Anomaly Symptom Recognition in Distributed IT Systems

vorgelegt von
M. Sc.
Alexander Acker

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

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

Promotionsausschuss:

Vorsitzender: Prof. Dr. Jan Nordholz
Gutachter: Prof. Dr. Odej Kao
Gutachter: Prof. Dr. David Bermbach
Gutachter: Prof. Dr. Carsten Griwodz

Tag der wissenschaftlichen Aussprache: 08. Oktober 2021

Berlin 2021

Acknowledgements

The realization of this thesis would not have been possible without the help of brilliant, supportive, understanding, and loving people around me. I want to take the opportunity to express my gratitude to them.

First, I want to thank my advisor Odej Kao, who allowed me to work in his research group full of brilliant researchers. You always granted me the freedom and trust, allowing me to realize my research ideas that eventually lead to this thesis. Thank you for your guidance, support, honesty, and a lot of patience. I also want to thank David Bermbach and Carsten Griwodz for agreeing to review this thesis.

I am thankful to my colleagues and former colleagues. It was always a pleasure to discuss ideas, share knowledge and experiences, or just talk about weekend plans with all of you. I especially want to thank Florian Schmidt, who supervised my master's thesis and ultimately inspired me to pursue a Ph.D. degree. A special thanks to Sasho and Anton. Jointly working with both of you pushed me to improve my personal and professional abilities. Anton, Florian, Marcel, and Sören, thank you for the numerous enjoyable trips to Munich. I will never forget how we jointly prepared our presentations or live demos and celebrated successful project deliveries with Bavarian beer. Thank you Lauritz, Jasmin, Lilly, Thorsten, Dominik, Kevin, Tim, Vincent, and Andreas. An enormous thank you to Jana for all the organizational and administrative help.

I was very lucky to be involved in several collaborative research projects with Huawei, Deutsche Telekom, Siemens, and KSB. It gave me the opportunity to work with remarkable people who were an incredible source of knowledge and experience. Thank you Feng, Eliezer, Jorge, Wu Fan, Tingyao, Bingnan, Matvey, Ajo, Joachim, Sebastian, Gerhard, Philipp, Tobias, and Jochen.

This work would not have been possible without the endless support of my family and friends. Thank you, Marie, for your love, support, and patience, especially during the highly intense phases of my work. Thank you мама and папа for guiding me through life, teaching me to follow my dreams with determination, and showing me the value of mutual respect. Thank you to my Sisters, Elena and Nathalie, my brother, Daniel, and my son, Jonas, for providing the distraction that I needed sometimes. Finally, I want to thank my friends for all the adventures and special moments we share.

Zusammenfassung

Die fortschreitende globale Digitalisierung treibt die Entwicklung von Netzwerktechnologien, Analyseplattformen und datengetriebenen Diensten voran. Damit einhergehend ist eine steigende Anzahl an Komponent wie Sensoren, Aktoren, Rechen- und Netzwerkknoten sowie unterschiedlichen Applikationen. Infolgedessen nimmt die Komplexität von IT Infrastrukturen stetig zu. Ein System mit hoher Komplexität ist anfällig für Fehler oder Ausfälle jedoch erwarten Anwender, dass die Dienste immer verfügbar sind. Darüber hinaus ist eine hohe Verfügbarkeit unerlässlich für die Nutzung von IT Systemen in kritischen Bereichen wie Medizin, Logistik, Energie oder der Fertigungsindustrie. Dort haben Ausfälle, die nicht schnell genug behoben werden, katastrophale Folgen. Die Konsequenz daraus ist, dass Betreiber zunehmend mit der Aufgabe überfordert sind die notwendige Verfügbarkeit zu gewährleisten.

Dies erfordert Lösungen, die den Betrieb und die Wartung von komplexen IT Systemen unterstützen. Dafür wird der Einsatz KI-gestützter Methoden erforscht, die die Wartbarkeit, Verfügbarkeit und Zuverlässigkeit von IT Systemen verbessern sollen. Diese werden eingesetzt um Systemkomponenten zu überwachen, die Überwachungsdaten zu analysieren und bei Bedarf automatisch Operationen auszuwählen und auszuführen, um einen effizienten Betriebszustand aufrechtzuerhalten. Durch diese Automatisierung von Wartungs- und Administrationsaufgaben soll eine höhere Robustheit gegenüber Ausfällen realisiert werden.

In dieser Arbeit erfoschen wir Methoden, die die Verfügbarkeit von IT Systemen erhöhen sollen, indem die notwendige Zeit für die Behebung von Fehlern und Ausfällen reduziert wird. Dazu werden Daten von Systemkomponenten, deren Betriebszustand von einer bekannten Norm abweicht, nach spezifischen Mustern durchsucht. Diese als *Anomaliesymptome* bezeichneten Muster dienen dazu Fehlerfälle zu identifizieren. Falls diese bereits in der Vergangenheit aufgetreten sind und erfolgreich Behoben wurden, ermöglicht das eine automatisierte und somit schnelle Wiederherstellung eines normalen Systemzustands. Die von uns entwickelten Erkennungsmethoden sind in der Lage Muster, die noch nicht bekannt sind, zu identifizieren und deren Behandlung an menschliche Experten zu delegieren. Dieser "Human-in-the-Loop"-Ansatz stellt eine schrittweise Übertragung des Wissens von menschlichen Experten in unser System dar.

Abstract

The progressing global digitalization is driving innovative network technologies, computation platforms, and data-driven services. The number of components such as sensors, actuators, computing, storage, and network nodes, as well as a variety of service applications increases and results in IT systems of high complexity. A complex system is prone to errors or failures, but users expect services always to be available. Furthermore, high availability is essential for utilizing IT systems in critical areas such as medicine, logistics, energy, or the manufacturing industry. There, failures that are not immediately resolved can lead to hazardous situations. Consequently, system operators are increasingly overwhelmed with the task of keeping complex IT systems at an operational state.

Solutions that support the operation and maintenance of complex IT systems are required to support humans. For this purpose, artificial intelligence for IT system operations (AIOps) is being explored to improve the availability, maintainability, and reliability of IT systems. It combines the research areas of artificial intelligence, machine learning, and system operation to monitor relevant components, analyze the monitoring data, and automatically select and execute operations to maintain an efficient operational state. The automation should enable improved robustness against failures.

This thesis introduces methods to increase the availability of IT systems by reducing the time required to resolve errors and failures. Thereby, system components whose operational state deviates from a known norm are referred to as *anomalies*. We employ pattern recognition to search monitoring data from anomalous components for specific patterns. The identification of these *anomaly symptoms* allows a comparison to historical occurrences of anomalies and an automatic selection of feasible operations to resolve them. Further, our implemented methods can identify patterns that are representing yet unknown anomalies. Such cases are delegated to human experts. This "human-in-the-loop" approach represents a step-by-step transfer of knowledge from human experts into our system.

Contents

List of Figures

List of Tables

Acronyms and Abbreviations

AI	Artificial Intelligence
AIOps	Artificial Intelligence for System Operations
BGP	Border Gateway Protocol
CPU	Central Processing Unit
DevOps	Development and Operation
DTW	Dynamic Time Warping
DAG	Directed Acyclic Graph
DSL	Domain Specific Language
GP	Grid Pattern
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
IaC	Infrastructure as Code
IMS	IP Multimedia Subsystem
IoT	Internet of Things
IP	Internet Protocol
IRQ	Interquartile Range
LSTM	Long Short-Term Memory
ML	Machine Learning
MTBF	Mean Time Between Failure
MTTR	Mean Time to Recovery
NFV	Network Function Virtualization
NRE	Network reliability Engineer

Ops	Operations
RC	Root Cause
RCA	Root Cause Analysis
ReCoMe	Recommendation and Remediation
RNN	Recurrent Neural Network
SIP	Session Initiation Protocol
SLA	Service Level Agreement
SRE	Site Reliability Engineer
SuO	System Under Operation
STL	Season and Trend Decomposition based on Loess
URL	Uniform Resource Locator
VM	Virtual Machine
VNF	Virtual Network Function
ZerOps	Zero-Touch Operations

Chapter 1

Introduction

Digitalization is an integral part of today's world. Our everyday life increasingly depends on web and mobile applications. Furthermore, numerous industries like energy, transportation, manufacturing, and health care depend on data-driven services. Thereby, the number of components within the whole IT ecosystem is growing fast. Reports from Gartner and Cisco predict that the number of interconnected devices will reach numbers between 20 and 30 billion by 2022 [84, 124]. These trends require network connectivity and computational resources to meet the growing demand for digitalization. Network technologies like 5G or 6G enable access and connectivity, while cloud computing systems provide computation and storage resources. Emerging technologies such as fog and edge computing promise low latencies and reduced data volume transmissions to data centers, which is achieved by providing location-agnostic service access. Data should be processed as close as possible to its source. Although these trends introduce increased flexibility and improve business opportunities, they come at the cost of significantly higher complexity.

Human users and a growing amount of machines expect highly available and reliable services. Moreover, the high availability of IT systems in critical industries is a fundamental requirement since unresolved problems can lead to hazardous situations. Danger to our environment or even human life must be prevented. However, system complexity grows, which leads to an increased proneness to failures [49, 178]. An empirical study done by Fiondella et al. [83] analyzes system outages of major IT companies and shows an exponential growth of the number of outage incidents per year. Another empirical study conducted by Garraghan et al. [93] analyzes a cloud computing environment of 12,500 servers. It observes the average amount of time between the occurrences of two consecutive failures on the same server, which is referred to as *Mean Time Between Failure* (MTBF). For servers, an MTBF of 12.5 days is identified. Considering the 12,500 servers, this translates to an average of 1000 failures per day. Furthermore, the time needed to resolve the failure is analyzed, which is referred to as *Mean Time To Recovery* (MTTR). The MTTR values for servers follow a heavy-tailed distribution with an overall mean value of 5.2 hours. Their study provides no explicit information about the type of failures. Other studies show that failed hardware causes less than 10% of failures, and around 80% are caused by software or operational mistakes [99, 111].

Human experts are currently responsible for the operation and maintenance of IT systems. However, recovering 1000 failures per day is a challenge. Additionally, the diversity of soft- and hardware components from different vendors and an increasing geographical distribution aggravates that difficulty. These aspects result in longer MTTR values and are reflected in the

increased demand for qualified and specialized system operators. Numerous job positions like DevOps experts, Site Reliability Engineers (SREs), or Network reliability Engineers (NREs) are created to cope with the challenges of system operation [27]. Nevertheless, human experts are increasingly overwhelmed with the operation of the rapidly growing systems [29, 48]. Solutions supporting the operation and maintenance of distributed systems are required to prevent a complete loss of control. Such support aims at adding autonomy to distributed systems by enabling capabilities to resolve occurring problems automatically [138]. Therefore, human involvement should be gradually reduced by implementing methods that autonomously keep system components at an operational state.

Artificial intelligence for IT System operations (AIOps) is an upcoming research field concerned with developing such methods. It combines the areas of artificial intelligence (AI), machine learning (ML), and system operations (Ops). Thereby, solutions are usually designed as a closed-loop control that consists of several methods [138, 213]. First, the observability of systems is realized by monitoring relevant system metric data at runtime. Second, anomaly detection analyzes these data to detect if component states are deviating from a known norm [210, 222]. Thereby, *anomaly* refers either to a failed system component or to a degraded state that potentially indicates an imminent failure. Latter enables the recovery of components before failures happen, giving human experts more time to react and plan. At this point, human operators are still left to analyze anomalies, compiling operations, and executing them. Working with current production systems puts high pressure on experts since every minute of downtime or service degradation entails losses in revenue for companies [78, 166]. Working with systems that are used in critical industries entails that operators must assume a potential danger for human lives if problems are not resolved fast enough. Thus, the third step closes the loop by automatically selecting and eventually executing *recovery* or *remediation* operations that transfer the anomalous component to a known norm. It finally enables fast recovery of anomalies and the reduction of the MTTR [6, 96].

In this work, we aim at enabling an automatic selection of remediation or recovery operations to restore the normal state of anomalous system components. This process is initiated after anomalies are detected. We assume to receive alarms that contain binary information, i.e., a component is either normal or anomalous [171, 211, 222]. Based on this, the recent monitoring data from the anomalous component is searched for specific patterns, which enable the recognition of anomaly states. The knowledge about a specific anomaly state, or *anomaly type*, is then used to select recovery operations and eventually restore a normal operational state. The patterns within the monitoring data that are specific for anomaly types are referred to as *anomaly symptoms*. We name the identification of anomaly type-specific symptoms in the monitoring data of anomalous components as *anomaly symptom recognition*.

The precise recognition of anomaly type-specific symptom patterns is challenging in complex IT systems, consisting of diverse components affected by numerous internal and external factors. Examples of diversity are different hardware specifications, changing software versions, or varying configuration parameters. Examples of internal or external factors are the availability of computation resources or the current workload. During anomalies, the observed monitoring data represent a mixture of anomaly-related patterns and patterns specific for the component type and the current internal and external factors, which lead to deviations between symptoms of the same anomaly type, i.e., *intra-type variation*. Furthermore, anomalies are expected to occur infrequently. It is challenging to identify variating symptom patterns based on a few available examples [28].

1.1 Problem Definition

An AIOps system requires an automatic selection of operations to resolve anomaly states of components that are part of a complex IT system. The research question is formulated as follows:

> "Given the monitoring data of an anomalous IT system component, how an automatic selection of operations to restore its normal functional state can be enabled?"

We realize the automatic selection of operations by recognizing anomaly type-specific symptom patterns in the monitoring data of an anomalous system component. The automation stems from the assumption that anomaly types represented by similar symptom patterns repeatedly occur in a complex IT system. Methods for symptom pattern recognition and anomaly type inference based on the monitoring data of anomalous system components are required. They must be efficient in the sense that precise pattern detection is realized without causing an extensive overhead.

We focus on the following four research questions:

1. We investigate the ability of pattern analysis methods to recognize anomaly type-specific symptoms in monitoring data from anomalous components and emphasize the limited availability of anomaly type examples.

2. Distributed systems are dynamic, and their components must be assumed to change over time. Also, new components can appear. We evaluate the ability of our pattern recognition methods to identify symptoms that represent yet unknown anomaly types.

3. The value of an IT system stems from the execution of a regular workload. Automation methods to keep it operational should consume a small fraction of available resources. We investigate the trade-off that is required between computational complexity and precise symptom recognition.

4. We integrate the anomaly symptom recognition methods into an AIOps platform and test whether an efficient and precise anomaly symptom recognition can be employed alongside regular workload execution. Thereby, different deployment concepts are tested to infer general deployment guidelines.

1.2 Contributions

This thesis proposes methods to enable the automatic selection of operations that resolve detected anomalies in components of a complex IT system. The concrete contributions are:

1. The precise recognition of anomaly type-specific symptom patterns is challenging in complex IT systems. The observed monitoring data represent a mixture of anomaly-related and varying normal patterns, resulting in intra-type variations. The existence of few available examples in combination with high intra-type variations is the major challenge to recognize anomaly type-specific symptoms. We develop two methods for

anomaly symptom recognition. One focuses on low computation overhead, while the other applies complex computation to recognize symptom patterns under high intra-type variation.

2. The occurrence of previously not observed anomaly types must be considered. Software systems undergo updates, hardware is changed, and new components are deployed. These are few examples of the system dynamics. Furthermore, an initially empty knowledge base, i.e., *cold-start*, needs to be considered when deployment is done on a new IT system. Based on this, we extend the methods and enable the recognition of yet unknown anomaly types. We further introduce a human-in-the-loop approach to gradually extend the knowledge base enabling automatic operation selection for an increasing number of anomaly types.

3. A system needs to process a regular workload. Only a fraction of available resources can be reserved for keeping it in an operational state. We analyze the trade-off between the computational overhead for high anomaly symptom recognition precision and the efficiency due to the limited availability of computation resources.

4. Different AIOps methods must interact with each other to achieve automatic handling of occurring anomalies. To conduct a functional investigation of the anomaly symptom recognition, we integrate our methods into the AIOps platform called ZerOps. We investigate the AIOps deployment requirements of cloud and fog computing systems. Thereby, different deployment concepts are tested. Based on the result, we elaborate on guidelines to realize an efficient AIOps platform deployment.

The work of this thesis builds upon the following list of peer-reviewed articles:

[1] Acker, Alexander and Schmidt, Florian and Gulenko, Anton and Kao, Odej. “Online Density Grid Pattern Analysis to Classify Anomalies in Cloud and NFV Systems”. In: *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2018, pp. 290–295.

[2] Acker, Alexander and Schmidt, Florian and Gulenko, Anton and Kietzmann, Reinhard and Kao, Odej. “Patient-individual morphological anomaly detection in multi-lead electrocardiography data streams”. In: *2017 IEEE International Conference on Big Data (Big Data)*. IEEE. 2017, pp. 3841–3846.

[3] Acker, Alexander and Wittkopp, Thorsten and Nedelkoski, Sasho and Bogatinovski, Jasmin and Kao, Odej. “Superiority of Simplicity: A Lightweight Model for Network Device Workload Prediction”. In: *2020 15th Conference on Computer Science and Information Systems (FedCSIS)*. IEEE. 2020, pp. 7–10.

