 01 2013, pp. 19–22. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6524909

[44] J. Sahoo, S. Mohapatra, and R. Lath, "Virtualization: A survey on concepts, taxonomy and associated security issues," in *2010 Second International Conference on Computer and Network Technology*, 2010, pp. 222–226. [Online]. Available: https://ieeexplore.ieee.org/document/5474503

[45] H. Chirammal, P. Mukhedkar, and A. Vettathu, *Mastering KVM virtualization.* Packt Publishing, 2016.

[46] R. P. Goldberg *et al.*, "Architectural principles for virtual computer systems," Ph.D. dissertation, Harward University, 1973.

[47] Microsoft, "Introduction to hyper-v on windows 10," https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/, accessed on 2023-08-09.

[48] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 164–177. [Online]. Available: https://doi.org/10.1145/945445.945462

[49] VirtualBox, "Virtualbox," https://www.virtualbox.org, accessed on 2023-08-09.

[50] Parallels, "Parallels," https://www.parallels.com, accessed on 2023-08-09.

[51] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[52] Containerd, "containerd - an industry-standard container runtime with an emphasis on simplicity, robustness and portability," https://www.containerd.io, accessed on 2023-08-09.

[53] "Kubernetes - Overview," https://kubernetes.io/docs/concepts/overview/, accessed on: 2023-08-18.

[54] L. Qian, Z. Luo, Y. Du, and L. Guo, "Cloud computing: An overview," in *Cloud Computing*, M. G. Jaatun, G. Zhao, and C. Rong, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 626–631.

[55] K. Jamsa, *Cloud computing.* Jones & Bartlett Learning, 2022.

[56] M. Kavis, *Cloud Service Models.* John Wiley & Sons, Ltd, 2014, ch. 2, pp. 13–22. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118691779.ch2

[57] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "Openstack: toward an open-source solution for cloud computing," *International Journal of Computer Applications*, vol. 55, no. 3, pp. 38–42, 2012.

[58] Microsoft, "Microsoft Azure," https://www.azure.microsoft.com, accessed on 2023-08-09.

[59] "Cloud services models - packtpub," https://web.archive.org/web/20230803150818/https://subscription.packtpub.com/book/cloud-and-networking/9781789134964/1/ch01lvl1sec03/cloud-services-models, accessed: 2023-08-03.

[60] Red Hat, "Types of cloud computing," https://www.redhat.com/en/topics/cloud-computing/public-cloud-vs-private-cloud-and-hybrid-cloud, 2022, accessed on 2023-08-09.

[61] Google, "Google Cloud Platform (GCP)," http://cloud.google.com, accessed on 2023-08-09.

[62] "Heroku," https://www.heroku.com/.

[63] Cloud Native Computing Foundation (CNCF), "CNCF WG-Serverless Whitepaper v1.0," https://gw.alipayobjects.com/os/basement_prod/24ec4498-71d4-4a60-b785-fa530456c65b.pdf, accessed on: 2023-08-24.

[64] Red Hat, "What is Function-as-a-Service (FaaS)?" https://www.redhat.com/en/topics/cloud-native-apps/what-is-faas, 2020, accessed on 2023-08-09.

[65] Amazon, "Amazon Web Services (AWS)," http://aws.amazon.com, accessed on 2023-08-09.

[66] IBM, "IBM Cloud," https://www.ibm.com/cloud, accessed on 2023-08-09.

[67] Cisco, "Cisco Cloud Solutions," https://www.cisco.com/c/en/us/solutions/cloud/index.html, 2022, accessed on 2023-08-09.

[68] Spot by NetApp, "Cloud cost: 4 cost models and 6 cost management strategies," https://spot.io/resources/cloud-cost/, 2023, accessed on 2023-08-09.

[69] D. Serrano, S. Bouchenak, Y. Kouki, F. A. de Oliveira Jr., T. Ledoux, J. Lejeune, J. Sopena, L. Arantes, and P. Sens, "Sla guarantees for cloud services," *Future Generation Computer Systems*, vol. 54, pp. 233–246, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X15000801

[70] S. Bhowmik, *Cloud computing.* Cambridge University Press, 2017.

[71] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based failure and evolution management," in *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, ser. NSDI'04. San Francisco, California: USENIX Association, 3 2004, p. 23. [Online]. Available: https://dl.acm.org/doi/10.5555/1251175.1251198

[72] T. Mizrahi, N. Sprecher, E. Bellagamba, and Y. Weingarten, "An Overview of Operations, Administration, and Maintenance (OAM) Tools," https://datatracker.ietf.org/doc/html/rfc7276, Jun. 2014, accessed on: 2023-08-16. [Online]. Available: https://www.rfc-editor.org/info/rfc7276

[73] F. Salfner, M. Lenk, and M. Malek, "A survey of online failure prediction methods," *ACM Computing Surveys*, vol. 42, no. 3, pp. 1–42, 3 2010.

[74] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site reliability engineering: How Google runs production systems.* " O'Reilly Media, Inc.", 2016.

[75] P. Notaro, J. Cardoso, and M. Gerndt, "A Survey of AIOps Methods for Failure Management," *ACM Trans. Intell. Syst. Technol.*, vol. 12, no. 6, nov 2021. [Online]. Available: https://doi.org/10.1145/3483424

[76] G. Steele, *Common LISP: the language.* Elsevier, 1990.

[77] P. Deransart, A. Ed-Dbali, and L. Cervoni, *Prolog: the standard: reference manual.* Springer Science & Business Media, 2012.

[78] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32.* Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[79] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[80] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.", 2009.

[81] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[82] C. M. Bishop and N. M. Nasrabadi, *Pattern Recognition and Machine Learning.* Springer, 2006.

[83] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[84] A. Casari and A. Zheng, "Feature engineering for machine learning," *O'Reilly Media, Inc.*, p. 218, 2018.

[85] S.-i. Amari, "Backpropagation and stochastic gradient descent method," *Neurocomputing*, vol. 5, no. 4-5, pp. 185–196, 1993.

[86] Encyclopedia Britannica, "Data mining - pattern mining." [Online]. Available: https://www.britannica.com/technology/data-mining/Pattern-mining#ref1073343

[87] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," *SIGMOD Rec.*, vol. 22, no. 2, p. 207–216, Jun. 1993. [Online]. Available: https://doi.org/10.1145/170036.170072

[88] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, "Dynamic itemset counting and implication rules for market basket data," in *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 255–264. [Online]. Available: https://doi.org/10.1145/253260.253325

[89] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *SIGMOD Rec.*, vol. 29, no. 2, p. 1–12, May 2000. [Online]. Available: https://doi.org/10.1145/335191.335372

[90] F. Lin, K. Muzumdar, N. P. Laptev, M.-V. Curelea, S. Lee, and S. Sankar, "Fast dimensional analysis for root cause investigation in a large-scale service environment," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 2, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3392149

[91] R. Srikant and R. Agrawal, "Mining sequential patterns: Generalizations and performance improvements," in *International conference on extending database technology.* Springer, 1996, pp. 1–17.

[92] J. Han, J. Pei, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu, "Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth," in *proceedings of the 17th international conference on data engineering.* IEEE, 2001, pp. 215–224.

[93] D.-Y. Chiu, Y.-H. Wu, and A. L. Chen, "An efficient algorithm for mining frequent sequences by a new strategy without support counting," in *Proceedings. 20th International Conference on Data Engineering.* IEEE, 2004, pp. 375–386.

[94] V. Murali, E. Yao, U. Mathur, and S. Chandra, “Scalable statistical root cause analysis on app telemetry,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).* USA: IEEE, 2021, pp. 288–297.

[95] L. Breiman, “Random Forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[96] H. N. Koivo, “Neural networks: Basics using matlab neural network toolbox,” *Author Website*, 2008.

[97] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[98] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[99] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” 2017.

[100] A. F. Agarap, “Deep learning using rectified linear units (relu),” *CoRR*, vol. abs/1803.08375, 2018. [Online]. Available: http://arxiv.org/abs/1803.08375

[101] D. Hendrycks and K. Gimpel, “Gaussian error linear units (gelus),” *arXiv preprint arXiv:1606.08415*, 2016.

[102] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Improving neural networks by preventing co-adaptation of feature detectors,” *CoRR*, vol. abs/1207.0580, 2012. [Online]. Available: http://arxiv.org/abs/1207.0580

[103] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[104] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[105] Z. Wang, W. Yan, and T. Oates, “Time series classification from scratch with deep neural networks: A strong baseline,” *CoRR*, vol. abs/1611.06455, 2016. [Online]. Available: http://arxiv.org/abs/1611.06455

[106] C. Olah, “Understanding lstm networks,” https://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015, accessed: 2023-08-08.

[107] Y. Bengio, P. Simard, and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, March 1994.

[108] H. I. Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P. Muller, “Deep learning for time series classification: a review,” *CoRR*, vol. abs/1809.04356, 2018. [Online]. Available: http://arxiv.org/abs/1809.04356

[109] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” 2014. [Online]. Available: https://arxiv.org/pdf/1409.0473.pdf

[110] Jun Zhang and K. F. Man, “Time series prediction using rnn in multi-dimension embedding phase space,” in *SMC’98 Conference Proceedings. 1998 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No.98CH36218)*, vol. 2, Oct 1998, pp. 1868–1873 vol.2.

[111] J. Devlin, M. Chang, K. Lee, and K. Toutanova, “BERT: pre-training of deep bidirectional transformers for language understanding,” *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[112] D. Jurafsky and J. H. Martin, “Speech and language processing (3rd ed. draft),” https://web.stanford.edu/~jurafsky/slp3/, 2023, accessed on 2023-08-11.