[4] Becker, Soeren and Schmidt, Florian and Gulenko, Anton and Acker, Alexander and Kao, Odej. “Towards AIOps in Edge Computing Environments”. In: *2020 IEEE International Conference on Big Data (Big Data)*. IEEE. 2020, pp. 3470–3475.

[5] Gulenko, Anton and Acker, Alexander and Kao, Odej and Liu, Feng. “AI-Governance and Levels of Automation for AIOps-supported System Administration”. In: *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. IEEE. 2020, pp. 1–6.

[6] Gulenko, Anton and Acker, Alexander and Schmidt, Florian and Becker, Soeren and Kao, Odej. "Bitflow: An In Situ Stream Processing Framework". In: *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. IEEE. 2020, pp. 182–187.

[7] Gulenko, Anton and Schmidt, Florian and Acker, Alexander and Wallschlaeger, Marcel and Kao, Odej and Liu, Feng. "Detecting anomalous behavior of black-box services modeled with distance-based online clustering". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE. 2018, pp. 912–915.

[8] Nedelkoski, Sasho and Bogatinovski, Jasmin and Acker, Alexander and Cardoso, Jorge and Kao, Odej. "Self-Attentive Classification-Based Anomaly Detection in Unstructured Logs". In: *2020 IEEE International Conference on Data Mining (ICDM)*. IEEE. 2020, pp. 1196–1201.

[9] Nedelkoski, Sasho and Bogatinovski, Jasmin and Acker, Alexander and Cardoso, Jorge and Kao, Odej. "Self-supervised Log Parsing". In: *Machine Learning and Knowledge Discovery in Databases: Applied Data Science Track*. Springer. 2021, pp. 122–138.

[10] Ott, Harold and Bogatinovski, Jasmin and Acker, Alexander and Nedelkoski, Sasho and Kao, Odej. "Robust and Transferable Anomaly Detection in Log Data using Pre-Trained Language Models". In: *Workshop Proceedings of the 43rd International Conference on Software Engineering*. ACM. 2021.

[11] Scheinert, Dominik and Acker, Alexander. "Telesto: A graph neural network model for anomaly classification in cloud services". In: *Service-Oriented Computing – ICSOC 2020 Workshops*. Springer International Publishing. 2021, pp. 214–227.

[12] Scheinert, Dominik and Acker, Alexander and Thamsen, Lauritz and Geldenhuys, Morgan K and Kao, Odej. "Learning Dependencies in Distributed Cloud Applications to Identify and Localize Anomalies". In: *Workshop Proceedings of the 43rd International Conference on Software Engineering*. ACM. 2021.

[13] Schmidt, Florian and Gulenko, Anton and Wallschlaeger, Marcel and Acker, Alexander and Hennig, Vincent and Liu, Feng and Kao, Odej. "Iftm-unsupervised anomaly detection for virtualized network function services". In: *2018 IEEE International Conference on Web Services (ICWS)*. IEEE. 2018, pp. 187–194.

[14] Schmidt, Florian and Suri-Payer, Florian and Gulenko, Anton and Wallschlaeger, Marcel and Acker, Alexander and Kao, Odej. "Unsupervised anomaly event detection for cloud monitoring using online arima". In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE. 2018, pp. 71–76.

[15] Schmidt, Florian and Suri-Payer, Florian and Gulenko, Anton and Wallschlaeger, Marcel and Acker, Alexander and Kao, Odej. "Unsupervised anomaly event detection for vnf service monitoring using multivariate online arima". In: *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2018, pp. 278–283.

[16] Schroeder, Daniel Thilo and Styp-Rekowski, Kevin and Schmidt, Florian and Acker, Alexander and Kao, Odej. “Graph-based Feature Selection Filter Utilizing Maximal Cliques”. In: *2019 Sixth International Conference on Social Networks Analysis, Management and Security (SNAMS)*. IEEE. 2019, pp. 297–302.

[17] Wallschlaeger, Marcel and Acker, Alexander and Kao, Odej. “Silent Consensus: Probabilistic Packet Sampling for Lightweight Network Monitoring”. In: *International Conference on Computational Science and Its Applications*. Springer. 2019, pp. 241–256.

[18] Wallschlaeger, Marcel and Gulenko, Anton and Schmidt, Florian and Acker, Alexander and Kao, Odej. “Anomaly Detection for Black Box Services in Edge Clouds Using Packet Size Distribution”. In: *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*. IEEE. 2018, pp. 1–6.

[19] Wittkopp, Thorsten and Acker, Alexander. “Decentralized federated learning preserves model and data privacy”. In: *Service-Oriented Computing – ICSOC 2020 Workshops*. Springer International Publishing. 2021, pp. 176–187.

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

1.3 Outline of the Thesis

The rest of this thesis is structured as follows.

Chapter 2 provides the necessary background on service deployment in cloud and fog computing systems, the dependability of such system, a failure model, and an overview of fault tolerance policies.

Chapter 3 gives an extensive overview of the related work. Thereby, we focus on the vision of autonomous computing, self-healing frameworks, AIOps in general, and related AIOps methods for monitoring data analysis.

Chapter 4 introduces the concept of automatic selection of remediation operations. We analyze the problems and requirements of a solution for complex IT systems. After that, our anomaly type recognition objective is presented, and its embedding into an AIOps system is shown. The system architecture is described and analyzed in detail before a modeling concept for anomaly symptom recognition is presented.

Chapter 5 presents our first method for anomaly symptom recognition. It combines a transformation of monitoring data to a grid space, resulting in different grid patterns for the respective anomaly symptoms. A comparison of grid patterns realizes the anomaly symptom recognition. Furthermore, we introduce a cloud computing environment, where evaluation experiments are conducted and evaluate the ability to recognize a set of anomaly types.

Chapter 6 shows the necessity of a method specialized in the classification of sequential data and realizes this in the form of a model based on gated recurrent units (GRUs). It employs a method of higher computational complexity that should be applied to anomalies with a high intra-type variation. A focus lies on improving the training of such models when few anomaly type examples are available by proposing an augmentation method for monitoring

data. The evaluation shows the effect of data augmentation and investigates the ability to recognize anomaly types on components that execute heterogeneous workloads.

Chapter 7 describes the implementation of the anomaly type recognition and the automatic operation selection into an AIOps platform named ZerOps. It is a system prototype wherein we integrate both our methods. We test different method combinations and deployment concepts to derive conclusions for the applicability of such systems.

Chapter 8 concludes our work, summarizes findings, and identifies directions for future research.

Chapter 2

Background

This chapter presents the background for this work. Our general focus is to increase a distributed systems' dependability by enabling an automatic selecting remediation operations when system components become anomalous. We first give a definition of dependability and explain cloud and fog computing systems focusing on their dependability. After that, several failure models from the literature are presented. They are essential to describe the transition of a normal system state to an failure. Based on this transition, we apply the terminology on the previously described distributed systems and derive the term *anomaly*. Third, we elaborate on different fault-tolerance policies, which aim at managing systems that are expected to fail over time. From this, we draw implications for the design of the automatic selection of operations to resolve anomalies and their embedding into a AIOps system.

2.1 Dependability of Distributed Systems

Computer systems are expected to eventually fail over time an thus, suffer from reduced dependability [100, 169, 175, 178, 191]. Different attributes are used to measure the dependability of a system of which the following five are commonly used [120, 141, 223]. The *availability* defines the percentage of time within a specific time frame, during which the service is delivered correctly. It is usually defined as the MTBF divided by the sum of MTBF and MTTR. The *reliability* is the expected time that a system operates without failures. It is essential to distinguish availability from reliability. A system can fail once a year and be unavailable for two weeks or can fail every day and be unavailable for one minute. Considering one year as the time frame, the former is more reliable but has the availability of 96.2% while the latter is less reliable but has the availability of 99.9%. It can be seen that the MTTR plays an essential role in the availability of a system. By reducing the MTTR the availability can be increased. *Safety* is a measure of catastrophic effects when failures occur, and *maintainability* quantifies the effort to repair a failed system.

A general rule is that a system increasing in size and complexity becomes less dependable [244]. Northrop et al. [178] analyze the challenges of large distributed systems and identify the preservation of high dependability as a critical problem. Based on these and other studies [19, 63, 111], the status quo is that complex distributed systems will fail over time. This reduced dependability is a fundamental problem since many IT companies rely on the availability of their services [78]. Therefore, the necessity for concepts that tolerate the occurrence

of failures by mitigating their effects was recognized early on [32, 33, 122, 191].

2.1.1 Cloud Computing

The relevance of distributed systems is growing, especially since the beginning of the commercial use of cloud computing [9, 85, 159]. Fox et al. [85] analyze the main drivers behind the success of cloud computing. They identify simpler payment methods for internet-based services, lower prices for electricity and data center housing, and growing demand for applications such as interactive web and mobile applications and big data analytic services as the main drivers. These observations partly comply with the essential characteristics of cloud computing given by NIST [159]: On-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service. Instead of deploying and managing their own IT infrastructures, many companies favor hosting their applications and services within cloud computing systems [78, 201].

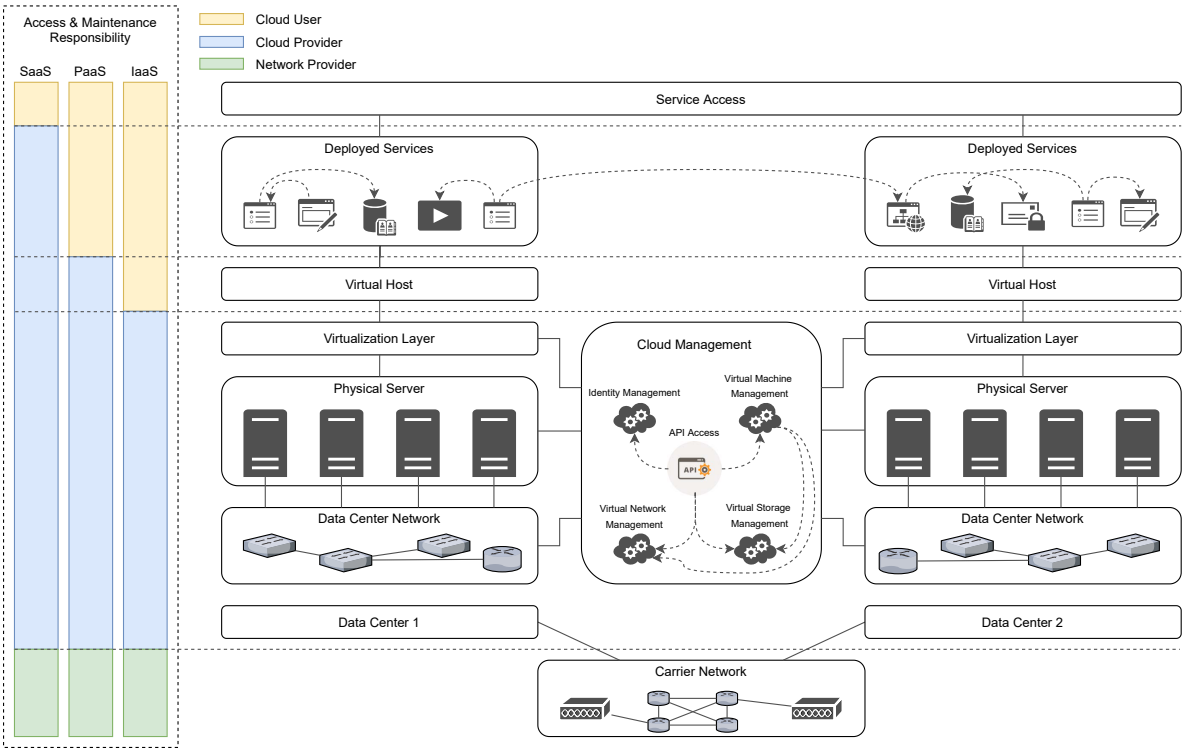


Figure 2.1: Overview of cloud computing systems.

Figure 2.1 shows an exemplary cloud computing system deployed across two data centers, its layered architecture, and its distributed nature. The carrier network connects both data centers and is usually managed by network providers. It can be seen that physical servers are connected via the data center's internal network. Networking within the data center, the computation, and storage hardware, and the virtualization and management software are maintained by cloud providers [20, 159]. The virtualization layer provides an abstraction to decouple physical hardware from virtual host provisioning. The cloud management is a collection of services controlling different functions of the cloud computing system. The depicted example consists of an access API and identity, virtual network, storage, and machine management ser-

vices. Real cloud management systems contain a variety of such interdependent management services [214]. The virtual hosts contain user applications that are usually accessed via the Internet. Example interdependencies between user applications are shown as well. The distribution of responsibilities between the cloud provider and application owner depends on the service model. There are three basic service models defined by NIST [159]

- ◇ **Software as a Service (SaaS)**: The application is entirely managed by the cloud provider and is accessed via interfaces. Only the services provided by the application are accessible to customers.

- ◇ **Platform as a Service (PaaS)**: Applications are developed via programming languages, libraries, services, and tools that the cloud provider supports. The customer has control over the deployment of the application.

- ◇ **Infrastructure as a Service (IaaS)**: The customer has control over virtually provisioned fundamental computing resources such as computation, storage, and networking. It is possible to deploy arbitrary software, including the operating system.

The number of services hosted within cloud computing systems increases constantly [7, 162, 201]. Service providers that deploy their applications in cloud computing systems entrust the operation of a significant part of the IT infrastructure to the cloud providers. For connectivity, cloud providers rely on the carrier network which is managed by network providers. In this setup, many stakeholders do both, provide own and rely on external services, resulting in a system where services are highly interdependent. The availability of each service can impact the availability of the whole system. The legal basis of such interdependencies is the definition of service level agreements (SLAs) [24]. They define particular service criteria between the provider and the user of a service and measure these via observable metrics. SLAs also contain defined penalties for service providers when these metrics fall below certain thresholds. These penalties are usually refunds or additional monetary compensation payments. A widely used metric is the availability of services. For example, as of April 2021, the Google Cloud Platform entitles the customer to a refund of 10%, 25%, and 100% when a single virtual machine availability falls respectively below 99.9%, 99% and 95% during one month [1]. The same concept is applied for Amazon AWS [2] and Microsoft Azure [3]. The availability and refund values differ. Whether a service is responsive defines its "availability", which is a binary decision. However, a reduced service quality such as a higher response time is possible. The SLAs, as mentioned above, do not cover these degraded service states.

To operate continuously growing distributes systems, industry and public research develop methods to increase availability while keeping the operational costs moderate [33, 69, 94, 153]. By combining the fact that complex systems are prone to failures and the requirement of high availability, we conclude that occurring failures must be recovered as fast as possible. A low MTTR directly translated to higher availability. A possible solution is to introduce additional automation into the handling of system anomalies.

[1] `https://cloud.google.com/compute/sla` (last access 28 April 2021)
[2] `https://aws.amazon.com/compute/sla` (last access 28 April 2021)
[3] `https://azure.microsoft.com/en-us/support/legal/sla` (last access 28 April 2021)

2.1.2 Fog and Edge Computing

The increasing number of interconnected devices is predicted to pose a significant challenge for the existing IT infrastructures [25, 84, 124]. An exponentially increasing data volume combined with low latency requirements of services makes it challenging to solve this with cloud computing alone. Despite their described benefits, cloud computing systems are centralized and usually far away from end-user devices, sensors, or other components. Sending data back and forth over long distances brings the disadvantages of high latency and high energy consumption for network transfer. Furthermore, it is difficult for network providers to keep up with the increasing data volume transferred via their networks.

Methods to analyze and reduce the network latency and bandwidth usage are researched [38, 98]. A widely pursued solution in industry and research is to move computational resources closer to the end devices [25, 150, 186]. This concept is depicted in Figure 2.2. A set of intermediate components that provide computational resources is located between end devices and the cloud computing system. Preprocessing, filtering, compressing, or complete data analysis can be done as close as possible to the data source [187], which potentially enabled more data privacy since a central storage can be avoided [98, 179] Only relevant and required data are expected to be sent to cloud data centers. There are different terminologies for solutions and computational paradigms that are referred to as *fog computing*, *mist computing*, *cloudlets*, or *edge computing* [130]. However, at the time this work was written, no commonly agreed consensus existed. Many works refer to the first set of nodes after end devices that provide computational resources as the *edge* [25, 150]. Others refer to end devices, on which it is possible to execute workload, as edge devices [130]. The set of intermediate nodes between the end devices and the cloud computing systems is usually referred to as *fog*.

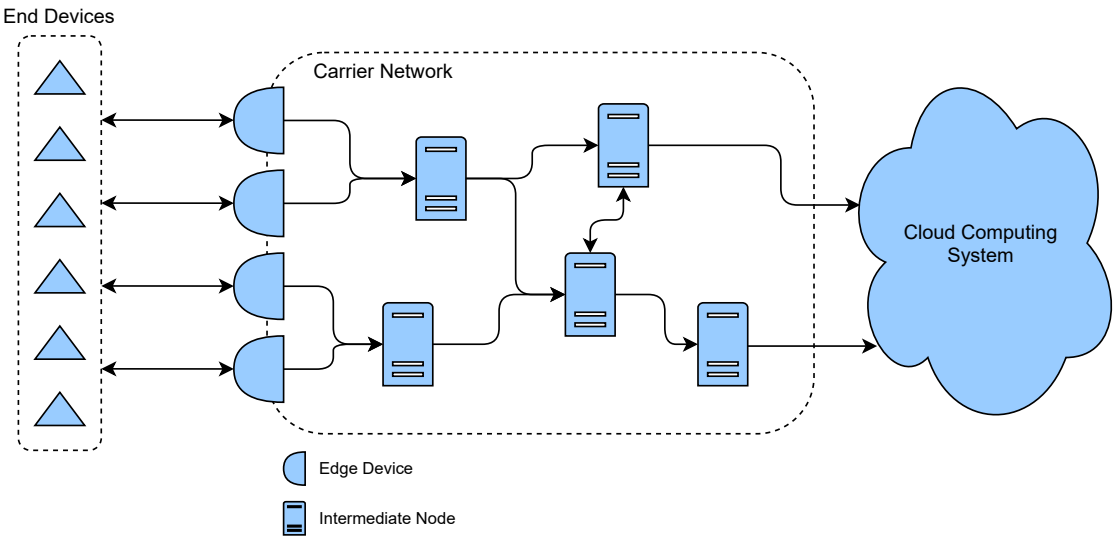


Figure 2.2: Moving computational resources closer to end devices. Adjusted from [25].

Providing nodes between end devices and the cloud computing system entail geographical distribution. Failures in such systems have a significant impact on dependability [18]. These nodes are not located in central data centers but are expected to be exposed and potentially non-stationary. This requires a widely available mobile connectivity [195], adaptivity [220], and access and mobility management [79]. Whenever anomalies occur, a remote connection is

required to resolve them. However, anomalies can affect the connectivity. The last option is a local recovery which requires human operators to move to the remote fog nodes. It results in an overall increased MTTR and, thus, a reduced availability [151]. Integrating additional automation into fog nodes to locally resolve eventually occurring anomalies is a solution to reduce MTTR and increase availability.

2.2 Failure Models

A failure model is required to analyze failures in IT systems [14, 68, 192]. It consists of definitions that describe the evolution from a normal system state to a failure. Thereby, a widely used definition of failures is provided by Avižienis and Laprie:

> "A system failure occurs when the delivered service deviates from the specified service, where the service specification is an agreed description of the expected service." [14]

Other works adopt this definition [144, 203] or a variation of it [68, 200, 227]. The central aspect of this definition is that external entities (other services or users) notice the discrepancy between expected and delivered service results. Furthermore, the definition implies that failures are effects of a system state that deviates from the norm.

There are different existent categorizations of failures [22, 67, 203, 227]. A widely applied failure categorization is provided by Cristian et al. [67] which is further extended by Laranjeira et al. [142]. It defines six failure categories:

- ◇ **Crash failure**: It is also referred to as fail-stop failure [203, 227]. The service is not responsive and remains unresponsive until it is repaired.
- ◇ **Omission failure**: The service does not respond to a request. The difference to a crash failure is that the service remains responsive in general but fails to respond to certain requests.
- ◇ **Performance failure**: The service response is delivered with an intolerable delay. Computing the results took longer than expected.
- ◇ **Timing failure**: The response was requested to be delivered at a certain time interval. However, the service responds outside of this time interval.
- ◇ **Incorrect computation failure**: The results contained in the response are incorrect.
- ◇ **Arbitrary failure** A default category where all failures are assigned to which do not fit any of the above categories.

Barroso and Hölzle [22] provide a simplified definition of four failure categories descendingly sorted by their severity.

- ◇ **Corrupted**: The service is not responding to requests (fail-stop failure). The state of the service is lost and cannot be restored.
- ◇ **Unreachable**: The service is not responding (fail-stop failure).

- ◇ **Degraded**: The service responds, but the received results deviate from the expected norm.
- ◇ **Masked**: The failure occurred but was detected internally. Countermeasures were executed to deliver expected results to external entities.

Thereby, crash failures as defined by Laranjeira et al. [142] are split into two sub-categories based on the recoverability of the service state. An unrecoverable state is considered more severe. Whenever a service response is not meeting the expectations of external entities, it is considered to be degraded. No sub-categorization based on the degradation type is done. Masked failures represent a contradiction to the general definition of failures by Avižienis and Laprie [14] since external entities are not aware of a discrepancy between the received and expected results. However, enabling the awareness of what external entities expect allows the detection of expectation deviations before it is externally observed. Without internal detection and subsequent masking, the discrepancy would be observable to external entities. Cotroneo et al. [65] further provide a gradual definition of performance anomalies. Thereby, the severity of a failure depends on the degree of discrepancy between expected and delivered results.

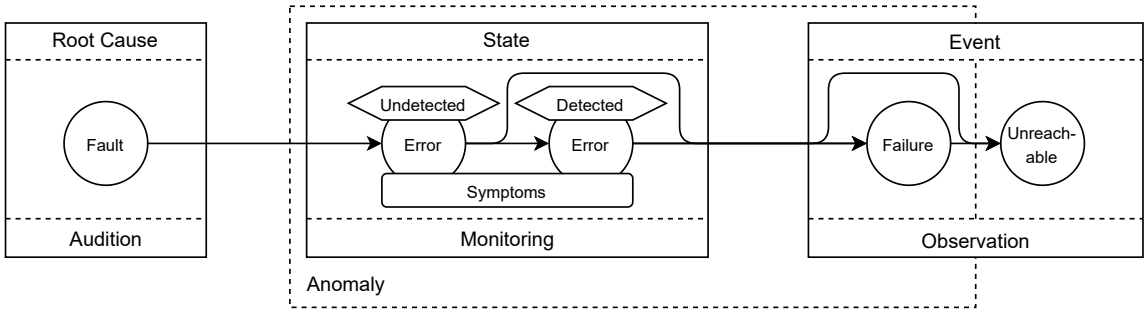


Figure 2.3: Failure model based on definitions from [141] and [144].

Failures are events and as such, consequences of defective states of system components [1, 141, 202]. The defective states are referred to as errors. An error must not necessarily result in a failure. As long as the service can be provided to external entities without deviations, a failure has not occurred. The origin of errors and failures are faults. These are system defects that do not influence the system up until certain conditions are met. A software bug in the source code is a fault that becomes an error if the code branch which contains it is executed. Damaged sections of a disk do not have an effect until data is written to or read from it. Figure 2.3 shows the fundamental relationships between faults, errors and failures [141, 144, 202]. The role of each component is represented at the top of the boxes. The tools required to detect them are shown at the bottom. An arrow connecting two elements represents a causal relationship. Faults are expected to be existent but do not influence the runtime behavior of the system. Therefore, they must be actively searched for to detect them. This procedure is referred to as audition [202]. Errors are states of running system components. The transition from a normal state to an error results from a fault that was triggered by encountering certain conditions. They can be either detected or undetected. The detection of errors is realized via the monitoring of system components. Specific deviations within the monitoring data are referred to as symptoms. Analyzing monitoring data to identify symptomatic patterns allows the general detection of an error state.

This failure model assumes the observability of all components to determine if an error state is causing failures. However, the effect of errors often cannot be observed. User-service providers usually cannot observe how their users perceive the requested service results. Components in cloud and fog computing systems are not accessible to all stakeholders. Deployed services usually cannot be accessed by cloud computing providers. Devices that are connected to the carrier network are not directly observable by network providers. Due to legally defined SLAs between the stakeholders, crash or corruption failures are noticed. Nevertheless, for other failures, the distinction between errors and failures cannot always be made. Recent publications introduce the term *anomaly* and use it for errors and failures where components are still reachable but are degraded or masked [5, 210, 241].

The above failure model considers one system component and its effects on external entities in the form of failures. Cristian [66] is extending this by considering the propagation of anomalies in distributed systems. System components that rely on each other's service results can become anomalous due to failures of dependant services. Thereby, the inability to cope with anomalies of dependent services represents the fault, and the occurrence of such anomalies is the condition that causes their failure (or anomaly). It can result in a propagation throughout interdependent services of a distributed system, entailing to recover several components. The identification of the source of this propagation is referred to as the root cause (RC) and the identification of it is named root cause analysis (RCA) [2, 240].

Anomalies are caused by different faults, which implies the necessity of different mechanisms to handle them [215, 242]. Resolving resource leakage is handled differently than a connectivity issue or overload of components. The detection of anomalies requires human experts to analyze monitoring data and search for symptoms. Thereby, recognizing fault-specific symptom patterns allows them to compile remediation operations and execute them to resolve the anomaly. This process highly depends on the knowledge and experience of these experts [29, 48]. This process entails the possibility to automate the selection of remediation operations by analyzing fault-specific symptom patterns.

2.3 Fault Tolerance

Distributed systems consist of different interconnected components and interdependent services. Like any IT system in general, such systems suffer from reduced dependability [175, 178, 223]. Components must be expected to contain faults, and thus, anomalies can occur, eventually resulting in an unreachable component. To maintain systems in an operational state, concepts to handle anomalies and unreachable components are required. In literature, this is referred to as *fault tolerance*. Avižienis et al. define four general fault tolerance categories to achieve this [14, 15]. The *fault prevention* includes methods to avoid the presence of faults in a system. These can be realized via policies to prevent programmers from writing code that contains bugs or to introduce generally standardized processes during the construction of hardware components. Although prevention can reduce the number of faults in a system, it is considered impossible to eliminate their existence [14, 225]. Aside from other factors, this is mainly due to the complexity of systems, their interconnection and potential incompatibility, and the number of parties involved in the development process. Faults are assumed to be present even in critical system domains such as spacecraft [86], medicine [125] or nuclear power industry [43], where fault prevention mechanisms are highly sophisticated. Employed fault tolerance methods aim

to prevent faults from becoming visible to external entities, i.e., the failure event should be avoided, masked, or its severity mitigated. *Fault removal* techniques deal with faults that are identified at runtime or during tests and remove them by adjusting the system. This is done via software updates or by using a different material when building hardware components. Analyzing systems and the historical occurrence of faults enable *fault forecasting*. Considering different properties of components like complexity, interoperability, or the number of involved parties during developments, allows estimating the number of faults that it eventually contains. Such estimations can be used to determine resources that are required for fault prevention during the development process and fault tolerance after deployment.

2.3.1 Fault Tolerance Policies

Our work is focused on fault tolerance, where faults are assumed to be a part of components in a distributed system. Whenever faults happen to cause anomalies or lead to unreadability of a component, operations should be selected automatically to reduce the MTTR. The work of Avižienis et al. [15] provides a taxonomy of fault tolerance methods which is further extended by Kaitovic and Malek [136]. Similar taxonomies are described in [81, 202, 225]. We align this taxonomy with the failure model explained in Section 2.2. The overview of the taxonomy is depicted in Figure 2.4. Fault tolerance policies are generally separated into two groups, *reactive* and *proactive*. Originally, these terms refer to failures of a system component. However, as previously explained, failure types where components are still reachable are often not observable. Therefore, we consider the remediation of anomalous components as proactive and the recovery of unreachable components as reactive.

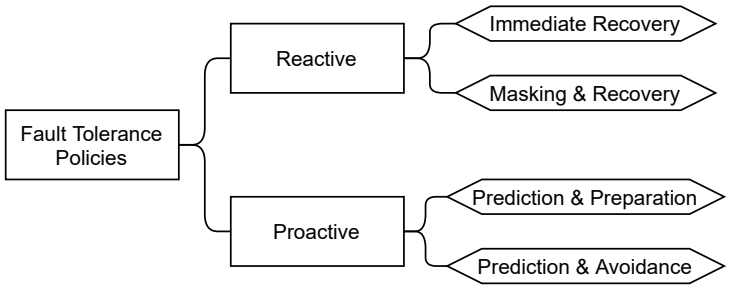


Figure 2.4: Taxonomy of fault tolerance policies. Adjusted from [136].

The reactive fault tolerance group is divided into two policy types: *Immediate recovery* and *masking & recovery*. In both cases, recovery operations are selected and executed but with different objectives. Former assumes operations to recover the unreachable component. Latter hides the unreachable component before recovering it, which assumes redundancy. During the recovery process, requests are relayed to other components that can serve them. Redundancy can be realized via load balancing between a set of replicas, cold or warm redundancy, or N-version programming [12].

The group of predictive fault tolerance aims at the prediction of failures where system components become unreachable. The prediction is either done by observing temporal regularities (e.g., periodicity) of failure events or analyzing monitoring data and searching for symptoms that indicate anomalies. Whenever failures are predicted, a fault tolerance policy is applied. Two general policy types are defined: *Prediction & preparation* and *prediction & avoidance*.

The prediction & preparation policy prepares the system for the unreachability of a component, usually aiming to reduce the MTTR. Examples are warming up spare components to speed up the switch-over when the primary component fails [60] or creating backups or checkpoints when a failure is predicted [11]. The prediction & avoidance policy tries to avoid the unreachability of a component. Balancing requests between a set of replica components allows reconfiguring the load balancer, such as no requests are assigned to the component for which a failure is predicted. This isolation allows selection and execution of remediation operations first and reassign it to handle requests afterward. Although the component might become unreachable during recovery, the overall service does not become unreachable because the replicas can provide it.

2.3.2 Temporal Analysis of Fault Tolerance Policies

We analyze the temporal progress of a system component's status to understand the implications behind the different fault tolerance policies. Therefore, Figure 2.5 illustrates two exemplary visualizations of a temporal progress. Thereby, the reactive and predictive fault tolerance policies are considered. A component can either have a normal, recovery, anomaly, or unreachable status. The anomaly and unreachable statuses are conforming the definitions provided in Section 2.2. The recovery status means that operations are executed for components that either become unreachable or to avoid their unreachability. A system that is being recovered might be unreachable (e.g., during an ongoing reboot) until the recovery operations are successfully finished. Furthermore, we separate the observed and real status of the component to distinguish detected and undetected anomalies. In the shown examples, it is assumed that the unreachability of a component is immediately apparent. Also, anomalies are causing a failure after a delay. In real systems, a component can become immediately unreachable. A reactive policy must handle such cases. Alternatively, anomalies must not cause a component to become unreachable at all but might cause degraded failures.

The top illustration shows the temporal progress of a reactive policy. Thereby, recovery operations are selected and executed when a component becomes unreachable. No monitoring data analysis to search for anomaly symptoms is employed. After anomalies occur, the component remains in an undetected anomaly state until it eventually becomes unreachable. Alternatively the anomaly can be reported by external entities which receive results that deviate from the expected norm. It remains in an unreachable or anomaly state until recovery operations are selected and executed. Applying this policy entails that the availability and eventual SLAs are affected. To reduce the impact on those, the time to select and execute operations must be reduced. This reduction translated to a shorter time that a component is unreachable and, thus, to overall higher availability. We state that this can be achieved by automatic selection of recovery operations.

Predictive fault tolerance policies are either analyzing the periodicity of past failures or the monitoring data to identify symptoms of anomalies. Such anomalies are assumed to either put the component into a degraded failure state or imminently result in unreachability. After the occurrence of an anomaly, the detection of its symptom patterns enables the alignment of the real (green line) and observed (red line) status. After that, it is possible to select operations that either prepare the system for the unreachability of this component or avoid its unreachability. It is favourable to select and execute these operations before the component fails. Otherwise, a reactive policy must be applied. After the detection of an anomaly, it is hard to predict when the

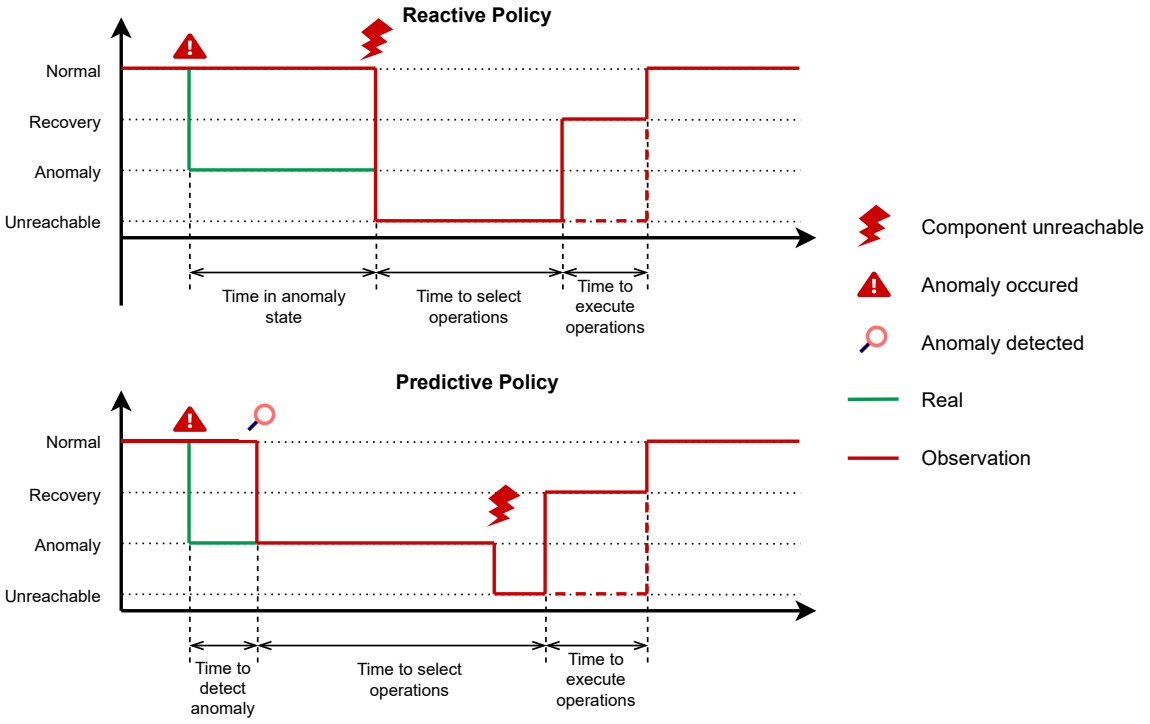


Figure 2.5: Comparing temporal progress of different fault tolerance policies.

component will fail [93]. Therefore, the available time for selecting and executing operations is not precisely known. Also, the impact of degraded failures can have severe consequences [132]. Therefore, the fast selection and execution of operations are crucial when predictive fault tolerance is applied.