[113] R. Anand and U. Jeffrey David, *Mining of massive datasets.* Cambridge university press, 2011.

[114] K. W. Church, "Word2vec," *Natural Language Engineering*, vol. 23, no. 1, pp. 155–162, 2017.

[115] M. V. Koroteev, "BERT: A review of applications in natural language processing and understanding," *CoRR*, vol. abs/2103.11943, 2021. [Online]. Available: https://arxiv.org/abs/2103.11943

[116] C. Sun, X. Qiu, Y. Xu, and X. Huang, "How to fine-tune bert for text classification?" in *Chinese Computational Linguistics*, M. Sun, X. Huang, H. Ji, Z. Liu, and Y. Liu, Eds. Cham: Springer International Publishing, 2019, pp. 194–206.

[117] F. Lin, M. Beadon, H. D. Dixit, G. Vunnam, A. Desai, and S. Sankar, "Hardware remediation at scale," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W).* USA: IEEE, 6 2018, pp. 14–17. [Online]. Available: http://dx.doi.org/10.1109/dsn-w.2018.00015

[118] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 12 1976.

[119] M. H. Halstead, *Elements of Software Science (Operating and programming systems series).* USA: Elsevier Science Inc., 1977.

[120] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 6 1994.

[121] L. Briand, J. Daly, and J. Wust, "A unified framework for coupling measurement in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 25, no. 1, pp. 91–121, 1 1999.

[122] L. Burnell and E. Horvitz, "Structure and chance: melding logic and probability for software debugging," *Communications of the ACM*, vol. 38, no. 3, pp. 31–ff., 3 1995.

[123] T. M. Khoshgoftaar and D. L. Lanning, "A neural network approach for early detection of program modules having high risk in the maintenance phase," *Journal of Systems and Software*, vol. 29, no. 1, pp. 85–91, 4 1995.

[124] A. Csenki, "Bayes predictive analysis of a fundamental software reliability model," *IEEE Transactions on Reliability*, vol. 39, no. 2, pp. 177–183, 6 1990.

[125] N. Karunanithi, D. Whitley, and Y. Malaiya, "Prediction of software reliability using connectionist models," *IEEE Transactions on Software Engineering*, vol. 18, no. 7, pp. 563–574, 7 1992.

[126] J. Hellerstein, F. Zhang, and P. Shahabuddin, "An approach to predictive detection for service management," in *Integrated Network Management VI. Distributed Management for the Networked Millennium. Proceedings of the Sixth IFIP/IEEE International Symposium on Integrated Network Management. (Cat. No.99EX302).* USA: IEEE, 1999, pp. 309–322. [Online]. Available: http://dx.doi.org/10.1109/inm.1999.770691

[127] D. Tang and R. Iyer, "Dependability measurement and modeling of a multicomputer system," *IEEE Transactions on Computers*, vol. 42, no. 1, pp. 62–75, 1993.

[128] G. Hughes, J. Murray, K. Kreutz-Delgado, and C. Elkan, "Improved disk-drive failure warnings," *IEEE Transactions on Reliability*, vol. 51, no. 3, pp. 350–357, 9 2002.

[129] J. F. Murray, G. F. Hughes, and K. Kreutz-Delgado, "Hard Drive Failure Prediction using Non-parametric Statistical Methods," 1 2003. [Online]. Available: http://dsp.ucsd.edu/~jfmurray/publications/Murray2003.pdf

[130] ——, “Machine learning methods for predicting failures in hard drives: A multiple-instance application,” *The Journal of Machine Learning Research*, vol. 6, pp. 783–816, 12 2005.

[131] S. Garg, A. Puliafito, M. Telek, and K. Trivedi, “Analysis of software rejuvenation using markov regenerative stochastic petri net,” in *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE’95.* Toulouse, France: IEEE Comput. Soc. Press, 1995, pp. 180–187. [Online]. Available: http://dx.doi.org/10.1109/issre.1995.497656

[132] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, “Software rejuvenation: analysis, module and applications,” in *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers.* IEEE, 6 1995, pp. 381–390.

[133] S. Garg, A. van Moorsel, K. Vaidyanathan, and K. Trivedi, “A methodology for detection and estimation of software aging,” in *Proceedings Ninth International Symposium on Software Reliability Engineering (Cat. No.98TB100257).* Paderborn, Germany: IEEE Comput. Soc, 1998, pp. 283–292. [Online]. Available: http://dx.doi.org/10.1109/issre.1998.730892

[134] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, “Fault injection for dependability validation: a methodology and some applications,” *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 166–182, 2 1990.

[135] S. Han, K. Shin, and H. Rosenberg, “Doctor: an integrated software fault injection environment for distributed real-time systems,” in *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*, Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium. USA: IEEE, 4 1995, pp. 204–213.

[136] T. K. Tsai and R. K. Iyer, “Ftape: A fault injection tool to measure fault tolerance,” NASA STI/Recon Technical Report N, 10th Computing in Aerospace Conference, 1995, p. 25333, 1 1995. [Online]. Available: https://experts.illinois.edu/en/publications/ftape-a-fault-injection-tool-to-measure-fault-tolerance

[137] K. Vaidyanathan and K. Trivedi, “A measurement-based model for estimation of resource exhaustion in operational software systems,” in *Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No.PR00443).* USA: IEEE, 1999, pp. 84–93. [Online]. Available: http://dx.doi.org/10.1109/issre.1999.809313

[138] A. Bouloutas, S. Calo, and A. Finkel, “Alarm correlation and fault identification in communication networks,” *IEEE Transactions on Communications*, vol. 42, no. 2/3/4, pp. 523–533, 2 1994.

[139] S. Kliger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo, *A Coding Approach to Event Correlation.* Boston, MA: Springer US, 1995, pp. 266–277. [Online]. Available: https://doi.org/10.1007/978-0-387-34890-2_24

[140] A. Aghasaryan, E. Fabre, A. Benveniste, R. Boubour, and C. Jard, “Fault detection and diagnosis in distributed systems: An approach by partially stochastic petri nets,” *Discrete Event Dynamic Systems*, vol. 8, no. 2, pp. 203–231, 1998.

[141] P. A. Dinda and D. R. O’Hallaron, “An evaluation of linear models for host load prediction,” in *Proceedings of The Eighth International Symposium on High Performance Distributed Computing (Cat. No.99TH8469).* USA: IEEE, 1999, pp. 87–96.

[142] A. Ward, P. Glynn, and K. Richardson, “Internet service performance failure detection,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 3, pp. 38–43, 12 1998.

[143] K. Butler and J. Momoh, “A neural net based approach for fault diagnosis in distribution networks,” in *IEEE Power Engineering Society. 1999 Winter Meeting (Cat. No.99CH36233)*, vol. 1. USA: IEEE, 1999, pp. 353–356 vol.1.

[144] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys*, vol. 41, no. 3, pp. 15:1–15:58, 7 2009.

[145] "AIOPS 2021 - Second International Workshop on Artificial Intelligence for IT Operations," https://aiops2021.github.io/, accessed on 2023-08-09.

[146] "The ICSE'23 Workshop on Cloud Intelligence / AIOps," https://cloudintelligenceworkshop.org/CFP.html, accessed on 2023-08-09.

[147] K. A. H. Kobbacy, S. Vadera, and M. H. Rasmy, "Ai and or in management of operations: history and trends," *Journal of the Operational Research Society*, vol. 58, no. 1, pp. 10–28, 1 2007.

[148] K. A. Kobbacy and S. Vadera, "A survey of ai in operations management from 2005 to 2009," *Journal of Manufacturing Technology Management*, vol. 22, no. 6, pp. 706–733, 7 2011.

[149] C. B. L. Neto, P. B. D. C. Filho, and A. N. Duarte, "A systematic mapping study on fault management in cloud computing," in *2013 International Conference on Parallel and Distributed Computing, Applications and Technologies.* USA: IEEE, 12 2013, pp. 332–337. [Online]. Available: https://ieeexplore.ieee.org/document/6904276

[150] M. A. Mukwevho and T. Celik, "Toward a smart cloud: A review of fault-tolerance methods in cloud systems," *IEEE Transactions on Services Computing*, vol. 14, no. 2, pp. 589–605, 2021.

[151] R. Chalapathy and S. Chawla, "Deep learning for anomaly detection: A survey," p. 50, Jan. 2019. [Online]. Available: http://arxiv.org/abs/1901.03407

[152] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys*, vol. 43, no. 2, pp. 11:1–11:29, 2 2011.

[153] B. Dhanalaxmi, G. A. Naidu, and K. Anuradha, "A review on software fault detection and prevention mechanism in software development activities," *Journal of Computer Engineering*, vol. 17, no. 6, pp. 25–30, 2015.

[154] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Computing Surveys*, vol. 48, no. 3, pp. 44:1–44:55, 2 2016.

[155] Meiliana, S. Karim, H. L. H. S. Warnars, F. L. Gaol, E. Abdurachman, and B. Soewito, "Software metrics for fault prediction using machine learning approaches: A literature review with promise repository dataset," in *2017 IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom).* USA: IEEE, 11 2017, pp. 19–23. [Online]. Available: http://dx.doi.org/10.1109/cyberneticscom.2017.8311708

[156] M. Schwabacher and K. Goebel, "A survey of artificial intelligence for prognostics," in *2007 AAAI Fall Symposium on Artificial Intelligence for Prognostics.* USA: AAAI, 2007, pp. 108–115. [Online]. Available: https://www.aaai.org/Library/Symposia/Fall/2007/fs07-02-016.php

[157] Z. Xue, X. Dong, S. Ma, and W. Dong, "A survey on failure prediction of large-scale server clusters," in *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007).* USA: IEEE, 7 2007, pp. 733–738. [Online]. Available: http://dx.doi.org/10.1109/snpd.2007.284

[158] D. Jauk, D. Yang, and M. Schulz, "Predicting faults in high performance computing systems: An in-depth survey of the state-of-the-practice," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 11 2019. [Online]. Available: http://dx.doi.org/10.1145/3295500.3356185

[159] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance anomaly detection and bottleneck identification," *ACM Computing Surveys*, vol. 48, no. 1, pp. 1–35, 9 2015.