It might seem obvious to favor the predictive over the reactive fault tolerance policy. However, the selection of a policy depends on the use case [34, 87, 136, 141]. We want to highlight the following aspects [4]:

- ◇ The overall assumption behind a predictive policy is that the negative effect on the availability of a component can be reduced by executing preparation or avoidance operations [15, 136]. If this cannot be achieved, employing a predictive policy cannot be justified.

- ◇ The monitoring of system components and the execution of analysis methods to identify anomaly symptoms causes computational overhead [110, 224]. Furthermore, it introduces additional complexity into the system [109]. On the one side, the eventual detection of anomalies enables the selection and execution of precedent preparation or avoidance operations increasing the overall system availability. On the other side, this increases the overall system complexity and computational overhead. These aspects need to be weighed against each other when choosing a fault tolerance policy.

- ◇ Detecting anomalies is imperfect. Whenever anomalies remain undetected until the component becomes unreachable, predictive policies lose their benefit. Such cases are handled as if a reactive policy is employed, but with increased overall system complexity and

[4] The list is not claimed to be complete but gives an overview of aspects relevant to this work.

computational overhead. Contrary, the false detection of anomalies can trigger the execution of unnecessary preparation or avoidance operations. Therefore, using a predictive fault tolerance policy depends on the correct detection of anomalies.

- ◇ Whether to prefer operations that prepare for or avoid the unreachability failure depends on the use case. After detecting an anomaly, it is challenging to predict the time until the component becomes unreachable. Replacing hardware devices that show anomalies before they fail can be expensive depending on the consequences of a failure and the time between anomaly detection and failure [235]. When components are rebooting, they are unavailable until the reboot is completed and potentially result in lost optimization-related states like caches or prefetching policies [47]. Such aspects must be considered when selecting operations, making the selection itself difficult.

Reactive fault tolerance policies require a fast selection and execution of recovery operations to minimize the negative impact on availability. The ability to detect anomalies is the prerequisite to apply a predictive fault tolerance policy. Anomaly detection methods are not perfect, and mispredictions must be assumed when operations are selected and executed. The selection of operations defines whether to prepare for or to avoid the unreachability of system components. This choice depends on many aspects that are hard and time-consuming to consider for human experts.

Chapter 3

Related Work

In this thesis, we develop methods for anomaly symptom recognition that enable the automatic selection of operations to resolve anomalies. On a high level, this can be categorized as a fault-tolerance method since anomalies and failures are assumed to occur. Automation should accelerate the process of resolving anomalies and failures and mitigate negative consequences. Therefore, we give a brief overview of the history of fault-tolerant IT systems and analyze the methods. Based on this, we infer the requirement of general and holistic approaches and analyze the concept of autonomic computing. It provides a variety of aspects where autonomy for IT systems is required. We focus our review of related literature on the aspect of self-healing and present frameworks and systems. The area of AIOps is concerned with researching the application of methods from artificial intelligence and machine learning to support IT system operation. We provide an overview of this research direction, focusing on symptom pattern recognition and automatic operation selection methods.

3.1 A Brief History of Fault-Tolerant IT Systems

Due to the growing size of distributed systems and increasingly strict criteria defined within SLAs, human operators require support to operate these systems [123]. However, the necessity to implement fault tolerance into IT systems exists for an extended period. Early works focus on the theoretical fundamentals of IT system reliability [21, 198]. Utilizing concepts from probability theory, they estimate the reliability properties of systems that consist of multiple components and predict eventual failures. Based on the work of Epstein and Sobel [77], Hosford [120] defines the term dependability as the probability that a IT system is operable when needed and provides quantitative measures. He further uses the terms *availability* and *reliability* and proposes formal definitions for both. Brian Randell [191] introduces modularity as a key requirement for fault tolerance of complex IT systems. The modularity allows the separation of a distributed system into fault blocks. He analysis different constellation and segmentation strategies and defines an analytical model to express the ability of a system to tolerate faults. Algirdas Avizienis provides a qualitative discussion on fault tolerance, system faults, and fault-tolerant design concepts [16]. He further investigates design concepts for IT systems that support fault tolerance [13]. His and the work of Laprie [14, 141] define failures in IT systems together with a general failure model and taxonomy of fault tolerance methods.

Early implementation of fault tolerance in IT systems can be found in critical application

fields like aircraft [67, 237], spacecraft [193], or military [167]. Gray and Siewiorekkey [100] define concepts and techniques to build highly available systems. They propose five key aspects that can be utilized to improve the fault tolerance of IT systems: modularity, fail-fast modules, independent failure modes, redundancy, and repair. In the area of fault-tolerant operating systems, Anita Borg's publications are well recognized. She develops a distributed UNIX operating system with a focus on fault tolerance [32, 33]. Redundancy is the primary approach to realize fault tolerance for hardware components of a system, Najjar and Gaudiot [169] analyze the fault tolerance of networks and propose concepts to design network topologies that tolerate the outage of single nodes. Their fault-tolerant topologies proclaim a design that contains redundant routes between network endpoints. Works on redundancy distinguish cold, warm or standby, and hot redundancy [216]. Thereby, the respectively redundant components are either turned off, in standby mode, or turned on, with respective implications to energy consumption and switch-over times. Early work in the area of fault tolerance in software systems concentrates around executing a task multiple times. Chen and Avizienis introduce the term *N-version programming* whereby N>2 equivalent programs with the identical initial specification are started [52]. A decision algorithm is required to evaluate the N results and make a final selection. Variants of this have been investigated for tasks like matrix operations [122], or the calculation of the invariant distribution of a Markov chain [142].

It can be seen that the focus of fault tolerance research shifted from a theoretical perspective towards practical and implementable solutions. This process correlates with the increasing availability and usage of IT systems. Thereby, an initial strong focus lies on redundancy which should mitigate the effects of occurring failures.

3.2 Fault Tolerance Methods

The growing availability of IT systems accelerates the pace at which data analysis platforms and digital applications are developed. On the one hand, the system complexity is considerably growing, but on the other hand, this comes with powerful hardware, which allows the employment of increasingly sophisticated fault tolerance methods. In related work, mostly three different fault-tolerance policies are investigated.

1. A reactive policy handles failures when they occur. Most works that employ redundancy can be categorized as reactive.

2. The periodic policy executes operations in regular intervals, where the interval is a configuration parameter and subject to optimization. A periodic failure immersion is assumed to realize this policy [115].

3. A proactive policy employs models to predict failures and handle them before they occur. An antecedent system state deviation, referred to as degraded state [39] or anomaly [64, 71, 90, 172, 173, 188, 238], is assumed to be recognizable. This enables the preparation of the system for failure or the execution of countermeasures to avoid failure.

Castelli et al. [46] compare the impact on system availability and component downtime between the three fault-tolerant policies. They apply the reactive policy as the baseline and show that the periodic policy could reduce the downtime by 25 %, whereas the proactive fault

tolerance policy achieves a downtime reduction of 60 %. Egwutuoha et al. [76] compare a reactive and proactive fault tolerance policy. They are monitoring the system health of compute nodes that host virtual machines. When detecting a degraded health status, VMs were migrated to a healthy node. By applying the proactive policy, the execution time of the application deployed within the VMs could be reduced by up to 30 % compared to the reactive policy. By applying Markov models, Eckart et al. [75] implement a combination of fault prediction and proactive fault tolerance. They analyze simulations of different RAID configurations and show that the mean time to data loss can be extended by up to 85% compared to a reactive policy. Instead of avoiding a failure, other approaches are preparing the system for the occurrence of the failure. An example is a periodic checkpointing of a running application's state. Whenever failures are predicted to occur, a checkpoint is computed to reduce the re-computation effort. Experimental evaluations of such an approach conducted by Bouguerra et al. [35] shows a performance-boosting of an HPC system by 30% while introducing only 6% of additional overhead.

Several works assume an ideal failure prediction model for proactive fault tolerance either implicitly [46, 76] or explicitly [75]. However, based on the imperfection of failure prediction models, these assumptions limit the evidential magnitude of the results and conclusions for practical applications. Since this imperfection is not considered, the presented performance improvements might differ when applied in productive environments. Occurring false predictions will decrease the improvement or might even fall below the performance of a reactive fault tolerance policy, which is shown by Aupy et al. [11] They address the impact of an imperfect failure prediction for proactive fault tolerance. Simulations under varying model parameters reveal that proactive approaches are not always the best option. A frequent false prediction of failures results in an increased overhead due to the execution of unnecessary operations. Therefore, analyzing the prediction performance of the models is an essential step towards practical applicability. By combining proactive migration and reactive checkpointing, Lan and Li [140] define the runtime of an application as the performance criteria for their evaluation. They manage to decrease the execution time by 27 % when the model perfectly predicts the failures. However, decreasing the prediction performance results in an increased runtime by up to 10 %.

The importance of the failure prediction correctness is shown to impact the performance of a fault tolerance policy significantly. Therefore, it is important to consider the ability to predict failures when applying predictive fault tolerance correctly. Salfner and Malek [204] present an availability model that includes prediction model performance measures. They aim to optimize the availability of a system by analyzing the impact of the model performance measure. It quantifies the model performance required for a proactive fault tolerance policy to outperform a reactive policy. Similar work is done by Kaitovic et al. [136]. They extend the proposed model of Salfner and Malek and include additional model performance parameters. A formal definition of a break-even point is given, which provides a quantitative answer to whether to use a reactive or proactive policy. Their results show a significant impact of false predictions on the availability of systems managed by proactive fault tolerance. Matos et al. [157] analyze a complex availability model having 14 parameters. By applying a parametric sensitivity analysis, they identify the parameters with the most substantial influence on availability. Besides that, they consider monetary costs for normal, degraded, failed system states and present optimization results based on cost-related parameter values.

Although depending on the ability to predict failures correctly, fault tolerance research focuses on the proactive fault tolerance policy. Thereby, increasingly precise methods to detect

anomalies are developed. However, the described proactive methods focus on specific failures. Whenever its occurrence is predicted, a defined operation is executed to avoid it or to prepare the system for its occurrence. The complexity of IT systems entails a variety of components that can be affected by different failures. This complexity requires the execution of different operations to resolve imminent failures. A more general view of fault tolerance in IT systems is needed.

3.3 Autonomic Computing

In 2001, IBM addressed the problem of generality for IT system fault tolerance. They raised the initiative of *autonomic computing* [119] and proposed their vision of IT systems that are self-managed [138]. The inspiration of autonomic computing systems is drawn from biology, more precisely, the autonomic nervous system. The analogy implies that IT systems should execute their tasks entirely without external - i.e., human - intervention, including the continuous adaption to environmental changes and the handling of occurring problems.

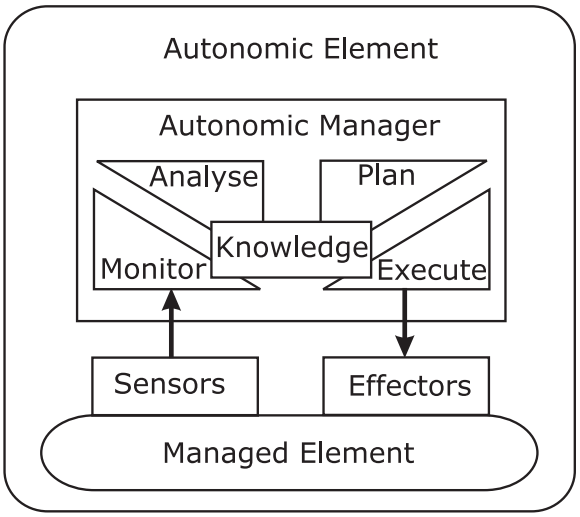


Figure 3.1: The MAPE-K closed loop for autonomic computing [123].

To achieve autonomic computing, IBM [126] design a closed-loop control. Later, it is referred to as MAPE-K [158]. Figure 3.1 shows the MAPE-K architecture. It is composed of an autonomic manager and a managed element. In our work, we refer to the managed element as the system under operation (SuO). The manager takes the role of a controller that accesses the sensor interfaces to assess the state of the managed element, performs analyses, and takes action if needed. This workflow of the manager contains four steps. First, monitoring (M) is realized by accessing the sensors of the managed system. Second, collected monitoring data are analyzed (A) to recognize indications that necessitate taking action. Third, operations are selected, and an execution plan (P) is compiled. Fourth, the effector interfaces are accessed to execute (E) the operations. Over time, the autonomic manager should improve and operate the managed element more efficiently. Knowledge (K) management is required to improve. It is continuously updated and central for future analyses and control decisions.

The employment of methods for analysis, planning, and execution depends on the objectives of an autonomic computing system. The vision is that every aspect in a life-cycle of an IT system (e.g., deployment, configuration, maintenance, or protection) should be executed autonomously. However, it is recognized that this is a vast field of problems that requires separation into sub-problems. In their original publication of autonomic computing [119], the authors defined four attributes of self-management: self-configuration, self-optimization, self-healing, and self-protection.

The self-configuration addresses the dynamics of complex IT systems. Whenever the environment changes, an autonomic configuration adjustment should be applied to ensure the continued operability of the system. This is often applied for wireless networks [121, 168, 184] and the Internet of things (IoT) [10, 51]. Wireless network connectivity is susceptible to environmental disturbances, which necessitates a fast adaption. However, the adjustment to environmental disturbances always comes at a cost, which could be energy, computation resources, or any monetary cost. Therefore, the self-configuration is often coupled with self-optimization [121, 184]. It results in the objective to keep the system operational but adds constraints regarding defined cost metrics. Besides environmental changes, IT systems are always prone to external threats. It is stated that the identification and protection against various types of attacks are required to enable autonomic computing. The attribute of self-protection is defined to cover this problem field. However, IT system security is a huge industrial and public research area, even before autonomic computing [8]. The attribute self-healing is defined to enable fault tolerance. IBM describes this by discovering problems or potential problems and finding alternatives of using available resources or applying reconfigurations to keep the system functional.

Overlappings exist between the objectives of the four attributes. Self-configuration, protection, and healing allow IT systems to react to occurring problems. The difference lies in the origin of the problems: external environmental disturbances for self-configuration, malicious external or internal attackers for self-protection, and inherent faults in soft- and hardware with the potential to cause anomalies or failures for self-healing. Self-optimization serves as a constraint that should prevent the system from excessively using resources to resolve the different problems. Our work is related to self-healing and considers self-optimization.

There are three well-recognized surveys on self-healing [94, 190, 213]. Ghosh et al. [94] give an overview of publications that are related to self-healing. Psaier and Dustdar [190] provide an extensive analysis of autonomic computing with a focus on self-healing before presenting self-healing publications based on their grouping into nine categories. Schneider et.al. [213] present a categorization of self-healing systems based on the management style, computing environment, and learning methodology. The published approaches can be further categorized as invasive and non-invasive. Former assumes that self-healing is part of a system itself and must be integrated into it during the development process (e.g., use respective materials for hardware or having self-healing within the source code of an application). Latter is realized via an additional self-healing system that passively monitors and operates the managed system by applying adjustments like reconfiguration or redeployment when required.

An example of an invasive approach is given by Dashofy et al. [70]. They provide an architecture of a development framework for event-based software systems with inherent self-healing. Further, guidelines are presented to enable the efficient self-healing of software systems. For the implementation, the software is described via an abstract XML-like language. After deployment, it is possible to adapt the XMl descriptions, whereupon changes are applied

to the application. Miorandi et al. present Embryo-ware [163, 164], which is inspired by biological reproduction, healing, and immunity. Thereby system components are provided with meta-information that is comparable to genomes. Genomes are replicated among all components. They contain definitions and constraints regarding the tasks of a component. Failures are compensated by reassigning the tasks of a failed system part among the remaining components.

In contrast to invasive methods, Garlan and Schmerl [92] present a non-invasive approach by using an adaptation of the managed system. In other publications [55, 91], they further extend the architecture and refer to it as *Rainbow*. Thereby, one or more models of a system are generated and adapted at run time. Based on these models, human experts can define invariants which are constraints for the collected system information. Violation of invariants is regarded as anomalies that require operations to resolve them. The architecture of Rainbow is related to this thesis. The main differences are the employed methods for recognizing anomaly patterns in the monitoring data and the selection of operations based on this. While Rainbow utilizes a constraint-based approach, we apply pattern matching to recognize anomaly symptom patterns and couple this with an automatic operation selection.

The systems presented so far focus on self-healing architectures, propose domain-specific solutions, or assume an adaption of the application source code. We conclude that additional generality of methods is required to enable fully automated IT system operation to cope with the fast pace at which these systems are developed.

3.4 AIOps

Autonomic computing requires data analysis methods. Thereby, the state of SuO components needs to be observed, and the monitoring data must be analyzed. Whenever deviations from an expected norm occur, operations should be compiled and executed to transfer it to a normal operation mode again. This process is realized as a set of data analysis methods taking system monitoring data as input and eventually executing operations to resolve anomalies or realize runtime optimization [17, 95, 213]. The recent advances in machine learning and artificial intelligence inspire the application of methods from these areas to analyze IT system monitoring data. The company Gartner, Inc. initially named this research direction *Algorithmic IT Operations* in 2014. In a later publication (2018), they renamed it to *AIOps* (artificial intelligence for IT operations) [97]. Due to their complexity, heterogeneity and increased distribution, AIOps methods are especially researched for systems like cloud [89], fog [18], and edge [133] computing, network function virtualization (NFV) [61], or high-performance computing [171].

Several publications are further investigating the general implications, opportunities, and challenges of AIOps. Dang, Lin, and Huang [69] analyze and motivate the utilization of AIOps methods to enhance fault tolerance of IT systems. They further identify challenges and innovations that AIOps will bring. A book written by Masood and Hashmi [156] discusses the topic of AIOps, how it differs from traditional IT system operation, and the potential to improve fault tolerance. The idea of AIOps is picked up by the industry as well. Big IT companies and startups apply AIOps solutions to their systems and add them to their product portfolio. Qu and Ha provide insights into the software development and system operation processes of Baidu [1] and explain how they envision the employment of AIOps. Microsoft [155], Amazon [2],

[1] `https://www.baidu.com/` (last access 22 May 2021)

[2] `https://aws.amazon.com/de/devops-guru/` (last access 22 May 2021)

and Google [97] are applying AI and ML to increase the dependability of their systems and offer such services for their cloud platforms allowing customers to integrate them with their services. Moogsoft [3] is a startup with a specific focus on AIOps. They offer solutions for system monitoring, anomaly detection, root cause analysis, log data analysis, and several others. They give insights into the models they are using, including clustering, probabilistic sequence analysis, and metric correlation analysis. Dynatrace [4] offers automatic discovery of system component interdependencies coupled with different analytical models to detect anomalies or to identify root causes. Several other projects such as BigPanda [5], Logz [6], or Resolve [7] offer similar product portfolios.

With AIOps systems, it should be possible to realize robustness against anomalies based on non-specific system data like metrics [64, 95, 239], or enable the processing of multiple data sources [31]. Thereby, domain-specific assumptions are reduced to realize generality. Several methods are published in this research direction including anomaly detection [64, 90, 172, 173, 188, 238], root cause analysis [114, 135, 148, 208, 233, 239], symptom analysis [30, 54, 94, 131, 137, 174], and operation selection [50, 128, 139, 145, 161].

Anomaly detection in the context AIOps should analyze monitoring data and recognize if SuO components deviate from the expected operational state. Recent solutions focus on one-class methods that utilize only the normal system states for training [64, 173, 238]. When deployed, they should recognize deviations from the trained norm and raise alarms. This concept is referred to as binary, two- or bi-class anomaly detection since these methods report either a normal state or an anomaly. Published solutions for this type of anomaly detection rely on fundamentals of stochastic theory [64, 188], classical machine learning [238], or deep learning [90, 173]. Moreover, the analysis is applied on different monitoring data, which are traces [172], logs [73], and metrics [173, 238].

IT systems are undergoing a transition from monolithic architectures to microservices enabling advantages such as resilience, agility, and scalability [176]. These are deployed on cloud, fog, or edge computing systems, resulting in complex and distributed environments. The interdependencies in such systems result in propagating anomalies, which means that an anomalous component can cause anomalies on dependent components, eventually resulting in a cascade of alarms [114]. Root cause analysis methods are concerned with the identification of the node that initially became anomalous. Again, the RCA is applied on different monitoring data sources, i.e., traces [148], logs [135], and metrics [208, 233, 239]. Most RCA follow a similar workflow. First, a graph structure is constructed based on the available normal data. Whenever anomalies occur, a subgraph is acquired containing all components that are reported as anomalous. Based on different graph search methods, a single node is identified in the subgraph. This node represents the anomalous component.

Anomaly detection and root cause analysis are essential parts of a AIOps system. However, additional analyses are required to enable autonomy for the automatic resolving of anomalies. We focus on the recognition of symptom patterns to identify anomaly types. Based on this, an operation to resolve the anomaly should automatically be selected.

[3] `https://www.moogsoft.com/` (last access 22 May 2021)
[4] `https://www.dynatrace.com/` (last access 22 May 2021)
[5] `https://www.bigpanda.io/` (last access 22 May 2021)
[6] `https://logz.io/` (last access 22 May 2021)
[7] `https://resolve.io/aiops` (last access 22 May 2021)

3.4.1 Anomaly Symptom Analysis

The origin of anomalies are faults in hard- or software components. When faults become anomalies, their symptoms are assumed to be observable in the monitoring data. Different faults require respective operations to be resolved [94]. Therefore, the symptom analysis is concerned with recognizing patterns in the monitoring data based on which the anomaly type that represents a specific fault should be inferred. This is often approached as a classification problem, whereby different anomalies are classes that should be predicted by a classification model based on available monitoring data.

Islam and Manivannan [131] propose a model for sequential metric data analysis to predict failures in the Google cluster workload trace. Based on the memory and CPU utilization metrics, they predict the failure codes of executed jobs and tasks. Furthermore, they utilize a cloud simulation system to recreate memory and CPU utilization metrics of executed jobs and tasks. Predicting return codes instead of anomaly types is owed to the missing labels in the dataset that they used. It is questionable if different return codes reflect the whole set of existing anomaly types.

Bodik et al. [30] referring to the classification of various data center crises as data center fingerprinting. Thereby, system resource metrics from all data center components are aggregated via quantile discretization, and feature selection methods are applied to choose only relevant metrics for distinguishing different data center crisis types. The reported results were achieved by an aggregation of system resource metrics collected over 30 minutes. For distributed system environments, the central aggregation of metrics from all system components limits scalability. Furthermore, the requirement of 30 minutes aggregation time windows results in a significant delay between emerging anomalies and their classification.

Kajó and Nováczki [137] provide a comparison of different machine learning algorithms together with a genetic algorithm approach. Given monitoring data from a System Architecture Evolution (SAE) core network, they select an optimized metric subset used as input for different classification algorithms. The focus lies on metric selection, whereby each metric is averaged over a 15 minutes interval. The delay prevents a timely selection of operations from resolving an anomaly. Furthermore, the models are trained on a set of 850 anomaly observations. A sparse occurrence of anomalies is expected, and thus only a few available examples for each anomaly type must be expected. The requirement of many training data poses a limit for practical application.

Another work of Cheng et al. [54] applies a multi-scale long short-term memory model to classify four different anomalies during update messages of the border gateway protocol (BGP). They use a sequence of collected metrics during each of the four anomalies and the normal system state. For model training, several hours of anomalous metric data are expected. However, the occurrence of anomalies is usually time-limited. The objective is to transfer the system back to a normal operation state as fast as possible. The expectation of long consecutive anomalous metric sequences limits the applicability.

Netti et al. [174] test four machine learning methods to classify eight anomaly types in an HPC cluster node. They use 144 metrics from the node, which are sampled at each second. 22 features from each metric series are extracted based on 60-second windows with a stride of 10 seconds. The machine learning methods are applied individually on each metric. Evaluation is done separately for each model and averaged across all models eventually. The employment of 144 models per node poses a limitation regarding scalability. Prediction with and potential

retraining of a large number of models requires a significant amount of computation resources.

Anomaly type symptom recognition is needed to infer the anomaly type of observed symptom in the metric data and couple this information with operations to resolve the anomaly. However, the precise recognition of patterns in metric data is hard to achieve. Existing methods either require many examples to train the models or apply a fine-grained separation of metric data into sub-metrics for which respective models are utilized. A system should use only a small fraction of available resources for monitoring data analysis. This requirement restricts the number of models that can be used to analyze the data. Furthermore, IT systems are dynamic and constantly change, which entails the occurrence of novel anomaly types. An anomaly symptom recognition method should be able to recognize yet unknown anomaly types. None of the presented methods address this issue.

3.4.2 Automatic Remediation or Recovery

This thesis aims to enable the automatic selection of operations to resolve anomalies. It is achieved by inferring anomaly types based on symptom patterns, which are coupled with specific operations. Therefore, we want to investigate related approaches and analyze how the automatic remediation or recovery of system components is achieved. In many publications, methods are proposed that resolve one specific anomaly type [11, 35, 46, 75, 76, 140, 149]. Thereby, it is implicitly assumed that either a single anomaly type occurs on the component or that all anomaly types that occur can be resolved by executing the same operation. Often, the timing of the execution is subject to optimization, or the system is adjusted in a way that accelerates the execution of the operation. Examples of such approaches are minimizing the time of a reboot operation for containers or virtual machines [45], efficient estimation and adjustment of resource provisioning in cloud systems [152, 170] or optimal checkpointing of data processing jobs [140]. In the case of checkpointing, a high overhead due to frequent checkpoint should be balanced with the time that is required to recompute the results when a failure occurs [11]. Another example is to hit a favorable moment for a restart of a component [46] or the ahead of time provision of virtual resources to decrease the time until they are operational [149].

Research has also been conducted on system disaster recovery. After hazardous accidents, the system is typically redeployed in the same or an alternative location. Pokharel et al. [189] present an approach based on geographic redundancy of distributed system components. By employing a Continuous Time Markov Chain, they analyze the overall system reliability in dependence on the reliability of individual system components. However, having full redundancies of every single component is expensive and requires more computation resources, eventually leading to increased energy consumption. A reactive approach is expounded by Lavriv et al. [143]. They define disaster as a phenomenon that puts the system into a state where it cannot be recovered from. A redeployment is required in this case. They compare a manual redeployment with a scripted redeployment workflow defined with infrastructure as code (IaC) scripts and show that the latter reduces the time until the system is operational again. Chang [50] investigate disaster recovery of Big Data systems across three different data centers and show that their methods of data distribution and replication significantly accelerate the recovery process of one data center. They rely on backups and four complementary retrieval methods. Although enabling automation of the recovery process, disaster recovery deals with a different problem than our approach. Hazardous accidents are immediately evident and do not require sophisticated monitoring data analysis methods. Also, the time frames until the system is expected to

be operational again differs as well. After an outage of the system part or the whole system, the redeployment is expected to require an extended period (hours or even days). In our case, anomalies should be resolved much faster, whereby mitigations within minutes or seconds are expected.

We use three categories for the further related publications.

Rule-based methods allow human experts to define rules based on IF-THEN expressions [139, 161]. The IF-part matches defined policies, e.g., alarms from specific components or recognized anomaly symptom patterns. The THEN-part is connected with the conditions of the IF-part and executes operations if the condition matches. Platforms exist that connect to alerting or monitoring systems to receive data that can be matched by IF-statements and support a variety of tools to execute operations [8][9]. We see these platforms as complementary to our proposed method. The detection of anomaly type symptom patterns can be defined as conditions via IF-statements to trigger the automatic selection of operations with THEN-matches.

Learning-based methods use reinforcement learning to infer remediation and recovery operations without human intervention [128, 245]. Thereby, a trial and error policy is applied. The methods are provided with a pool of commands, out of which sequences of commands are composed and executed against anomalous system components. The effects are observed and considered successful if the system reaches a normal operational state after executing a command sequence. Although this enables AIOps systems to resolve anomalies automatically, reinforcement learning has three disadvantages. As stated by Zhu et al. [245], a growing amount of available commands results in exponential growth of the search space. Reinforcement learning methods require thousands of trial and error attempts to find a command sequence that resolves the anomaly. A guided search based on heuristics can be applied to mitigate this. However, even a guided search relies on trial and error to recover from an anomaly. System operators are very conservative when it comes to the execution of commands with the unpredictable outcome and is regarded as an infeasible system in productive deployment [105, 107, 218]. Additionally, a sequence of commands must exist in the search space. Otherwise, the reinforcement method will not be able to find a solution. Therefore, human experts need to verify that the reinforcement method can find a solution by checking if such a solution exists. It is contradictory to let an algorithm search for a solution which existence is already known to the expert.

Case-based methods utilize the occurrence of previous anomalies or failures as cases and match them against future occurrences [145, 165]. These can be automatically resolved by executing respective operations. Resolving occurred cases works under two central assumptions. First, anomalies and failures, i.e., cases in this context, must reoccur. Second, operations that were executed for past cases will resolve the future case as well. Based on this, it is possible to couple past anomalies or failures with the operations used to resolve them, which allows the automatic selection of these operations for future occurrences. Thereby, the challenge is to recognize anomalies or failures. Publications from Montani and Anglano [165] as well as Li et al. [145] require service-specific metrics like responses to HTTP requests or application error messages to match the cases. We propose a method based on anomaly type symptom patterns in system metric data to recognize reoccurring anomalies and match theses with a set of operations to resolve them.

[8] `https://stackstorm.com/` (last access 21 May 2021)

[9] `https://www.rundeck.com/` (last access 21 May 2021)

Chapter 4

Anomaly Symptom Recognition in an AIOps System

This chapter introduces the analysis of anomaly type-specific symptom patterns to automatically select operations and resolve anomalies in components of a distributed system. First, we describe the role of symptom pattern recognition and its enabling of automatic operation selection in an AIOps system and analyze the challenges and assumptions. Second, an overview of a closed-loop control is given. We introduce a set of modules and their respective tasks to realize an automatic recover of anomalies and highlight the focus of our thesis. The framework architecture of our proposed ReCoMe engine is presented and explained in detail. It contains consecutive modules to enable system metric data analysis, symptoms pattern recognition, anomaly type inference, and automatic operations selection. Third, we model the anomaly symptom recognition as a classification problem applied to monitoring data. Different modeling concepts are analyzed, and requirements for anomaly symptom recognition are inferred.

4.1 Challenges and Assumptions

Our work aims to accelerate the remediation and recovery of anomalous or failed system components by enabling an automatic operation selection. However, the operation selection cannot be considered a stand-alone problem. It relies on aspects like the applied fault tolerance policy, the ability to detect anomalies, the recognition of anomaly type-specific symptom patterns, or interfaces to the relevant system components. Therefore, we consider our approach in the context of an AIOps system. Enabling anomaly recovery automation for IT systems is a complex problem and usually split up into several subproblems whose solution methods can be combined [94, 154, 208]. Considered subproblems are the *monitoring*, *anomaly detection*, *root cause analysis*, *operation selection*, and *operation execution* [94, 109, 213].

Our methods are applied on a *System under Operations* (SuO), which can be any component of an IT system. The monitoring of SuOs is concerned with an efficient observation of monitoring data from relevant components [2]. Anomaly detection provides methods to raise alarms whenever a component state deviates from the known norm [206]. In distributed systems, anomalies can propagate. The component from which the propagation started is referred to as the root cause. Root cause analysis methods are concerned with its localization. If anomalies are detected, operations need to be selected and executed to resolve them. The propagation

entails selecting and executing respective remediation operations for every component - potentially in a specific order - to recover the SuO.

Anomalies can result from different faults and usually require respective operations to be resolved [215, 242]. Our anomaly symptom recognition methods assume a categorization of faults. They can be recovered by respective operations whenever faults lead to anomalies. However, anomaly states cannot be directly observed at runtime. Instead, their manifestation in the form of symptoms must be recognized by analyzing the monitoring data. We propose a respective categorization of anomalies based on their symptom patterns. Suchlike categories are referred to as anomaly types, and we refer to the assignment of anomalies to categories based on symptom patterns as anomaly symptom recognition. The recurrence of such anomaly types enables the automatic selection of operations to resolve them.

4.1.1 Challenges

We consider two types of challenges. The first type focuses on anomaly symptom recognition based on patterns in monitoring data. The second type refers to the implementation of anomaly symptom recognition and automatic operation selection into an AIOps system.