[160] J. Wang, D. Rossell, C. G. Cassandras, and I. C. Paschalidis, "Network anomaly detection: A survey and comparative analysis of stochastic and deterministic methods," in *52nd IEEE Conference on Decision and Control*, 52nd IEEE Conference on Decision and Control. USA: IEEE, 12 2013, pp. 182–187. [Online]. Available: https://ieeexplore.ieee.org/document/6759879

[161] T. T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 4, pp. 56–76, 2008.

[162] M. Solé, V. Muntés-Mulero, A. I. Rana, and G. Estrada, "Survey on models and techniques for root-cause analysis," 7 2017. [Online]. Available: http://arxiv.org/abs/1701.08546

[163] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 8 2016.

[164] Z. Gao, C. Cecati, and S. X. Ding, "A survey of fault diagnosis and fault-tolerant techniques - part i: Fault diagnosis with model-based and signal-based approaches," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 6, pp. 3757–3767, Jun. 2015.

[165] ——, "A survey of fault diagnosis and fault-tolerant techniques - part ii: Fault diagnosis with knowledge-based and hybrid/active approaches," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 6, pp. 3768–3774, 6 2015.

[166] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," *Information and Software Technology*, vol. 64, pp. 1–18, Aug. 2015.

[167] P. Notaro, J. Cardoso, and M. Gerndt, "Systematic Mapping Study in AIOps - Technical Report," =https://zenodo.org/record/4109932, 2020, accessed on 2023-08-11.

[168] "IEEE Xplore." [Online]. Available: https://ieeexplore.ieee.org/Xplore/home.jsp

[169] "ACM Digital Library." [Online]. Available: https://dl.acm.org/

[170] "arXiv.org e-Print archive." [Online]. Available: https://arxiv.org/

[171] "Google Scholar." [Online]. Available: https://scholar.google.com/

[172] S. Jalali and C. Wohlin, "Systematic literature studies: database searches vs. backward snowballing," in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '12*, Lund, Sweden, 2012, p. 29.

[173] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, "Autopilot: workload autoscaling at Google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, Apr. 2020, pp. 1–16. [Online]. Available: https://doi.org/10.1145/3342195.3387524

[174] F. Gand, I. Fronza, N. El Ioini, H. Barzegar, S. Azimi, and C. Pahl, "A Fuzzy Controller for Self-adaptive Lightweight Edge Container Orchestration," in *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2020. [Online]. Available: http://dx.doi.org/10.5220/0009379600790090

[175] R. R. Noel, R. Mehra, and P. Lama, "Towards Self-Managing Cloud Storage with Reinforcement Learning," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, Jun. 2019. [Online]. Available: http://dx.doi.org/10.1109/ic2e.2019.000-9

[176] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey," *ACM Computing Surveys*, vol. 51, no. 4, pp. 1–33, Sep. 2018. [Online]. Available: http://dx.doi.org/10.1145/3148149

[177] G. Hoffmann, K. Trivedi, and M. Malek, "A Best Practice Guide to Resources Forecasting for the Apache Webserver," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06).* IEEE, 2006. [Online]. Available: http://dx.doi.org/10.1109/PRDC.2006.5

[178] R. Marcus and O. Papaemmanouil, "WiSeDB: a learning-based workload management advisor for cloud databases," *Proceedings of the VLDB Endowment*, vol. 9, no. 10, pp. 780–791, Jun. 2016. [Online]. Available: http://dx.doi.org/10.14778/2977797.2977804

[179] Y. Wu, Y. Yuan, G. Yang, and W. Zheng, "Load prediction using hybrid model for computational grid," in *2007 8th IEEE/ACM International Conference on Grid Computing.* IEEE, Sep. 2007. [Online]. Available: http://dx.doi.org/10.1109/grid.2007.4354138

[180] J. Grohmann, P. K. Nicholson, J. O. Iglesias, S. Kounev, and D. Lugones, "Monitorless: Predicting Performance Degradation in Cloud Applications with Machine Learning," in *Proceedings of the 20th International Middleware Conference.* ACM, Dec. 2019. [Online]. Available: http://dx.doi.org/10.1145/3361525.3361543

[181] A.-Y. Son, E.-N. Huh, S.-H. Na, and P.-W. Lee, "Migration scheme based machine learning for QoS in cloud computing: Survey and research challenges," in *2017 4th International Conference on Computer Applications and Information Processing Technology (CAIPT).* IEEE, Aug. 2017. [Online]. Available: http://dx.doi.org/10.1109/caipt.2017.8320748

[182] A.-y. Son and E.-N. Huh, "Study on a migration scheme by fuzzy-logic-based learning and decision approach for QoS in cloud computing," in *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN).* IEEE, Jul. 2017. [Online]. Available: http://dx.doi.org/10.1109/icufn.2017.7993836

[183] D. Basu, X. Wang, Y. Hong, H. Chen, and S. Bressan, "Learn-as-You-Go with Megh: Efficient Live Migration of Virtual Machines," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS).* IEEE, Jun. 2017. [Online]. Available: http://dx.doi.org/10.1109/icdcs.2017.173

[184] H. Hlavacs and T. Treutner, "Predicting web service levels during VM live migrations," in *2011 5th International DMTF Academic Alliance Workshop on Systems and Virtualization Management: Standards and the Cloud (SVM).* IEEE, Oct. 2011. [Online]. Available: http://dx.doi.org/10.1109/svm.2011.6096464

[185] L. Wang and E. Gelenbe, "Adaptive Dispatching of Tasks in the Cloud," *IEEE Transactions on Cloud Computing*, vol. 6, no. 1, Jan. 2018. [Online]. Available: http://dx.doi.org/10.1109/tcc.2015.2474406

[186] Y. Ran, H. Hu, X. Zhou, and Y. Wen, "DeepEE: Joint Optimization of Job Scheduling and Cooling Control for Data Center Energy Efficiency Using Deep Reinforcement Learning," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS).* IEEE, Jul. 2019. [Online]. Available: http://dx.doi.org/10.1109/icdcs.2019.00070

[187] V. T. Ravi and G. Agrawal, "A dynamic scheduling framework for emerging heterogeneous systems," in *2011 18th International Conference on High Performance Computing.* IEEE, Dec. 2011. [Online]. Available: http://dx.doi.org/10.1109/hipc.2011.6152724

[188] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin, "VCONF: a reinforcement learning approach to virtual machines auto-configuration," in *Proceedings of the 6th international conference on Autonomic computing - ICAC '09.* ACM Press, 2009. [Online]. Available: http://dx.doi.org/10.1145/1555228.1555263

[189] C. Streiffer, H. Chen, T. Benson, and A. Kadav, "DeepConfig: Automating Data Center Network Topologies Management with Machine Learning," Dec. 2017. [Online]. Available: http://arxiv.org/abs/1712.03890

[190] N. Yigitbasi, T. L. Willke, G. Liao, and D. Epema, "Towards Machine Learning-Based Auto-tuning of MapReduce," in *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems.* IEEE, Aug. 2013. [Online]. Available: http://dx.doi.org/10.1109/mascots.2013.9

[191] N. Liu, Z. Li, J. Xu, Z. Xu, S. Lin, Q. Qiu, J. Tang, and Y. Wang, "A Hierarchical Framework of Cloud Resource Allocation and Power Management Using Deep Reinforcement Learning," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS).* IEEE, Jun. 2017. [Online]. Available: http://dx.doi.org/10.1109/icdcs.2017.123

[192] T. Renugadevi, K. Geetha, N. Prabaharan, and P. Siano, "Carbon-Efficient Virtual Machine Placement Based on Dynamic Voltage Frequency Scaling in Geo-Distributed Cloud Data Centers," *Applied Sciences*, vol. 10, no. 8, p. 2701, Jan. 2020, 00006 Number: 8 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2076-3417/10/8/2701

[193] Y. Yao, L. Huang, A. B. Sharma, L. Golubchik, and M. J. Neely, "Power Cost Reduction in Distributed Data Centers: A Two-Time-Scale Approach for Delay Tolerant Workloads," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 200–211, Jan. 2014. [Online]. Available: http://dx.doi.org/10.1109/tpds.2012.341

[194] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, and W. P. Zeggert, "Proactive management of software aging," *IBM Journal of Research and Development*, vol. 45, no. 2, pp. 311–332, 3 2001.

[195] K. Zhang, J. Xu, M. R. Min, G. Jiang, K. Pelechrinis, and H. Zhang, "Automated it system failure prediction: A deep learning approach," in *2016 IEEE International Conference on Big Data (Big Data).* USA: IEEE, 12 2016, pp. 1291–1300. [Online]. Available: http://dx.doi.org/10.1109/BigData.2016.7840733

[196] T. Pitakrat, D. Okanović, A. van Hoorn, and L. Grunske, "Hora: Architecture-aware online failure prediction," *Journal of Systems and Software*, vol. 137, pp. 669–685, 3 2018.

[197] C. Bansal, S. Renganathan, A. Asudani, O. Midy, and M. Janakiraman, "Decaf: Diagnosing and triaging performance issues in large-scale cloud services," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '20. New York, NY, USA: Association for Computing Machinery, 2 2020, pp. 201–210. [Online]. Available: https://doi.org/10.1145/3377813.3381353

[198] M. Kavis, *Monitoring Strategies.* John Wiley & Sons, Ltd, 2014, ch. 12, pp. 137–147. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118691779.ch12

[199] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceeding of the 28th International Conference on Software Engineering - ICSE '06.* New York, NY, USA: Association for Computing Machinery, 2006, pp. 452–461. [Online]. Available: http://dx.doi.org/10.1145/1134285.1134349

[200] T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, 7 2000.

[201] T. Ostrand, E. Weyuker, and R. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, 4 2005.

[202] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 531–577, 8 2011.

[203] X. Yang, K. Tang, and X. Yao, "A learning-to-rank approach to software defect prediction," *IEEE Transactions on Reliability*, vol. 64, no. 1, pp. 234–246, 3 2015.

[204] K. Dejaeger, T. Verbraken, and B. Baesens, “Toward comprehensible software fault prediction models using bayesian network classifiers,” *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 237–257, 2 2013.

[205] R. Moser, W. Pedrycz, and G. Succi, “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,” in *Proceedings of the 13th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2008, pp. 181–190. [Online]. Available: http://dx.doi.org/10.1145/1368088.1368114

[206] S. Wang, T. Liu, and L. Tan, “Automatically learning semantic features for defect prediction,” in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE ’16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 297–308. [Online]. Available: http://dx.doi.org/10.1145/2884781.2884804

[207] E. Giger, M. D’Ambros, M. Pinzger, and H. C. Gall, “Method-level bug prediction,” in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM ’12*. New York, NY, USA: Association for Computing Machinery, 2012, pp. 171–180. [Online]. Available: http://dx.doi.org/10.1145/2372251.2372285

[208] K. O. Elish and M. O. Elish, “Predicting defect-prone software modules using support vector machines,” *Journal of Systems and Software*, vol. 81, no. 5, pp. 649–660, 5 2008.

[209] A. Okutan and O. T. Yıldız, “Software defect prediction using bayesian networks,” *Empirical Software Engineering*, vol. 19, no. 1, pp. 154–181, 8 2012.

[210] J. Li, P. He, J. Zhu, and M. R. Lyu, “Software defect prediction via convolutional neural network,” in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. USA: IEEE, 7 2017, pp. 318–328. [Online]. Available: http://dx.doi.org/10.1109/qrs.2017.42

[211] J. Nam, S. J. Pan, and S. Kim, “Transfer defect learning,” in *2013 35th International Conference on Software Engineering (ICSE)*. USA: IEEE, 5 2013, pp. 382–391. [Online]. Available: http://dx.doi.org/10.1109/icse.2013.6606584

[212] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, “On fault representativeness of software fault injection,” *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, 1 2013.

[213] A. Siami Namin, J. H. Andrews, and D. J. Murdoch, “Sufficient mutation operators for measuring test effectiveness,” in *Proceedings of the 13th International Conference on Software Engineering - ICSE ’08*. New York, NY, USA: Association for Computing Machinery, 2008, pp. 351–360. [Online]. Available: http://dx.doi.org/10.1145/1368088.1368136

[214] K. Vaidyanathan and K. Trivedi, “A comprehensive model for software rejuvenation,” *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 2, pp. 124–137, 2 2005.

[215] J. Alonso, J. Torres, J. L. Berral, and R. Gavalda, “Adaptive on-line software aging prediction based on machine learning,” in *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*. USA: IEEE, 6 2010, pp. 507–516. [Online]. Available: http://dx.doi.org/10.1109/dsn.2010.5544275

[216] A. Moody, G. Bronevetsky, K. Mohror, and d. B. R. Supinski, “Design, modeling, and evaluation of a scalable multi-level checkpointing system,” in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. USA: IEEE, 11 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/sc.2010.18

[217] H. Okamura, Y. Nishimura, and T. Dohi, “A dynamic checkpointing scheme based on reinforcement learning,” in *10th IEEE Pacific Rim International Symposium on Dependable Computing, 2004. Proceedings*. USA: IEEE, 3 2004, pp. 151–158. [Online]. Available: https://ieeexplore.ieee.org/document/1276566

[218] I. Jangjaimon and N.-F. Tzeng, “Effective cost reduction for elastic clouds under spot instance pricing through adaptive checkpointing,” *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 396–409, 2 2015.

[219] J. Li, R. J. Stones, G. Wang, Z. Li, X. Liu, and K. Xiao, “Being accurate is not enough: New metrics for disk failure prediction,” in *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS).* USA: IEEE, 9 2016, pp. 71–80. [Online]. Available: http://dx.doi.org/10.1109/srds.2016.019

[220] F. Mahdisoltani, I. Stefanovici, and B. Schroeder, “Proactive error prediction to improve storage system reliability,” in *2017 USENIX Annual Technical Conference (USENIX ATC 17).* Santa Clara, CA: USENIX Association, 2017, pp. 391–402. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/mahdisoltani

[221] I. Narayanan, K. Vaid, D. Wang, M. Jeon, B. Sharma, L. Caulfield, A. Sivasubramaniam, B. Cutler, J. Liu, and B. Khessib, “Ssd failures in datacenters: What? when? and why?” in *Proceedings of the 9th ACM International on Systems and Storage Conference*, ser. SYSTOR ’16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: http://dx.doi.org/10.1145/2928275.2928278

[222] G. Hamerly and C. Elkan, “Bayesian approaches to failure prediction for disk drives,” in *Proceedings of the Eighteenth International Conference on Machine Learning*, ser. ICML ’01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 6 2001, pp. 202–209. [Online]. Available: https://dl.acm.org/doi/10.5555/645530.655825

[223] A. Ma, F. Douglis, G. Lu, D. Sawyer, S. Chandra, and W. Hsu, “Raidshield: Characterizing, monitoring, and proactively protecting against disk failures,” in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST’15. USA: USENIX Association, 2015, pp. 241–256. [Online]. Available: https://www.usenix.org/conference/fast15/technical-sessions/presentation/ma

[224] B. Zhu, G. Wang, X. Liu, D. Hu, S. Lin, and J. Ma, “Proactive drive failure prediction for large scale storage systems,” in *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST).* IEEE, 5 2013, pp. 1–5. [Online]. Available: http://dx.doi.org/10.1109/msst.2013.6558427

[225] C. Xu, G. Wang, X. Liu, D. Guo, and T.-Y. Liu, “Health status assessment and failure prediction for hard drives with recurrent neural networks,” *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3502–3508, 2016.

[226] N. A. Davis, A. Rezgui, H. Soliman, S. Manzanares, and M. Coates, “Failuresim: A system for predicting hardware failures in cloud data centers using neural networks,” in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD).* IEEE, 2017, pp. 544–551. [Online]. Available: http://dx.doi.org/10.1109/cloud.2017.75

[227] S. Zheng, K. Ristovski, A. Farahat, and C. Gupta, “Long short-term memory network for remaining useful life estimation,” in *2017 IEEE International Conference on Prognostics and Health Management (ICPHM).* IEEE, 2017, pp. 88–95. [Online]. Available: http://dx.doi.org/10.1109/icphm.2017.7998311

[228] Y. Zhao, X. Liu, S. Gan, and W. Zheng, “Predicting disk failures with hmm- and hsmm-based approaches,” in *Proceedings of the 10th Industrial Conference on Advances in Data Mining: Applications and Theoretical Aspects*, ser. ICDM’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 390–404.

[229] S. Zhang, W. Meng, J. Bu, S. Yang, Y. Liu, D. Pei, J. Xu, Y. Chen, H. Dong, X. Qu, and e. al, “Syslog processing for switch failure diagnosis and prediction in datacenter networks,” in *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS).* USA: IEEE, 6 2017, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/iwqos.2017.7969130

[230] Y. Wang, Q. Miao, E. W. M. Ma, K.-L. Tsui, and M. G. Pecht, “Online anomaly detection for hard disk drives based on mahalanobis distance,” *IEEE Transactions on Reliability*, vol. 62, no. 1, pp. 136–145, 3 2013.

[231] C. H. A. Costa, Y. Park, B. S. Rosenburg, C.-Y. Cher, and K. D. Ryu, “A system software approach to proactive memory-error avoidance,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’14. New Orleans, LA, USA: IEEE Press, 2014. [Online]. Available: https://doi.org/10.1109/SC.2014.63

[232] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo, “Failure prediction in ibm bluegene/l event logs,” in *Seventh IEEE International Conference on Data Mining (ICDM 2007).* USA: IEEE, 10 2007, pp. 583–588. [Online]. Available: http://dx.doi.org/10.1109/icdm.2007.46

[233] I. Fronza, A. Sillitti, G. Succi, M. Terho, and J. Vlasenko, “Failure prediction based on log files using random indexing and support vector machines,” *Journal of Systems and Software*, vol. 86, no. 1, pp. 2–11, 1 2013.