Due to the nature of complex IT systems, anomaly symptom recognition based on recurring symptom patterns is challenging. Cloud and fog computing systems consist of various components. Symptoms of the same anomaly types can differ significantly depending on the component they occur on [42]. Furthermore, system components are interconnected, and their states depend on several internal and external aspects, which affect the symptoms. We summarize the following central challenges for anomaly symptom recognition:

1. **Rareness of anomaly type examples**: In contrast to normal component states, anomalies are expected to occur rarely and have a limited duration. Therefore, anomaly symptom recognition must cope with few examples of anomaly symptoms when trying to recognize them. Identifying general similarities or dissimilarities of patterns in data based on few examples is considered a challenging task [28]

2. **Component heterogeneity**: We consider complex IT systems with a variety of components. Different service types like web servers, content streaming services, or identity management services run on virtual resources with different specifications like virtual CPUs or available memory. The sources of virtual resources are physical nodes with potentially different hardware specifications. These dynamics are represented as variations in observed monitoring data and pose a challenge to recognizing specific anomaly symptoms.

3. **Varying external factors**: External entities such as human end-users, other services, sensors, or effectors are expected to interact with SuO components. Furthermore, the SuO components are interconnected. These high interdependencies result in chaotic usage patterns [4, 41]. It affects the respective SuO components, their utilization and leads to variating patterns in their monitoring data.

4. **Incomplete knowledge**: Components of a SuO can change, or new components can be integrated over time. These dynamics entail the eventual occurrence of anomaly types

that were not observed yet. The knowledge base that contains all known anomaly symptoms must be considered incomplete at all times. It requires an anomaly symptom recognition method to admit that symptoms do not match any known anomaly types yet are unknown.

Both the component heterogeneity and varying external factors lead to variations of symptom patterns of the same anomaly type. We refer to this as the intra-type variation. This variation in combination with anomaly-type examples' rareness makes it difficult to recognize anomaly type-specific symptoms.

Several challenges need to be considered when integrating anomaly symptom recognition and automatic operation selection into an AIOps system.

- ◇ The incomplete knowledge assumption requires an approach to handle unknown anomaly types. Furthermore, a knowledge base that contains historical anomaly symptoms cannot always be assumed. The acquisition of anomaly symptom examples until automation can be realized is referred to as the cold-start problem. A concept on how to handle empty knowledge bases and novel anomaly types is needed.

- ◇ Integrating monitoring data analysis into an IT system must assume a limited availability of computation resources to execute data analyses. The main value of the SuO stems from the execution of regular workloads. The analysis must utilize a small fraction of the available resources to limit interference with the regular workload. However, the recognition of anomaly symptoms with a high intra-type variation requires computation-intensive calculations. A trade-off between both aspects is required.

- ◇ Anomalies can propagate in distributed systems and affect several components. Multiple anomalous components entail identifying the root cause and managing the order in which anomalies should be resolved. Such order requirement has implications for the automatic operation selection.

- ◇ Operations to resolve anomaly types depend on the anomalous component, the current system load, the preference of different human operators, and several other aspects. Therefore, an anomaly type can eventually be resolved by more than one operation. When assuming more than one available operation for an anomaly type, a concept to select an operation from a set of available candidates must be realized.

4.1.2 Assumptions

The **first assumption** is the existence of a preceding anomaly detection method that raises alarms whenever states of system components deviate from a known norm. The anomaly symptom recognition and operation selection are only executed when taking action is required, i.e., when anomalies are detected. Detection of anomalies entails that their symptoms must be observable. It can be the case that anomalies remain undetected. A reason for this invisibility might be the fact that the symptoms are present in data that are not observed or that the symptoms cannot be detected due to high noise caused by regular workload execution. Undetectable anomalies cannot be considered.

When several components are reported to be anomalous within a short time frame, a propagating anomaly must be expected. To reasonably manage the order in which the anomalies are

resolved, knowledge about the root cause is required. Therefore, our **second assumption** is the existence of a root cause analysis that can be requested on-demand to localize the root cause component from a set of anomalous components.

The anomaly symptom recognition groups anomalies based on common symptoms to automatically select recovery or remediation operations. Historical occurrences of respective anomaly types are expected to have been resolved by previously defined operations. Based on this, the **third assumption** is that anomalies of the same type occur multiple times.

Although it is often envisioned, immediate automation of operation selection is hard to achieve when handling unknown anomaly types. Different policies exist to select recovery operations for yet unknown anomaly types [94]. The trial and error policy aims at executing combinations of known operations until the anomaly is eventually resolved. This policy is problematic since the effect of operations under unknown anomaly states is challenging to foresee, and reverting these effects is not always possible. This approach can require a significant amount of time since many combinations of different operations exist. Furthermore, providers and operators of IT systems are very conservative when employing such uncertain methods within production [218]. As an alternative to this policy, we apply a human-in-the-loop approach. It aims to gradually transfer expert knowledge from the human into the AIOps system. Whenever anomalies are encountered for which no operations can be automatically selected, human experts are consulted. Therefore, the **fourth assumption** is a cooperation between the operation selection method and human experts whenever required.

4.2 AIOps System

To manage increasingly complex systems, human operators require automation and support. Such support can be achieved by employing different methods to monitor components, detect anomalies, or automatically select remediation and recovery operations. These methods are eventually combined in AIOps systems. Traditionally, human experts are notified by alarms whenever system anomalies are detected, leaving the selection and execution of operations to them [2, 93]. It requires them to analyze the symptoms, compile operations and execute them to resolve anomalies. This manual work is time-consuming, difficult, and partly repetitive. Therefore, an automatic selection of operations to resolve anomalies should serve as assistance to human experts. We propose the ReCoMe (recommendation and remediation) engine to achieve this by utilizing four steps: ingest control, symptom analysis, anomaly type inference, and operation selection. It is designed to be embedded into an AIOps system for compatibility with existing frameworks. The structural overview of the ReCoMe engine, together with its embedding into a closed-loop control, is depicted in Figure 4.1.

We assume a monitorable SuO and an anomaly detection method that raises alarms when system components deviate from the known norm. The ReCoMe engine is invoked whenever such alarms are received. Four analysis modules are applied to realize the automatic selection of operations in order to resolve the anomaly. Thereby, a root cause analysis should be called on-demand if alarms from multiple components are reported simultaneously. Following the principles of cohesion and decoupling, selecting a remediation operation is separated from its execution. An external operation executer is expected to provide SuO-specific operation implementations and manage their execution. In the following, we start with an explanation of the closed-loop system and its components. After that, we specifically describe the ReCoMe

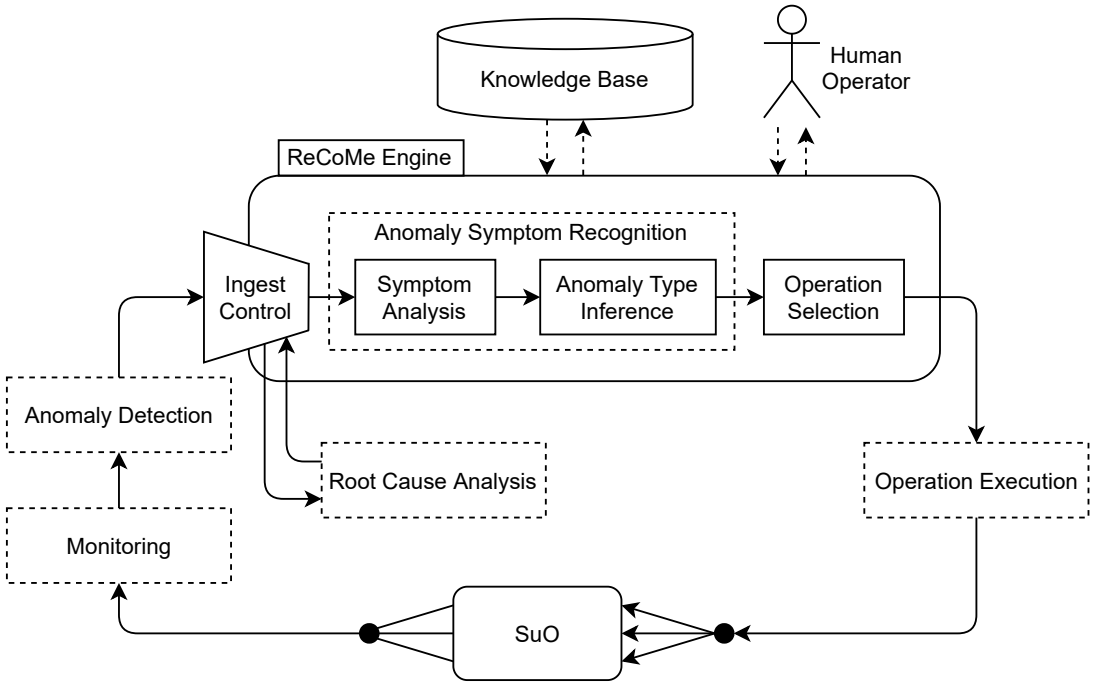


Figure 4.1: RoCoMe Engine for automated SuO remediation embedded in a closed-loop control.

engine and its four modules in detail.

4.2.1 Closed-Loop Control, Sensors and Effectors

Most AIOps systems are designed as a closed-loop control. It consists of two parts: The AIOps system and the system that is being operated (SuO). The AIOps system should ensure the correct and efficient execution of the SuO. Therefore, the current state of SuO components is sensed, compared to normal and desired behavior, and corrected if anomalous deviations are detected. Sensing and correction entail the requirement of two interfaces - sensor and effector - between the two systems.

The state of SuO components is not directly observable. Monitoring data from relevant SuO components are collected and analyzed to infer their state. Therefore it is essential to acquire such data from the system. The acquisition can be done *passively* by sample measurements of metrics like memory allocation, CPU utilization, or disk and network I/O statistics or by collecting system log data [146]. Alternatively, system information can be acquired *actively* via probing. Two examples for this are traces that track requests through multiple components or system layers [217] and profiles, which provide runtime information of predefined workloads executed on certain components [101]. The majority of systems allow a passive observation of metric data, which is the primary data source throughout this work.

Monitoring systems are measuring the metrics of components at regular time intervals. Such sequences of measurements allow not only to assess the component's state based on the recent observation but provide the opportunity to analyze the progress of metric data over time. Such temporally progressing data are referred to as time series [112]. We define the monitored metric data as a multivariate time series $X = (X_t \in \mathbb{R}^d : t = 1, 2, \ldots)$, where d is the number of univariate time series, i.e., the number of metrics that are monitored. Each element X_t is an observation

of all metrics at a certain point in time t and thus is a vector with d elements. A single metric, i.e., a univariate time series, without a specific point in time is written as X^i. To reference a specific metric i at time t, X_t^i is written. A time series interval between time a and b, with $a < b$ is written as $X_{a:b} = \{X_t : a \leq t \leq b\}$. Combining both, it is possible to reference a single metric i within an time interval a and b as $X_{a:b}^i$.

Metric data are a latent representation of the SuO component's state. The analysis methods of an AIOps system continuously analyze these data to infer the states of components. The AIOps system's overall task is to continuously decide whether operations need to be executed or not based on the results of the analysis methods. This process can be expressed as a function $\mathscr{F}$

$$\mathscr{F} : \mathbb{R}^d \mapsto \mathbb{O} \tag{4.1}$$

that maps from the metric data space $\mathbb{R}^d$ to the operation space $\mathbb{O}$. The mapping function in Equation 4.1 can be applied on each time series sample or an aggregation of multiple samples. The objective of the function $\mathscr{F}$ is to decide on an operation. It can be either an operation to restore an expected, i.e., normal, SuO state, or the *noop* operation representing to do nothing if all components are running as expected.

4.2.2 Anomaly Detection

Anomaly detection is usually the first analysis method in an AIOps system. It is considered a prerequisite that raises alarms whenever anomalies are detected. These alarms are indicating that the execution of certain operations is required to restore a normal operational state. Within the AIOps system, it triggers the selection of operations to resolve the anomaly. Generally, anomaly detection is an umbrella term for a group of methods that aim at the identification of observations that are deviating from the known norm [103, 116]. Therefore, monitoring data analysis methods are employed to distinguish normal from anomalous data samples, usually via a binary output [171, 211, 222]. A component is indicated to be either normal or in an anomalous state.

In general, anomaly detection methods can be grouped based on the availability of labeled data. Supervised methods assume labels for both anomalies and normal data and are explicitly trained to distinguish between those. Semi-supervised methods assume labels for part of the available data while the remaining samples are unlabeled. Models can be trained either only on normal samples, i.e., anomalies are excluded from the training process, or available anomaly samples can be used to enhance the ability of the model to distinguish normal samples and anomalies. Unsupervised methods do not assume any labels. Thereby, assumptions are utilized during model training. A common assumption is that the available data mainly contains normal samples, i.e., anomalies are rare [108, 211]. Anomaly detection models are training to represent the normal state of components. Everything that deviates from the known norm is assumed to be anomalous.

Since IBM presented the concept of autonomic computing [119] anomaly detection was considered the inevitable first analysis step. However, the basic definition of anomalies as deviations from the known norm does not contain further implications other than the fact that the current monitoring data is not normal. This observation does not enable full anomaly recovery automation since it leaves the task of further symptom analysis, operation selection, and execution to human experts. A subsequent analysis of symptoms is required to enable the automatic operation selection. Thereby, the aspect of propagating anomalies in distributed systems

needs to be considered. It entails the emergence of multiple alarms in short time frames, which requires the management of these alarms.

4.2.3 Operation Execution

The implementation of operations to resolve anomalies is component- and tool-specific. Operation execution refers to the realization, i.e., implementation and execution, of operations. It depends on the SuO component effector interface and might even change over time, e.g., when software updates occur. Furthermore, it depends on the tools that are utilized to interact with the effector interfaces. This aspect should be demonstrated using an example. Considering a simple restart operation, the command implementation to achieve a restart of a component depends on its effector interface. Virtual machines can be restarted by accessing it via, e.g., ssh or through the cloud computing management services API. Containers are usually restarted via the API of the container orchestrator. Due to vendor-specifics, even more diversities exist across different physical hardware devices such as network nodes or compute nodes. Furthermore, an API can change between software versions. The API call to restarting a container can differ after updating the container orchestrator software to a newer version.

Therefore, we apply the principles of cohesion and decoupling to separate the selection of an operation from its implementation and execution. The ReCoMe engine enables the automatic operation selection via anomaly symptom recognition. After that, the task of executing implemented operations is delegated to the operation execution module. It performs the actual execution, observes the execution progress and outcome, and can report whether the operation execution was successful or failed.

4.3 ReCoMe Engine

Applying anomaly detection on metric data should raise alarms whenever deviations from a known norm occur. Traditionally, this requires human operators to investigate these alarms, analyze symptoms, select or compile operations, and execute them. Executing these steps is a challenging, time-consuming, and often repetitive task. We propose the remediation and recommendation (ReCoMe) engine that supports system operators by automatically selecting operations to resolve anomalies and transfer SuO components to a normal state.

An overview of the ReCoMe engine is depicted in Figure 4.2. It provides an interface to the metric data of SuO components and the respective alarms of the anomaly detection. For each SuO component, the alarm channel receives alarms raised when the state of the component changes. Therefore, two alarm message types are defined:

- ◇ From normal to anomaly: $N \mapsto A$, and
- ◇ from anomaly to normal: $A \mapsto N$.

After an $N \mapsto A$ alarm is received, recent metric data from the anomalous SuO component are requested via the metric data channel. Until an $A \mapsto N$ alarm is received, the ReCoMe engine continuously receives recent metric samples whenever available. Note that the metric channel is only utilized when $N \mapsto A$ alarms are raised and is disabled shortly after a $A \mapsto N$ alarm is received. Normal SuO components generally do not transmit metric data to the ReCoMe engine

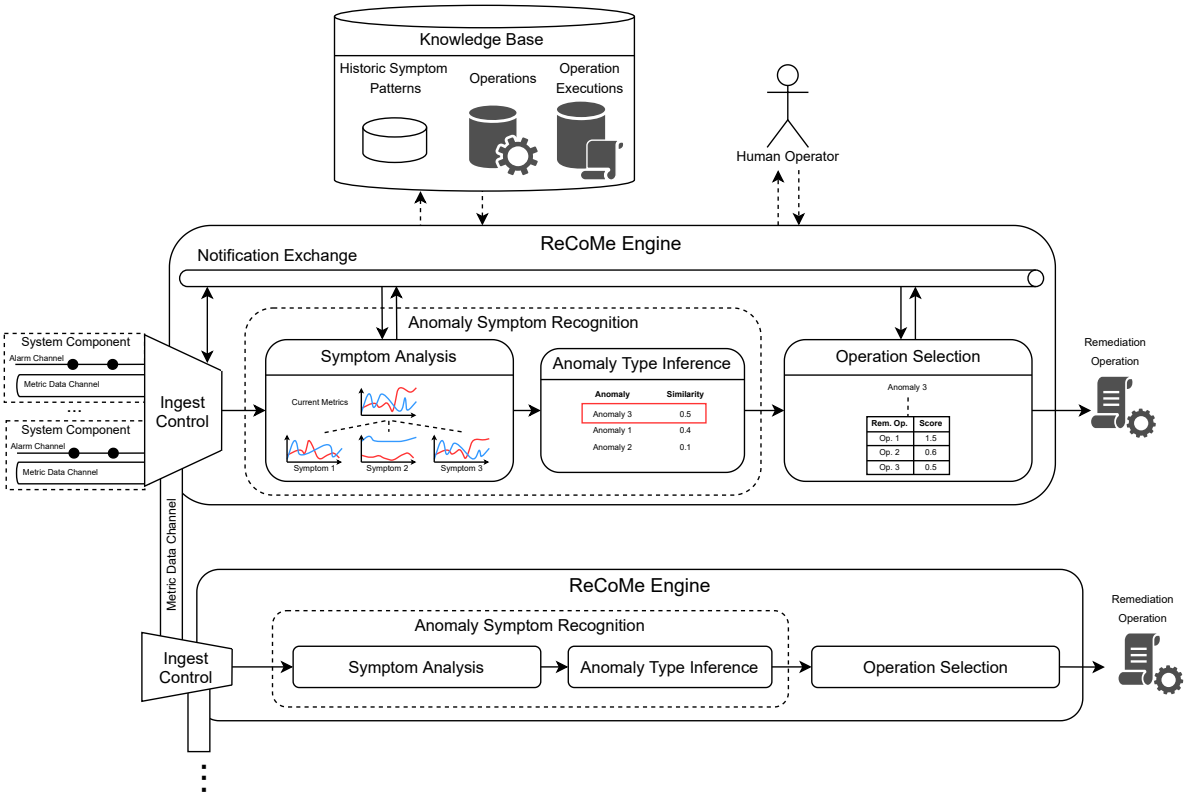


Figure 4.2: Overview of all RoCoMe Engine components, the main input and output interfaces as well as interfaces to the human expert and the knowledge base.

since normal states do not entail automatic operation selection. Thus, the anomaly detection serves as a filter that prevents the handling of irrelevant (i.e., normal) SuO component states and activates the ReCoMe engine only when it becomes relevant (i.e., to handle anomalous SuO component states).

The core functionality of the ReCoMe engine consists of three modules. Thereby, the anomaly symptom recognition is split into two parts: Symptom analysis and anomaly type inference.

1. The ingest control module provides interfaces to the metric data of SuO components and the respective anomaly detection. It receives alarm messages via the alarm channel and metric samples via the metric channel. The SuO is expected to be a distributed system with various interconnected and interdependent components. This entails that several anomalies might coincide due to propagation. Whenever the ingest control module receives alarms from several components simultaneously, it can request an external RCA module to identify the root cause. Furthermore, it manages the order of component remediation and recovery. First, the root cause anomaly is resolved. After that, it checks all components for the receipt of an $A \mapsto N$ alarm. If such an alarm is missing, it again initializes the automatic operation selection. It further enables hierarchical chaining of ReCoMe engines. Whenever a ReCoMe engine cannot resolve the anomaly, the metric data and meta-information about the anomalous component can be transferred to another ReCoMe engine.

2. The anomaly symptom recognition module consists of two parts. First, recent metric data of the anomalous SuO component are received. These data are analyzed by symptom analysis. It tries to recognize symptom patterns of known anomaly types. Until an $A \mapsto N$ alarm is observed, it updates the matching results with each newly received metric data sample. The result is an anomaly type list, sorted by a score. The score represents the similarity of the currently detected anomaly's symptoms to all known anomaly type symptoms. Second, the anomaly type inference module analyzes the scores of the anomaly type list. Based on the symptom analysis result, it infers the anomaly type. This includes the consideration that the results reflect a yet unknown anomaly.

3. The operation selection maps the inferred anomaly type to a list of available operations. It is assumed that a specific anomaly type can be resolved by more than one operation, leading to the availability of several operation candidates. The operation selection should enable the selection to rank the operation based on a preference score calculation. In the case of an unknown anomaly type, its handling is relayed to a human expert or a hierarchically next ReCoMe engine.

The final output of the ReCoMe engine is an operation that should be executed against the effector interface of the anomalous SuO component to restore its normal state. Following the principles of cohesion and decoupling from specific SuOs, the actual operation implementation and execution task is not realized by the ReCoMe engine. This task is delegated via requests to an external module that provides implemented operations, ensures the correct operation execution, and reports the result.

The ReCoMe engine applies two general concepts when trying to remediate anomalous SuO components. First, the identified anomaly type is known, and a set of operations exist to resolve it. In this case, the automatic selection of operations can be made. Second, when operations cannot be automatically selected, the ReCoMe engine delegates the task to a human operator who should resolve the system anomaly or a hierarchically next ReCoMe engine. Thereby, it is assumed that eventually, an expert provides operations for the yet unknown anomaly type, enabling the automatic operation selection in the future. These operations are stored persistently in a knowledge base and used to automatically improve the ReCoMe engine's ability to select operations. In the following, the knowledge base structure and all modules are explained in detail.

4.3.1 Knowledge Base

Various information is required to realize the anomaly symptom recognition and automatic selection of operations. This information is stored in the knowledge base, which acts as the ReCoMe engine experience. Generally, the required information can be categorized into three parts which are relational tabular structures. Each table contains a column with a unique identifier for each entry. This *ID* property will not be explicitly mentioned when the respective table structures are explained. In the following, the structure of each part is described.

The first part represents the storage of historical anomaly symptoms. It enables the ReCoMe engine to group anomalies based on common symptoms and match occurring anomalies with these known types. The anomaly symptom recognition is essential for mapping anomaly types to operations, enabling automatic operation selection. For each anomaly that occurred within the SuO, three properties are stored.

- ◇ **Time stamp**: Timestamp at which the anomaly occurred. It allows to analyze the MTBF, identify seasonality patterns and other aspects related to time.
- ◇ **Symptom representation**: Metric data are considered to be a latent representation of a SuO component state. They contain anomaly type-specific symptoms. A representation of these symptoms is stored to enable grouping and type recognition. A representation can be either the metric data series or a transformed representation (e.g., compressed, filtered).
- ◇ **Anomaly type**: The anomaly type is a label that groups anomalies. When applying anomaly symptom recognition, this label assigns a type to the detected anomaly by comparing it to symptom representations of known types. Note that the knowledge base is assumed to contain several entries of every existent anomaly type over time.

The second part represents the storage of available operations. It is required to enable the mapping of anomaly types to a set of available operations done by the operation selection module. Each table entry consists of two properties.

- ◇ **Component selector**: This selector is a set of SuO component properties that is used to select the components against which this operation can be executed.
- ◇ **Anomaly types**: A list of anomaly types can be resolved by executing this operation.

The third part represents the storage of meta-information about operation executions. If multiple operations are available to resolve an anomaly, these metrics are used to calculate preference scores to rank them. Whenever an operation is executed, five properties are stored.

- ◇ **Operation ID**: Reference to an operation from the operation table.
- ◇ **Anomaly type**: Anomaly type that was tried to be resolved. This type matches entries of the historic anomaly symptom table. It is also an item within the anomaly types property of the operation referenced by the operation ID.
- ◇ **Successfully executed**: This property does not imply that the anomaly was successfully resolved but instead states if the operation was executed without raising errors (e.g., checking the return value). It is received from the operation execution module. The successful execution of operations can depend on the current state of the SuO (e.g., a migration of a virtual machine can fail if there are no compute nodes that can host it). Since the state of SuO's is usually changing over time, operations can fail. This property is a binary flag. By analyzing several executions of the same operations, the distribution of this flag can be interpreted as the robustness of the operation to different SuO states.
- ◇ **Successfully remediated anomaly**: This property is a binary flag that indicates whether the anomaly was successfully resolved after the execution of this operation. For this to be true, the operation needs to be successfully executed, and an $A \mapsto N$ alarm must be received.
- ◇ **Execution duration**: The time that it took to execute the operation. This property can be used to prefer fast operations over such that require more time.

These four parts of the knowledge base are the basis for the ReCoMe modules.

4.3.2 Ingest Control

The SuO is generally assumed to be a distributed system. The ingest control has four tasks to meet the requirements of such systems. First, the distributed systems entail the management of anomaly alarms and the request of metric data from different components. Both requirements are realized via respective alarm and metric data channels for each SuO component. Second, metrics of components have different ranges and are respectively influenced by noise. The Ingest control applies a pre-processing of the metric data before redirecting them to the anomaly symptom recognition. Third, we consider SuOs such as fog computing environments, where components are geographically distributed. Thereby locally deployed AIOps systems can operate autonomously without having to send data to central instances. However, anomalies cannot always be handled locally. The ingest control allows to hierarchically couple ReCoMe engines. If an anomaly cannot be resolved, it can redirect the metric data to a hierarchically next ReCoMe engine and thereby delegate the task of resolving the anomaly. Fourth, propagating anomalies result in the simultaneous occurrence of $N \mapsto A$ alarms from multiple components. The ingest control module selects the components that require the execution of operations to resolve the anomaly and initiates the automatic selection of operations. These reported anomalies can be categorized into three types.

1. A component can be the source of the propagating anomaly. It is the anomalous component from which the anomaly propagated to neighboring components and possibly beyond.

2. Components that became anomalous due to propagation require active remediation, i.e., fixing the anomaly source will not resolve the anomaly on such components.

3. Components that became anomalous due to propagation do not require active remediation, i.e., fixing the source anomaly will resolve the anomaly on such components.

To determine type 1, the ingest control can access an external root cause analysis module to determine the root cause of a group of anomalous components. Every anomalous component of type 1 or 3 leads to invoking the subsequent ReCoMe engine modules to select remediation operations. The distinction between the type 2 and 3 components can be made by resolving the root cause component first and observing whether an $A \mapsto N$ message is received for the other anomalous components.

4.3.3 Anomaly Symptom Recognition

The detection of anomaly type-specific symptom patterns should allow the inference of anomaly types at runtime. We refer to this as anomaly symptom recognition, whereby metric data from anomalous system components are received and analyzed. First, distinctive patterns in the metric data should be identified and compared to available patterns of historical anomaly types. This step is referred to as symptom analysis. It expects metric data samples as input and calculates a score representing the similarity of the currently observed patterns to the symptoms of known anomaly types. The result is a key-value structure with anomaly types as keys and the respective score as the value. Second, an anomaly type should be inferred based on the scores. We refer to this step as anomaly type inference. It analyzes the scores and selects the highest scored anomaly type or decides that the scores indicate that the currently observed symptoms

represent a yet unknown anomaly type. The anomaly type inference output is an anomaly type (either one of the historical types or "unknown") and a confidence score reflecting the model's confidence in the inferred anomaly type.

Anomaly Symptom Analysis

There are two phases for the symptom analysis. First, a batch of recent metrics is requested, and symptom analysis is performed. This is done initially when $N \mapsto A$ alarms are received. Second, the anomaly symptom recognition module continuously receives metric data samples. Thereby, the symptom analysis results are constantly updated until an $A \mapsto N$ alarm is received. After the initial symptom analysis and after each update, the results are sent to the anomaly type inference.

The analysis of anomaly type symptoms based on metric data is challenging. Anomalies of the same type appear on components constantly influenced by various factors such as changing system load, varying parametrization, parallelism, different co-locations, CPU temperature, and many more. These dynamics are reflected in a high intra-type variation which means that symptoms can appear differently for anomalies of the same type. The infrequent occurrence of anomalies aggravates this problem. It translates to the availability of few examples for each anomaly type, eventually with a high intra-type variation. The availability of few examples combined with high intra-type variation is considered a significant problem in the area of pattern analysis [28].

These difficulties entail the possibility of imprecise recognition results. The assignment of an anomaly to a wrong type results in the selection of operations that probably will not resolve it. Propose wrong operations to human experts will increase the time they need to bring the system back into a normal state. The potential automatic execution of incorrect operations might not have any effect in the best case or result in an aggravation of the anomaly state. Human experts need to understand the effects of incorrect operation executions, execute operations to revert these effects, and fix the anomaly. Therefore, the symptom analysis method should be conservative in its predictions. Admitting that an anomaly type is unknown and delegating its handling to a human expert is considered less severe than the wrong anomaly type assignment leading to an automatic selection of incorrect operations.

The ReCoMe engine is designed to update its knowledge base constantly. Therefore, after anomalies are matched, the representation of their type-specific symptoms is added to the list of historical anomalies. To be utilized in future calculations, it needs to be assigned to an anomaly type group. However, the symptom analysis module cannot verify the correctness of its results. Thus, it relies on the feedback that contains information about the automatic operation selection process from the operation selection module. The assignment to an anomaly type group is done automatically if the group can be unambiguously inferred from the feedback message. Otherwise, a list of potential anomaly types is presented to a human operator, who has to select the correct type.

Anomaly Type Inference

Symptom patterns of a detected anomaly can be unambiguously matched to a known anomaly type in an optimal case. Such precise recognition means that the symptom analysis yielded a high similarity to a specific type and low similarities to all other types. However, the intra-type

variation of symptoms can lead to ambiguity. Also, the symptom analysis results are updated with each recent metric data observation. The selection of operations and the eventual communication of the analysis results with human experts require a quantitative expression of confidence in the anomaly symptom recognition. This expression has practical importance. High confidence that is assigned to an anomaly type can result in a precise selection of an operation. For such cases, an automated execution initiation can be considered. Lower confidence value results would require human expert confirmation before they are executed.

First, the ReCoMe engine must recognize if an anomaly type is unknown, i.e., examples of that type were not observed yet. Computer systems are constantly evolving, and new system anomaly sources are introduced over time [65, 111, 219]. Examples of such system evolutions are hardware modernizations, software updates, and network reconfigurations. Therefore, anomaly types that were not observed yet must be expected to occur over time. Such anomaly types are represented by dissimilarity to all known anomaly type symptoms. The anomaly type inference analyzes the symptom analysis results and calculates their ambiguity. It decides whether the observed anomaly's type is yet unknown or if the anomaly type with the highest symptom pattern similarity score can be selected. A high ambiguity represents an unknown anomaly type, while a low ambiguity allows the inference of a known anomaly type.

Second, the symptom analysis initially calculates the similarity after receiving an $N \mapsto A$ alarm based on a batch of recent monitoring data. Subsequently, it updates the results after every new metric sample until the anomaly is resolved. Therefore, the anomaly type inference module must decide a point where it infers the anomaly type and transmits the result to the operation selection module. A frequently changing inference of different anomaly types entails that the operation selection selects different operations. A changing operation selection results in potential roll-backs of previously executed operations or human experts are confronted with varying recommendations. It implies insecurity and reduces the trust of human experts in the system. Therefore, the anomaly type inference module must consider a timely decision on the one hand but wait until the results are stable to prevent frequent anomaly type inference changes.

Third, the subsequent operation selection requires the confidence of the anomaly type recognition towards the predicted anomaly type. Therefore, the anomaly inference calculates a quantitative value based on the symptom analysis result reflecting its confidence towards the inferred anomaly type.

An anomaly type inference depends on the properties of the symptom analysis method. Although there is a logical separation of symptom analysis and anomaly type inference, an implementation of both modules has interdependencies with limits to a strong separation. Generally, the anomaly type inference module serves as a mediator between symptom analysis and operations selection. It can be understood as a set of post-processing steps to improve the applicability of the overall anomaly symptom recognition for automatic operation selection.

4.3.4 Operation Selection

The operation selection module should automatically select an operation to resolve the currently observed anomaly. It receives the anomaly symptom recognition result, which is considered as the basis for the operation selection. The selection of the operation is achieved in four steps.

First, the anomaly type received from the anomaly symptom recognition is used to query available operations from the operation knowledge base. Thereby, the received anomaly type is

compared to the anomaly type list of each operation entry in the knowledge base. If both have elements in common, the remediation operation is considered a feasible candidate.

Second, it checks whether each selected operation can be executed against the anomalous system component. Therefore, the component selector of the operation knowledge base entry is utilized. The operation selection queries meta-information of the anomalous component and compares them with the information stored in the component selector field. If they match, the operation remains a feasible candidate. Otherwise, it is excluded.

Third, an operation should be selected from the set of feasible candidates. Depending on the received anomaly type list and the number of defined operations to resolve each type, the feasible set of operations can contain multiple elements. The operation selection module implements the management of this feasible operations set. Thereby, a preference score is calculated for each element. These preference scores are calculated by loading all entries from the operation execution knowledge base for a specific operation. A score function takes these entries together with the confidence value for the inferred anomaly type as input and calculates the preference score. Applying this to each feasible operation results in a set of operations descendingly sorted by the respective preference score.

The operation selection module represents the interface between the ReCoMe engine and the human operator. Whenever the feasible set of operations is empty, a human expert must define an operation for the current anomaly. This allows to map the observed symptoms to the provided operation and eventually enables the automatic selection of operations for future occurrences of this anomaly type. Furthermore, the operation selection module can initiate the execution of an operation by transmitting a request to the operation execution module. It expects to receive the execution outcome, e.g., whether the operation execution was successful or not, or how long it took to execute it. However, the operation execution module cannot verify that a successful execution resolved the anomaly. This information is received from the anomaly detection module via a $A \mapsto N$ alarm. After receiving such an alarm, it provides feedback to the anomaly symptom analysis module.