[234] T. Chalermarrewong, T. Achalakul, and S. C. W. See, “Failure prediction of data centers using time series and fault tree analysis,” in *2012 IEEE 18th International Conference on Parallel and Distributed Systems.* USA: IEEE, 12 2012, pp. 794–799. [Online]. Available: http://dx.doi.org/10.1109/icpads.2012.129

[235] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase, “Correlating instrumentation data to system states: a building block for automated diagnosis and control,” in *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, ser. OSDI’04. San Francisco, CA: USENIX Association, 12 2004, p. 16. [Online]. Available: https://dl.acm.org/doi/10.5555/1251254.1251270

[236] T. Islam and D. Manivannan, “Predicting application failure in cloud: A machine learning approach,” in *2017 IEEE International Conference on Cognitive Computing (ICCC).* USA: IEEE, 6 2017, pp. 24–31. [Online]. Available: http://dx.doi.org/10.1109/ieee.iccc.2017.11

[237] F. Salfner and M. Malek, “Using hidden semi-markov models for effective online failure prediction,” in *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007).* USA: IEEE, 10 2007, pp. 161–174. [Online]. Available: http://dx.doi.org/10.1109/srds.2007.35

[238] B. Sharma, P. Jayachandran, A. Verma, and C. R. Das, “Cloudpd: Problem determination and diagnosis in shared dynamic clouds,” in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* USA: IEEE, 6 2013, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1109/dsn.2013.6575298

[239] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, “Magpie: Online modelling and performance-aware systems,” in *Proceedings of the 9th Conference on Hot Topics in Operating Systems*, vol. 9. Lihue, Hawaii, USA: USENIX Association, 5 2003, p. 15. [Online]. Available: https://dl.acm.org/doi/10.5555/1251054.1251069

[240] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, “Execution anomaly detection in distributed systems through unstructured log analysis,” in *2009 Ninth IEEE International Conference on Data Mining.* USA: IEEE Computer Society, 12 2009, pp. 149–158. [Online]. Available: http://dx.doi.org/10.1109/ICDM.2009.60

[241] A. Lakhina, M. Crovella, and C. Diot, “Diagnosing network-wide traffic anomalies,” in *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications - SIGCOMM ’04.* New York, NY, USA: Association for Computing Machinery, Aug. 2004, pp. 219–230. [Online]. Available: http://dx.doi.org/10.1145/1015467.1015492

[242] H. Xu, Y. Feng, J. Chen, Z. Wang, H. Qiao, W. Chen, N. Zhao, Z. Li, J. Bu, Z. Li, and e. al, “Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications,” in *Proceedings of the 2018 World Wide Web Conference on World Wide Web - WWW ’18*, ser. WWW ’18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 187–196. [Online]. Available: http://dx.doi.org/10.1145/3178876.3185996

[243] C. Zhang, D. Song, Y. Chen, X. Feng, C. Lumezanu, W. Cheng, J. Ni, B. Zong, H. Chen, and N. V. Chawla, "A deep neural network for unsupervised anomaly detection and diagnosis in multivariate time series data," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33. USA: AAAI, 2019, pp. 1409–1416.

[244] 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*, ser. KDD '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 2828–2837.

[245] J. Audibert, P. Michiardi, F. Guyard, S. Marti, and M. A. Zuluaga, "Usad: Unsupervised anomaly detection on multivariate time series," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 3395–3404. [Online]. Available: https://doi.org/10.1145/3394486.3403392

[246] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with csight," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 468–479. [Online]. Available: https://doi.org/10.1145/2568225.2568246

[247] H. Nguyen, Z. Shen, Y. Tan, and X. Gu, "Fchain: Toward black-box online fault localization for cloud systems," in *2013 IEEE 33rd International Conference on Distributed Computing Systems*. USA: IEEE, 7 2013, pp. 21–30. [Online]. Available: http://dx.doi.org/10.1109/icdcs.2013.26

[248] D. Liu, Y. Zhao, H. Xu, Y. Sun, D. Pei, J. Luo, X. Jing, and M. Feng, "Opprentice: Towards practical and automatic anomaly detection through machine learning," in *Proceedings of the 2015 Internet Measurement Conference*. New York, NY, USA: Association for Computing Machinery, 2015, pp. 211–224. [Online]. Available: http://dx.doi.org/10.1145/2815675.2815679

[249] A. Lakhina, M. Crovella, and C. Diot, "Mining anomalies using traffic feature distributions," in *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '05. New York, NY, USA: Association for Computing Machinery, 2005, pp. 217–228. [Online]. Available: https://doi.org/10.1145/1080091.1080118

[250] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun, "Classification of software behaviors for failure detection: A discriminative pattern mining approach," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 557–566. [Online]. Available: https://doi.org/10.1145/1557019.1557083

[251] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USA: USENIX Association, 2014, pp. 217–231. [Online]. Available: https://dl.acm.org/doi/10.5555/2685048.2685066

[252] 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 *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. USA: International Joint Conferences on Artificial Intelligence Organization, 7 2019, pp. 4739–4745.

[253] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, J. Chen, X. He, R. Yao, J.-G. Lou, M. Chintalapati, F. Shen, and D. Zhang, "Robust log-based anomaly detection on unstable log data," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 807–817. [Online]. Available: https://doi.org/10.1145/3338906.3338931

[254] W. Wang, M. Zhu, J. Wang, X. Zeng, and Z. Yang, "End-to-end encrypted traffic classification with one-dimensional convolution neural networks," in *2017 IEEE International Conference on Intelligence and Security Informatics (ISI).* USA: IEEE, 7 2017, pp. 43–48. [Online]. Available: http://dx.doi.org/10.1109/isi.2017.8004872

[255] T. Auld, A. W. Moore, and S. F. Gull, "Bayesian neural networks for internet traffic classification," *IEEE Transactions on Neural Networks*, vol. 18, no. 1, pp. 223–239, 1 2007.

[256] A. Este, F. Gringoli, and L. Salgarelli, "Support vector machines for tcp traffic classification," *Computer Networks*, vol. 53, no. 14, pp. 2476–2490, 9 2009.

[257] A. W. Moore and D. Zuev, "Internet traffic classification using bayesian analysis techniques," in *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems - SIGMETRICS '05.* New York, NY, USA: Association for Computing Machinery, Jun. 2005, pp. 50–60. [Online]. Available: http://dx.doi.org/10.1145/1064212.1064220

[258] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering.* USA: IEEE, 5 2015, pp. 415–425. [Online]. Available: http://dx.doi.org/10.1109/icse.2015.60

[259] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "Log20: Fully automated optimal placement of log printing statements under specified overhead threshold," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 10 2017, pp. 565–581. [Online]. Available: http://dx.doi.org/10.1145/3132747.3132778

[260] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, "Towards highly reliable enterprise network services via inference of multi-level dependencies," in *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 13–24. [Online]. Available: https://doi.org/10.1145/1282380.1282383

[261] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering - SIGSOFT '02/FSE 10*, ser. SIGSOFT '02/FSE-10. New York, NY, USA: Association for Computing Machinery, 2002, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1145/587051.587053

[262] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: Association for Computing Machinery, 2005, pp. 342–351. [Online]. Available: https://doi.org/10.1145/1062455.1062522

[263] Y. Sun, Y. Zhao, Y. Su, D. Liu, X. Nie, Y. Meng, S. Cheng, D. Pei, S. Zhang, X. Qu *et al.*, "Hotspot: Anomaly localization for additive kpis with multi-dimensional attributes," *IEEE Access*, vol. 6, pp. 10 909–10 923, 2018.

[264] Z. Li, C. Luo, Y. Zhao, Y. Sun, K. Sui, X. Wang, D. Liu, X. Jin, Q. Wang, and D. Pei, "Generic and robust localization of multi-dimensional root causes," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE. USA: IEEE, 2019, pp. 47–57.

[265] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: statistical model-based bug localization," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, p. 286, 9 2005.

[266] R. Abreu, P. Zoeteweij, and A. J. van Gemund, "Spectrum-based multiple fault localization," in *2009 IEEE/ACM International Conference on Automated Software Engineering.* USA: IEEE, 11 2009, pp. 88–99. [Online]. Available: http://dx.doi.org/10.1109/ASE.2009.25

[267] M. Renieris and S. Reiss, "Fault localization with nearest neighbor queries," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* USA: IEEE, 2003, pp. 30–39. [Online]. Available: https://ieeexplore.ieee.org/document/1240292

[268] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 3 2014.

[269] F. Lin, K. Muzumdar, N. P. Laptev, M.-V. Curelea, S. Lee, and S. Sankar, "Fast dimensional analysis for root cause investigation in a large-scale service environment," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 2, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3392149

[270] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. USA: USENIX Association, 10 2012, pp. 307–320. [Online]. Available: https://dl.acm.org/doi/10.5555/2387880.2387910

[271] S. Kandula, D. Katabi, and J.-P. Vasseur, "Shrink: a tool for failure diagnosis in ip networks," in *Proceeding of the 2005 ACM SIGCOMM workshop on Mining network data - MineNet '05*, ser. MineNet '05. New York, NY, USA: Association for Computing Machinery, 2005. [Online]. Available: http://dx.doi.org/10.1145/1080173.1080178

[272] A. Samir and C. Pahl, "A controller architecture for anomaly detection, root cause analysis and self-adaptation for cluster architectures," 2019. [Online]. Available: https://orbilu.uni.lu/handle/10993/42062

[273] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: problem determination in large, dynamic internet services," in *Proceedings International Conference on Dependable Systems and Networks.* Washington, DC, USA: IEEE Comput. Soc, 6 2002, pp. 595–604. [Online]. Available: http://dx.doi.org/10.1109/DSN.2002.1029005

[274] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: error diagnosis by connecting clues from run-time logs," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 143–154, 3 2010.

[275] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Proceedings of the 25th International Conference on Software Engineering.* USA: IEEE, 2003, pp. 465–475. [Online]. Available: http://dx.doi.org/10.1109/icse.2003.1201224

[276] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP '05. New York, NY, USA: Association for Computing Machinery, 2005, pp. 105–118. [Online]. Available: http://dx.doi.org/10.1145/1095810.1095821

[277] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 102–111. [Online]. Available: http://dx.doi.org/10.1145/2889160.2889232

[278] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 74–89, 12 2003.

[279] Q. Shao, Y. Chen, S. Tao, X. Yan, and N. Anerousis, "Efficient ticket routing by resolution sequence mining," in *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 605–613. [Online]. Available: http://dx.doi.org/10.1145/1401890.1401964

[280] C. Zeng, W. Zhou, T. Li, L. Shwartz, and G. Y. Grabarnik, “Knowledge guided hierarchical multi-label classification over ticket data,” *IEEE Transactions on Network and Service Management*, vol. 14, no. 2, pp. 246–260, 6 2017.

[281] W. Zhou, L. Tang, T. Li, L. Shwartz, and G. Y. Grabarnik, “Resolution recommendation for event tickets in service management,” in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. USA: IEEE, 5 2015, pp. 287–295. [Online]. Available: http://dx.doi.org/10.1109/inm.2015.7140303

[282] Q. Wang, W. Zhou, C. Zeng, T. Li, L. Shwartz, and G. Y. Grabarnik, “Constructing the knowledge base for cognitive it service management,” in *2017 IEEE International Conference on Services Computing (SCC)*. USA: IEEE, 6 2017, pp. 410–417. [Online]. Available: http://dx.doi.org/10.1109/scc.2017.59

[283] 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 *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’12. USA: USENIX Association, 2012, pp. 293–306.

[284] J. Bogatinovski, S. Nedelkoski, A. Acker, J. Cardoso, and O. Kao, “Qulog: Data-driven approach for log instruction quality assessment,” in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ser. ICPC ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 275–286. [Online]. Available: https://doi.org/10.1145/3524610.3527906

[285] Z. E. Li, M. Liang, L. O’Brien, and H. Zhang, “The cloud’s cloudy moment: A systematic survey of public cloud service outage,” *International Journal of Cloud Computing and Services Science (IJ-CLOSER)*, vol. 2, 12 2013.

[286] Q. Lin, K. Hsieh, Y. Dang, H. Zhang, K. Sui, Y. Xu, J.-G. Lou, C. Li, Y. Wu, R. Yao, M. Chintalapati, and D. Zhang, “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*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 480–490. [Online]. Available: https://doi.org/10.1145/3236024.3236060

[287] A. Dörflinger, Y. Guan, S. Michalik, S. Michalik, J. Naghmouchi, and H. Michalik, “ECC memory for fault tolerant RISC-V processors,” in *Architecture of Computing Systems–ARCS 2020: 33rd International Conference, Aachen, Germany, May 25–28, 2020, Proceedings 33*. Springer, 2020, pp. 44–55.

[288] E. Pinheiro, W.-D. Weber, and L. A. Barroso, “Failure trends in a large disk drive population,” in *5th USENIX Conference on File and Storage Technologies (FAST 07)*, ser. FAST ’07. USA: USENIX Association, Feb. 2007, p. 2. [Online]. Available: https://www.usenix.org/conference/fast-07/failure-trends-large-disk-drive-population

[289] S. Lu, B. Luo, T. Patel, Y. Yao, D. Tiwari, and W. Shi, “Making Disk Failure Predictions SMARTer!” in *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, ser. FAST’20. USA: USENIX Association, 2020, p. 151–168. [Online]. Available: https://tinyurl.com/bdenn2y6

[290] C. Luo, P. Zhao, B. Qiao, Y. Wu, H. Zhang, W. Wu, W. Lu, Y. Dang, S. Rajmohan, Q. Lin *et al.*, “Ntam: Neighborhood-temporal attention model for disk failure prediction in cloud platforms,” in *Proceedings of the Web Conference 2021*, 2021, pp. 1181–1191.

[291] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, “Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field,” in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Rio de Janeiro, Brazil: IEEE, 2015, pp. 415–426.

[292] B. Schroeder, E. Pinheiro, and W.-D. Weber, “DRAM Errors in the Wild: a Large-scale Field Study,” *Communications of the ACM*, vol. 54, no. 2, pp. 100–107, 2011. [Online]. Available: https://tinyurl.com/5t4bxt38

[293] V. Sridharan and D. Liberty, "A study of DRAM failures in the field," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11. [Online]. Available: https://doi.org/10.1109/SC.2012.13

[294] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory Errors in Modern Systems: The Good, The Bad, and The Ugly," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 297–310. [Online]. Available: https://doi.org/10.1145/2694344.2694348

[295] M. V. Beigi, Y. Cao, S. Gurumurthi, C. Recchia, A. Walton, and V. Sridharan, "A systematic study of ddr4 dram faults in the field," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 991–1002.

[296] Q. Yu, W. Zhang, P. Notaro, S. Haeri, J. Cardoso, and O. Kao, "HiMFP: Hierarchical intelligent memory failure prediction for cloud service reliability," in *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2023, pp. 216–228.

[297] I. Giurgiu, J. Szabo, D. Wiesmann, and J. Bird, "Predicting DRAM reliability in the field with machine learning," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track*, ser. Middleware '17. New York, NY, USA: Association for Computing Machinery, Dec. 2017, pp. 15–21, 00026. [Online]. Available: https://doi.org/10.1145/3154448.3154451

[298] I. Boixaderas, D. Zivanovic, S. Moré, J. Bartolome, D. Vicente, M. Casas, P. M. Carpenter, P. Radojković, and E. Ayguadé, "Cost-aware prediction of uncorrected DRAM errors in the field," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. Atlanta, Georgia: IEEE Press, Nov. 2020, pp. 1–15, 00005.

[299] X. Du, C. Li, S. Zhou, M. Ye, and J. Li, "Predicting Uncorrectable Memory Errors for Proactive Replacement: An Empirical Study on Large-Scale Field Data," in *2020 16th European Dependable Computing Conference (EDCC)*, Sep. 2020, pp. 41–46.

[300] X. Du and C. Li, "Predicting Uncorrectable Memory Errors from the Correctable Error History: No Free Predictors in the Field," in *The International Symposium on Memory Systems*, ser. MEMSYS 2021. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3488423.3519316

[301] C. Li, Y. Zhang, J. Wang, H. Chen, X. Liu, T. Huang, L. Peng, S. Zhou, L. Wang, and S. Ge, "From correctable memory errors to uncorrectable memory errors: What error bits tell," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '22. IEEE Press, 2022.

[302] X. Wang, Y. Li, Y. Chen, S. Wang, Y. Du, C. He, Y. Zhang, P. Chen, X. Li, W. Song, Q. xu, and L. Jiang, "On workload-aware dram failure prediction in large-scale data centers," in *2021 IEEE 39th VLSI Test Symposium (VTS)*, 2021, pp. 1–6.

[303] P. Zhang, Y. Wang, X. Ma, Y. Xu, B. Yao, X. Zheng, and L. Jiang, "Predicting DRAM-Caused Node Unavailability in Hyper-Scale Clouds," in *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2022, pp. 275–286. [Online]. Available: https://doi.org/10.1109/DSN53405.2022.00037

[304] C. Mendoza, V. Dasari, and M. P. McGarry, "Using SFP Data to Detect Contaminated Fiber Connectors," in *2018 IEEE 10th Latin-American Conference on Communications (LATINCOM)*, 2018, pp. 1–6.

[305] A. Chakravarty, S. Giridharan, M. Kelly, A. Poojary, and V. Zeng, "Characterizing Large-Scale Production Reliability for 100G Optical Interconnect in Facebook Data Centers," in *Frontiers in Optics*. Optica Publishing Group, 2017. [Online]. Available: https://tinyurl.com/3pmb3tm8

[306] Z. Wang, M. Zhang, D. Wang, C. Song, M. Liu, J. Li, L. Lou, and Z. Liu, “Failure prediction using machine learning and time series in optical network,” *Opt. Express*, vol. 25, no. 16, pp. 18 553–18 565, Aug 2017. [Online]. Available: https://doi.org/10.1364/OE.25.018553

[307] J. Li, Z. Wang, C. Wang, Q. Chen, P. Wang, R. Lu, S. Fu, and C. Xie, “Data analytics practice for reliability management of optical transceivers in hyperscale data centers,” in *Optical Fiber Communication Conference (OFC) 2020.* Optica Publishing Group, 2020, p. T3K.6. [Online]. Available: https://tinyurl.com/yc8ma6e9

[308] T. Tanaka, T. Inui, S. Kawai, S. Kuwabara, and H. Nishizawa, “Monitoring and diagnostic technologies using deep neural networks for predictive optical network maintenance,” *J. Opt. Commun. Netw.*, vol. 13, no. 10, pp. E13–E22, Oct 2021. [Online]. Available: https://tinyurl.com/bddw7k2c

[309] D. Liu, Y. Yang, Z. Tang, and Z. He, “Implementation of optical module performance prediction and maintenance on data-driven,” in *Eighth Symposium on Novel Photoelectronic Detection Technology and Applications*, vol. 12169. SPIE, 2022, pp. 3332–3336.

[310] T. Goyal, A. Singh, and A. Agrawal, “Cloudsim: simulator for cloud computing infrastructure and modeling,” *Procedia Engineering*, vol. 38, pp. 3566–3572, 1 2012.

[311] S. Zhang, Y. Liu, W. Meng, Z. Luo, J. Bu, S. Yang, P. Liang, D. Pei, J. Xu, Y. Zhang, Y. Chen, H. Dong, X. Qu, and L. Song, “Prefix: Switch failure prediction in datacenter networks,” in *Abstracts of the 2018 ACM International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 64–66. [Online]. Available: https://doi.org/10.1145/3219617.3219643

[312] A. Jindal, I. Shakhat, J. Cardoso, M. Gerndt, and V. Podolskiy, “IAD: Indirect Anomalous VMMs Detection in the Cloud-Based Environment,” in *Service-Oriented Computing – ICSOC 2021 Workshops*, H. Hacid, M. Aldwairi, M. R. Bouadjenek, M. Petrocchi, N. Faci, F. Outay, A. Beheshti, L. Thamsen, and H. Dong, Eds. Springer International Publishing, 2022, pp. 190–201.

[313] F. Cerveira, J. Domingos, R. Barbosa, and H. Madeira, “Measuring lead times for failure prediction,” in *2021 IEEE 26th Pacific Rim International Symposium on Dependable Computing (PRDC)*, Dec. 2021, pp. 1–5, iSSN: 2473-3105.

[314] LeMichael and TamirYuval, “ReHype: Enabling VM Survival Across Hypervisor Failures,” *ACM SIGPLAN Notices*, Mar. 2011, publisher: ACM PUB27 New York, NY, USA. [Online]. Available: https://dl.acm.org/doi/10.1145/2007477.1952692

[315] A. Rawat, R. Sushil, A. Agarwal, and A. Sikander, “A new approach for vm failure prediction using stochastic model in cloud,” *IETE Journal of Research*, vol. 67, no. 2, pp. 165–172, 2021. [Online]. Available: https://doi.org/10.1080/03772063.2018.1537814

[316] A. Jindal, P. Staab, P. Kulkarni, J. Cardoso, M. Gerndt, and V. Podolskiy, “Memory leak detection algorithms in the cloud-based infrastructure,” 2021.

[317] P. Notaro, Q. Yu, S. Haeri, J. Cardoso, and M. Gerndt, “An Optical Transceiver Reliability Study based on SFP Monitoring and OS-level Metric Data,” in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2023, pp. 1–12.

[318] Ponemon Institute, “2016 Cost of Data Center Outages,” https://www.ponemon.org/research/ponemon-library/security/2016-cost-of-data-center-outages.html, 2016, accessed on: 2023-08-23.

[319] M. M. Botezatu, I. Giurgiu, J. Bogojeska, and D. Wiesmann, “Predicting disk replacement towards reliable data centers,” in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 39––48. [Online]. Available: https://doi.org/10.1145/2939672.2939699

[320] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, vol. 78, no. 1, p. 012022, jul 2007. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/78/1/012022

[321] G. Wang, L. Zhang, and W. Xu, "What Can We Learn from Four Years of Data Center Hardware Failures?" in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017, pp. 25–36. [Online]. Available: https://ieeexplore.ieee.org/document/8023108

[322] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 350–361. [Online]. Available: https://doi.org/10.1145/2018436.2018477

[323] B. Schroeder and G. A. Gibson, "Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You?" in *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, ser. FAST '07. USA: USENIX Association, 2007. [Online]. Available: https://tinyurl.com/4bh75jzy

[324] J. G. Elerath, "AFR: problems of definition, calculation and measurement in a commercial environment," in *Annual Reliability and Maintainability Symposium. 2000 Proceedings. International Symposium on Product Quality and Integrity.* IEEE, 2000, pp. 71–76.

[325] S. Zhang, W. Meng, J. Bu, S. Yang, Y. Liu, D. Pei, J. Xu, Y. Chen, H. Dong, X. Qu, and L. Song, "Syslog Processing for Switch Failure Diagnosis and Prediction in Datacenter Networks," in *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*, 2017. [Online]. Available: https://doi.org/10.1109/IWQoS.2017.7969130

[326] J. Xiao, Z. Xiong, S. Wu, Y. Yi, H. Jin, and K. Hu, "Disk Failure Prediction in Data Centers via Online Learning," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3225058.3225106

[327] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler, "An analysis of latent sector errors in disk drives," in *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 289–300. [Online]. Available: https://doi.org/10.1145/1254882.1254917

[328] X. Du and C. Li, "Memory failure prediction using online learning," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '18. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 38–49. [Online]. Available: https://doi.org/10.1145/3240302.3240309

[329] Z. Wu, H. Xu, G. Pang, F. Yu, Y. Wang, S. Jian, and Y. Wang, "DRAM Failure Prediction in AIOps: Empirical Evaluation, Challenges and Opportunities," *arXiv:2104.15052 [cs]*, May 2021. [Online]. Available: http://arxiv.org/abs/2104.15052

[330] A. Vahdat, H. Liu, X. Zhao, and C. Johnson, "The Emerging Optical Data Center," in *Optical Fiber Communication Conference/National Fiber Optic Engineers Conference 2011.* Optica Publishing Group, 2011. [Online]. Available: https://doi.org/10.1364/OFC.2011.OTuH2

[331] American National Standard - INCITS, "Working Draft ATA/ATAPI Command Set - 3 (ACS-3)," https://people.freebsd.org/~imp/asiabsdcon2015/works/d2161r5-ATAATAPI_Command_Set_-_3.pdf, pp. 56–58, accessed on: 2023-08-17.

[332] L. Aronson *et al.*, *Digital Diagnostic Monitoring Interface for Optical Transceivers*, Small Form Factor Committee (SFF) Std., Rev. 12.0, 2000. [Online]. Available: https://tinyurl.com/2p8dsc7m

[333] Acronis Knowledge Base, "S.M.A.R.T." https://kb.acronis.com/tag/smart-0?ckattempt=1, accessed on: 2023-08-30.

[334] “Hard Drive Data and Stats,” accessed on: 2023-08-30. [Online]. Available: https://www.backblaze.com/cloud-storage/resources/hard-drive-test-data

[335] C. Xie, C. Wang, Q. Chen, Z. Wang, P. Wang, R. Lu, and L. Wang, “Characteristics of Field Operation Data for Optical Transceivers in Hyperscale Data Centers,” in *Optical Fiber Communication Conference (OFC) 2022.* Optica Publishing Group, 2022, p. Th2A.1. [Online]. Available: https://doi.org/10.1364/OFC.2022.Th2A.1

[336] G. Snively, *Gigabit Interface Converter (GBIC)*, Storage Networking Industry Association (SNIA) Std., Rev. 5.5, 2000, accessed: 2023-03-03. [Online]. Available: https://tinyurl.com/2p9aju8b

[337] *Small Form-Factor Pluggable (SFP) Transceiver MultiSource Agreement*, Storage Networking Industry Association (SNIA) Std., 2001, revision 1.0, accessed: 2023-02-24. [Online]. Available: https://tinyurl.com/57kp27nj

[338] C. Kachris, K. Bergman, and I. Tomkos, *Optical Interconnects for Future Data Center Networks.* Springer Science & Business Media, 2012. [Online]. Available: https://doi.org/10.1007/978-1-4614-4630-9

[339] IEEE Communications Society (ComSoc). (2022) 5G Optical Transceiver Market Trends and Technologies. Accessed: 2023-03-03. [Online]. Available: https://tinyurl.com/ya76ax3p

[340] Persistent Market Research. (2022) Market Study on Optical Transceivers: Need for High-speed Networking Infrastructure to Drive Market Expansion. Accessed: 2022-10-24. [Online]. Available: https://tinyurl.com/2p8fskvx

[341] S. Aleksic, “The Future of Optical Interconnects for Data Centers: A Review of Technology Trends,” in *2017 14th International Conference on Telecommunications (ConTEL)*, 2017. [Online]. Available: https://doi.org/10.23919/ConTEL.2017.8000037

[342] IBM - SAN Switch Documentation. (2022) Maintaining SFP, SFP+, or QSFP+ transceivers and fiber-optic cables. Accessed: 2023-03-03. [Online]. Available: https://tinyurl.com/2w78d7jw

[343] W. W. Peterson and D. T. Brown, “Cyclic codes for error detection,” *Proceedings of the IRE*, vol. 49, no. 1, pp. 228–235, 1961.

[344] Avago Technologies. (2023) Digital diagnostic monitoring interface (dmi) transceivers: Applications and implementation. Accessed: 2023-02-17. [Online]. Available: https://tinyurl.com/5n8ncmhk

[345] P. Notaro, S. Haeri, J. Cardoso, and M. Gerndt, “Command-line risk classification using transformer-based neural architectures,” *in press*, 2023, submitted to the 2023 Conference on Empirical Methods for Natural Language Processing (2023). [Online]. Available: http://tiny.cc/nlp-command-line

[346] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, “Why do internet services fail, and what can be done about it?” in *4th Usenix Symposium on Internet Technologies and Systems (USITS 03)*, 2003.

[347] H. Adkins, B. Beyer, P. Blankinship, P. Lewandowski, A. Oprea, and A. Stubblefield, *Building secure and reliable systems: best practices for designing, implementing, and maintaining systems.* O’Reilly Media, 2020. [Online]. Available: https://google.github.io/building-secure-and-reliable-systems/raw/toc.html

[348] I. Karakurt, S. Özer, T. Ulusinan, and M. C. Ganiz, “A machine learning approach to database failure prediction,” in *2017 International Conference on Computer Science and Engineering (UBMK)*, 2017, pp. 1030–1035.

[349] A. Kamra, E. Terzi, and E. Bertino, “Detecting anomalous access patterns in relational databases,” *The VLDB Journal*, vol. 17, no. 5, pp. 1063–1077, 2008.

[350] J. Gao, H. Wang, and H. Shen, “Task failure prediction in cloud data centers using deep learning,” *IEEE Transactions on Services Computing*, vol. 15, no. 3, pp. 1411–1422, 2022.

[351] M. Hajiaghayi and E. Vahedi, “Code failure prediction and pattern extraction using lstm networks,” 2018. [Online]. Available: https://arxiv.org/abs/1812.05237

[352] D. Hendler, S. Kels, and A. Rubin, “Detecting malicious powershell commands using deep neural networks,” *CoRR*, vol. abs/1804.04177, 2018. [Online]. Available: http://arxiv.org/abs/1804.04177

[353] Z. Hussain, J. K. Nurminen, T. Mikkonen, and M. Kowiel, “Command Similarity Measurement Using NLP,” in *10th Symposium on Languages, Applications and Technologies (SLATE 2021)*, vol. 94. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 13:1–13:14. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/14430

[354] W. Kehe, C. Wenchao, C. Fei, and A. Yanwen, “A method of remote operation and maintenance intelligent audit,” 2014. [Online]. Available: https://patents.google.com/patent/CN104156439A/

[355] A. M. Romanenko, I. O. Tolstikhin, and S. V. Prokudin, “System and method for evaluating malware detection rules,” 2013. [Online]. Available: https://patents.google.com/patent/CN104156439A/

[356] D. Trizna, “Shell language processing: Unix command parsing for machine learning,” 2022.

[357] X. Zhang, J. Zhao, and Y. LeCun, “Character-level convolutional networks for text classification,” 2016.

[358] M. M. Yamin and B. Katt, “Detecting malicious windows commands using natural language processing techniques,” in *Innovative Security Solutions for Information Technology and Communications*, J.-L. Lanet and C. Toma, Eds. Cham: Springer International Publishing, 2019, pp. 157–169.

[359] A. Zhou, T. Huang, C. Huang, D. Li, and C. Song, “Pycomm: Malicious commands detection model for python scripts,” *Journal of Intelligent & Fuzzy Systems*, vol. 42, pp. 1–13, 11 2021.

[360] R. Sennrich, B. Haddow, and A. Birch, “Neural machine translation of rare words with subword units,” in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725. [Online]. Available: https://aclanthology.org/P16-1162

[361] P. Gage, “A new algorithm for data compression,” *C Users J.*, vol. 12, no. 2, pp. 23–38, feb 1994.

[362] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, “Intellicode compose: Code generation using transformer,” 2020.

[363] M. Kerrisk. (2021) Manpages - sections of the manual pages. [Online]. Available: https://man7.org/linux/man-pages/man7/man-pages.7.html

[364] P. Notaro, S. Haeri, and J. Cardoso, “Apparatus and method for auditing rule-based command risk assessment systems,” Germany Patent 11 987 272, 2023, filed at European Patent Office (EPO), currently under revision. Filing date: 2023-04-28.

[365] Y. W. *et al.*, “Google’s neural machine translation system: Bridging the gap between human and machine translation,” 2016. [Online]. Available: https://arxiv.org/abs/1609.08144

[366] Google Research. Github - google-research/bert. [Online]. Available: https://github.com/google-research/bert

[367] S. Ghosh, M. Shetty, C. Bansal, and S. Nath, “How to fight production incidents? an empirical study on a large-scale cloud service,” in *Proceedings of the 13th Symposium on Cloud Computing*, ser. SoCC ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 126–141. [Online]. Available: https://doi.org/10.1145/3542929.3563482

[368] S. Velayutham and G. Shanmugam, “Artificial intelligence assisted canary testing of cloud native ran in a mobile telecom system,” 2021. [Online]. Available: http://uu.diva-portal.org/smash/get/diva2:1581042/FULLTEXT01.pdf

[369] S. Pritchard, V. Nagaraju, and L. Fiondella, “Automating staged rollout with reinforcement learning,” in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2022, pp. 16–20.

[370] M. Solé, V. Muntés-Mulero, A. I. Rana, and G. Estrada, “Survey on models and techniques for root-cause analysis,” 2017. [Online]. Available: http://arxiv.org/abs/1701.08546

[371] P. Notaro, S. Haeri, J. Cardoso, and M. Gerndt, “LogRule: Efficient structured log mining for root cause analysis,” *IEEE Transactions on Network and Service Management*, 2023.

[372] J. Liu, J. Zhu, S. He, P. He, Z. Zheng, and M. R. Lyu, “Logzip: Extracting hidden structures via iterative clustering for log compression,” *CoRR*, vol. abs/1910.00409, 2019. [Online]. Available: http://arxiv.org/abs/1910.00409

[373] K. Yao, M. Sayagh, W. Shang, and A. E. Hassan, “Improving state-of-the-art compression techniques for log management tools,” *IEEE Transactions on Software Engineering*, vol. 48, no. 8, 2022.

[374] R. Vaarandi and M. Pihelgas, “Logcluster - a data clustering and pattern mining algorithm for event logs,” in *2015 11th International Conference on Network and Service Management (CNSM)*, 2015, pp. 1–7.

[375] J. Lu, F. Li, L. Li, and X. Feng, “Cloudraid: Hunting concurrency bugs in the cloud via log-mining,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 3–14. [Online]. Available: https://doi.org/10.1145/3236024.3236071

[376] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl, “Detailed diagnosis in enterprise networks,” in *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, ser. SIGCOMM ’09. New York, NY, USA: Association for Computing Machinery, 2009, p. 243–254. [Online]. Available: https://doi.org/10.1145/1592568.1592597

[377] H. Yan, L. Breslau, Z. Ge, D. Massey, D. Pei, and J. Yates, “G-rca: A generic root cause analysis platform for service quality management in large ip networks,” *IEEE/ACM Transactions on Networking*, vol. 20, no. 6, pp. 1734–1747, 2012.

[378] A. A. Mahimkar, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and Q. Zhao, “Towards automated performance diagnosis in a large iptv network,” in *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, ser. SIGCOMM ’09. New York, NY, USA: Association for Computing Machinery, 2009, p. 231–242. [Online]. Available: https://doi.org/10.1145/1592568.1592596

[379] R. Bhagwan, R. Kumar, R. Ramjee, G. Varghese, S. Mohapatra, H. Manoharan, and P. Shah, “Adtributor: Revenue debugging in advertising systems,” in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’14. USA: USENIX Association, 2014, p. 43–55.

[380] 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*, ser. ICSE ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 214–224. [Online]. Available: https://doi.org/10.1145/2884781.2884795

[381] Y. Sun, Y. Zhao, Y. Su, D. Liu, X. Nie, Y. Meng, S. Cheng, D. Pei, S. Zhang, X. Qu, and X. Guo, “Hotspot: Anomaly localization for additive kpis with multi-dimensional attributes,” *IEEE Access*, vol. 6, pp. 10 909–10 923, 2018.

[382] 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*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 60–70. [Online]. Available: https://doi.org/10.1145/3236024.3236083

[383] Z. Li, C. Luo, Y. Zhao, Y. Sun, K. Sui, X. Wang, D. Liu, X. Jin, Q. Wang, and D. Pei, "Generic and robust localization of multi-dimensional root causes," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 47–57.

[384] X. Zhang, Y. Bai, P. Feng, W. Wang, S. Liu, W. Jiang, J. Zeng, and R. Wang, "Network alarm flood pattern mining algorithm based on multi-dimensional association," in *Proceedings of the 21st ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, ser. MSWIM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 71–78.

[385] D. Delvia Arifin, Shaufiah, and M. A. Bijaksana, "Enhancing spam detection on mobile phone short message service (sms) performance using fp-growth and naive bayes classifier," in *2016 IEEE Asia Pacific Conference on Wireless and Mobile (APWiMob)*. USA: IEEE, 2016, pp. 80–84.

[386] S. Suriadi, C. Ouyang, W. M. P. van der Aalst, and A. H. M. ter Hofstede, "Root Cause Analysis with Enriched Process Logs," in *Business Process Management Workshops*, M. La Rosa and P. Soffer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 174–186.

[387] M. Castelluccio, C. Sansone, L. Verdoliva, and G. Poggi, "Automatically analyzing groups of crashes for finding correlations," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 717–726. [Online]. Available: https://doi.org/10.1145/3106237.3106306

[388] S. Bagui and P. C. Dhar, "Positive and negative association rule mining in hadoop's mapreduce environment," *Journal of Big Data*, vol. 6, no. 1, pp. 1–16, 2019.

[389] R. Liu, K. Yang, Y. Sun, T. Quan, and J. Yang, "Spark-based rare association rule mining for big datasets," in *2016 IEEE International Conference on Big Data (Big Data)*. USA: IEEE, 2016.

[390] I. F. Videla-Cavieres and S. A. Ríos, "Extending market basket analysis with graph mining techniques: A real case," *Expert Syst. Appl.*, vol. 41, no. 4, p. 1928–1936, Mar. 2014. [Online]. Available: https://doi.org/10.1016/j.eswa.2013.08.088

[391] V. Fuller, T. Li, J. Yu, and K. Varadhan, "RFC1519: Classless Inter-Domain Routing (CIDR): An Address Assignment and Aggregation Strategy," IETF, USA, Tech. Rep., 1993.

[392] T. A. Kumbhare and S. V. Chobe, "An overview of association rule mining algorithms," *International Journal of Computer Science and Information Technologies*, vol. 5, no. 1, pp. 927–930, 2014.

[393] J. M. Luna, M. Ondra, H. M. Fardoun, and S. Ventura, "Optimization of quality measures in association rule mining: an empirical study," *International Journal of Computational Intelligence Systems*, vol. 12, pp. 59–78, 2018.

[394] Weaveworks, "Sock Shop - A Microservices Demo Application," https://microservices-demo.github.io/, 2017, accessed on: 2023-09-07.

[395] Elastic, "Elastic - examples (github)," https://tinyurl.com/apache-access-01, 2017.

[396] "Apache accesslog - 2," https://tinyurl.com/apache-access-02.

[397] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," *CoRR*, 2018. [Online]. Available: http://arxiv.org/abs/1811.03509