4.4 Modeling Concepts for Anomaly Symptom Recognition

The anomaly symptom recognition is the central prerequisite to enable the automatic selection of operations. It analyses the metric data of system components to detect anomaly type-specific symptom patterns. Thereby the similarity of the observed symptoms to all known anomaly type-specific symptoms is calculated. After processing the result by an anomaly type inference, the inferred type is used to select feasible operations to resolve the anomaly. Considering the previously described challenges, anomaly symptom recognition is a challenging task. An essential focus of this thesis lies in developing methods to solve these problems. In this section, we define symptom analysis as a classification problem. After that, we analyze general classification modeling concepts and derive model requirements to realize the anomaly symptom recognition.

4.4.1 Classification of System Metric Data

We formulate the task of anomaly symptom recognition as a classification problem. Thereby, the goal is to assign a sequence of metric data samples X to one of K discrete classes. These

classes are associated with anomaly types. We write for the set of known classes $C_k = \{c_i : 1 \leq i \leq K_k\}$, where K_k is the number of known anomaly types.

Matching anomaly types of IT system components differ from the conventional classification problem. The set of known anomaly types is never complete. New anomalies can occur in a system with an association to a yet unknown anomaly type. Based on this, an additional unknown (c_u) class is added to the class set. We define our set of discrete classes as $C = C_k \cup \{c_u\}$ and $K = K_k + 1$. The following descriptions provide an intuitive understanding of both class types:

- C_k - All anomaly types in the set C_k were previously observed, i.e., known. It can be assumed that at least one operation exists that can be executed to resolve the respective anomaly class. If a detected anomaly is classified as an element of this set, the operation selection module can automatically select the operations.

- c_u - Computer systems change over time. Software components undergo regular version updates, and hardware is modernized over time. Therefore, the set of known anomaly types C_k is considered to be never complete. This incompleteness entails the necessity to be able to recognize unknown anomaly types. An anomaly symptom recognition model must admit that a detected anomaly that cannot be associated with any known type is indeed unknown. This unknown recognition is crucial since operations are assumed to be not available for unknown anomaly types.

The classification is realized via a function that maps metric data to an anomaly type, generally defined as $f_C : X \mapsto C$. Thereby, the input space is divided into decision regions whose boundaries are referred to as decision boundaries or decision surfaces [28]. They are positioned so that intra-class samples are located in the same region while inter-class samples are located in different regions. Each region is associated with a class (i.e., anomaly type in our case).

The challenges associated with distributed systems like heterogeneity or varying external factors lead to a complex f_C. Defining an analytical model for such functions implies an exhaustive knowledge of all possible anomalies and their manifestations. The analytical definition is considered infeasible for distributed systems such as cloud or fog computing systems [211]. Therefore, we infer f_C from available anomaly type examples. The placement of decision boundaries within the input space is subject to optimization based on a set of examples in combination with an objective [28]. Informally, the objective is to set decision boundaries such as anomalies of the same type are located within the same decision region while anomalies of different types are located in different regions. Inferring the function f_C is referred to as *training* or *fitting*. The result is a trained function f_C or *classification model*. In the following, we analyze different decision boundary examples and the properties of monitoring data to derive the anomaly symptom recognition requirements.

4.4.2 Classification Model Requirements

How decision boundaries are placed in the input space defines how classification model results are expressed. For more than two classes, a 1-of-K coding scheme is commonly used. Thereby, the classification result is represented as a vector $\mathbf{y}$ of length K, where each element encodes the association with the respective class:

$$\mathbf{y} = \{y_j : 1 \leq j \leq K\} \tag{4.2}$$

We analyze different decision boundary placement policies and derive the requirements for anomaly symptom recognition. Figure 4.3 shows six plots with classification model examples and respectively highlighted decision regions and boundaries. The red, blue, and green dots represent examples from class 1, class 2, and class 3. They are referred to as training samples that were used to optimize the decision boundary placement. After training, the black dots are subject to classification and, therefore, referred to as test samples. In this example, all data samples are located in a two-dimensional space. The data samples within the left column of plots are linearly separable, while the data samples in the right column of plots are non-linearly separable.

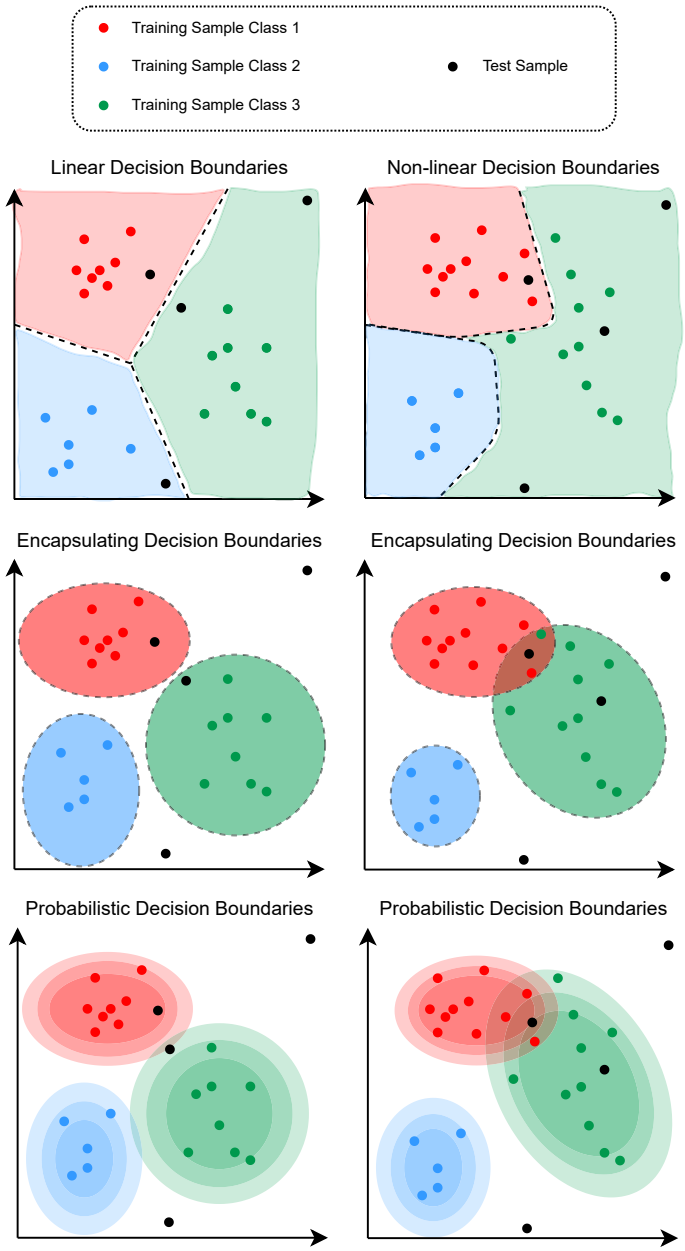


Figure 4.3: Visualization of classification decision regions and boundaries.

Decision regions within the input space can be either defined by hard (first two rows of

plots) or soft (last row of plots) boundaries. The separation in the first row of plots is done by linear (left) or non-linear (right) discriminant functions. Such functions are dividing the whole input space into respective decision regions. As a result, each test sample is assigned to exactly one class, and thus, exactly one vector element takes the value 1 while all others are 0. A sample that is located in the red, green or blue area results respectively in $\mathbf{y} = (1,0,0)^T$, $\mathbf{y} = (0,1,0)^T$, and $\mathbf{y} = (0,0,1)^T$

The above-described aspects also apply to the encapsulating decision boundaries (second row of plots). However, an encapsulating decision boundary encloses a sub-space of the original input space as a decision region. The two plots of the second row show examples of spherical regions [1]. This results in the existence of regions within the input space that are not encapsulated by a decision region. Samples located in these areas are represented by a result vector that contains only elements with value zero: $\mathbf{y} = (0,0,0)^T$.

For encapsulating decision boundaries, the regions can overlap (right plot of the second row of plots). Whether to allow or disallow overlapping depends on the use case. It might be required to decide on one or no class, or it can be feasible to provide a set of candidate classes. Overlapping entails the possibility of assigning a sample to more than one class. Therefore, more than one element of the vector $\mathbf{y}$ can take the value 1. In the left plot of the second row, a test sample is located in the overlapping red and green regions. This results in $\mathbf{y} = (1,1,0)^T$ Note that hard boundaries in general entail that elements of the result vector are binary, i.e. can take either the value 1 or 0.

$$\forall y \in \mathbf{y} : y \in \{0,1\}. \tag{4.3}$$

Whether exactly one $(\mathbf{1})$, at most one $(\mathbf{2})$ element or an arbitrary number of elements $(\mathbf{3})$ is allowed to take the value 1 can be expressed by summing the elements $s = \sum_{\forall y \in \mathbf{y}} y$ and define the allowed range of the result:

$$s \in \mathbb{N},\ (\mathbf{1})\ s = 1,\ (\mathbf{2})\ 0 \leq s \leq 1,\ (\mathbf{3})\ 0 \leq s \leq K. \tag{4.4}$$

The probabilistic decision boundaries are shown in the last row of plots. In the example plots, one single-modal Gaussian distribution[2] is assumed for each class. They map a sample to probability values that represents the association with the respective class. Note that the contour plots in both plots are showing the area of high probability for each class. There is a very low non-zero association probability with each of the classes for all the white regions. Samples that are located between two probabilistic decision regions can have similar membership probabilities to both classes. Therefore, the input space is covered by a superposition of all class distributions. Each element of the result vector represents the probability that a sample is associated with each class. It is a real value between 0 and 1:

$$\forall y \in \mathbf{y} : y \in (0,1). \tag{4.5}$$

This output allows expressing uncertainty towards a classification decision. In the left plot of the last row of plots, the result for the test sample located in the red area is $\mathbf{y} = (0.8,0.2,0.01)^T$. It can be seen that the model assigns a high association probability to class 1 and lower association probabilities to classes 2 and 3. Another example is a test sample located in the upper right white area. The result is $\mathbf{y} = (0.01,0.01,0.001)^T$. Low association probabilities for each

[1] Note that other shapes are possible as well.

[2] Note that other probability distributions can be used as well.

element express an uncertainty towards all classes. In the right plot of the last row of plots, a test sample is located in an area where classes 1 and 2 have an overlapping of high probabilities. This test sample results in a vector $\mathbf{y} = (0.8, 0.8, 0.01)^T$, expressing that the model cannot distinguish the sample's association between classes 1 and 2 but shows a low association probability to class 3. A normalization of $\mathbf{y}$ can be applied, after which the sum of all $\mathbf{y}$ elements is 1 [28]. However, it becomes difficult to interpret the uncertainty in the results. Depending on the use case, an additional post-processing procedure might be required to assign classes based on the membership probabilities of samples. By defining thresholds for each class probability distribution, the soft probabilistic decision boundaries can be transformed into hard encapsulating decision boundaries.

The necessity of detecting previously unknown anomaly types (c_u) is required. A model must express that a sample is not matching any of so far known classes. Furthermore, similarity to classes and an expression of confidence in the classification result are needed. This can be realized with encapsulating and probabilistic decision boundaries. Therefore, we focus on these decision boundary types when implementing an anomaly symptom recognition model.

Chapter 5

Density Grid Pattern Model

This chapter describes a method for anomaly symptom recognition, which was published in [3]. It is utilized on system components with a regular and homogeneous workload, such as virtual machines or containers that contain a particular service. The general workflow of the method is depicted in Figure 5.1. The ingest control does the preprocessing of the metric series. After that, the symptom analysis based on grid pattern comparison is applied to the data. Thereby, the metric data that contain anomaly type-specific symptom patterns are transformed into a grid representation, i.e., a grid pattern. Patterns for which the anomaly type is known are labeled and stored in the knowledge base. We define a function to calculate the similarity between grid patterns. The symptom analysis is realized by comparing grid representations of anomaly type symptoms with existing labeled grid pattern examples and calculating respective similarities. A list of anomaly types together with respectively computed similarity scores is forwarded to the anomaly type inference. It decides on a class that is either a known anomaly or the "unknown" class. Latter indicates that the observed symptom patterns are dissimilar to all known anomaly type symptoms and potentially represent an anomaly type that is yet unknown. The anomaly type inference requires parameters to make this decision, which is set based on the so-far available grid patterns in the knowledge base.

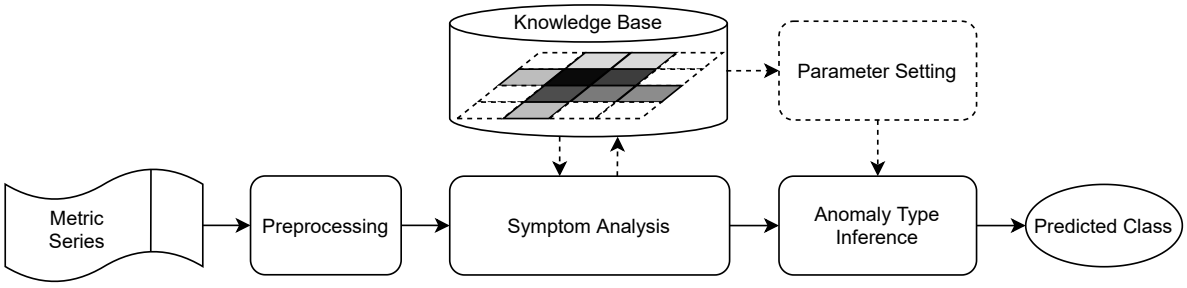


Figure 5.1: General workflow of the grid pattern comparison method.

5.1 Anomaly Symptom Analysis

The symptom analysis is realized via two steps. First, the received metric series is transformed into a density grid representation. The result is a grid pattern that represents the anomaly type-specific symptoms. We emphasize the low computational overhead of the transformation, which discretizes the input space and aggregates the temporal dimension while preserving general temporal information. Second, a similarity measure for density grids is defined to compare them. Having a set of grid patterns labeled by the respective anomaly type allows inferring the anomaly type of an unlabeled pattern via a comparison of the observed pattern with the available examples.

5.1.1 Preprocessing

Before the anomaly symptom recognition is executed, a range normalization by rescaling each element of X_t to the range $(0,1)$ is applied

$$\hat{X}_t^i = \frac{X_t^i - X_{min}^i}{X_{max}^i - X_{min}^i}. \tag{5.1}$$

For resource metric data of IT system components, the limits of most resource types are well-known (e.g., number of CPU cores and their frequency or maximum available memory). This is the case for the considered metrics. For some data types like latencies between network endpoints, context switches, or packet errors, it is challenging to decide on upper or lower boundaries. Those can be determined based on available metric series data.

5.1.2 Density Grid Transformation

First, the density grid transformation of the received metric data is executed to achieve a compact grid representation. The process is depicted in Figure 5.2.

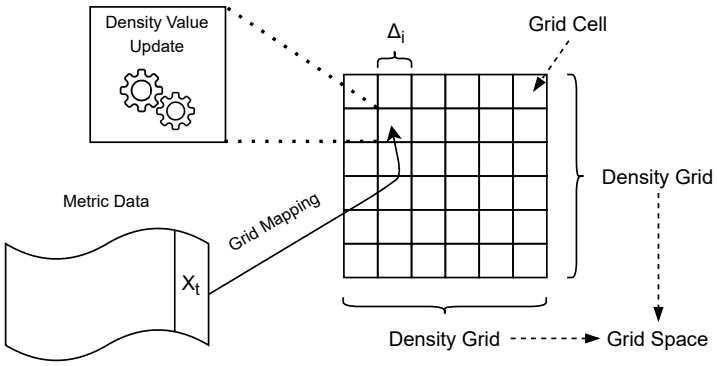


Figure 5.2: Density grid transformation.

We describe the grid transformation as temporal aggregation and spatial discretization. Each received and normalized metric data sample $X_t \in \mathbb{R}^d$ is defined within a real space $\mathbb{R}^d = \mathbb{R}_1 \times \mathbb{R}_2 \times \cdots \times \mathbb{R}_d$. The spatial discretization is achieved by splitting the value range of each dimension into a defined number h of intervals. Thereby, a known lower and upper

bound is assumed for each element of X_t, which is given since a rescaling is applied during preprocessing. Based on this, it is possible to calculated the interval Δ lengths of each dimension i via

$$\Delta_i = \frac{X^i_{max} - X^i_{min}}{h}, \text{ where } 1 \leq i \leq d. \tag{5.2}$$

This is applied on all d dimensions resulting in respective density grids G_i. A suchlike discretized bounded real space is a set of d density grids and referred to as grid space $\mathbb{G}$:

$$\mathbb{G}^d = G_1 \times G_2 \times \cdots \times G_d. \tag{5.3}$$

An element of a grid space is a grid cell g. It is a key-value structure, where the key is a tuple of integer values $\mathbf{q}$ that defines the location of the cell in the grid space and the value is an aggregated representation of all X_t that are mapped to it:

$$g = <\mathbf{q} : \delta>, \text{ where } \mathbf{q} \in \mathbb{N}^d \text{ and } \delta \in \mathbb{R}. \tag{5.4}$$

We refer to the δ as density value. The cardinality of the grid space can be calculates via $|\mathbb{G}| = \prod_{1 \ldots d} h$. Such relationship with the dimensionality d of the metric data X_t is often referred to as the curse of dimensionality [28]. It states that the volume of space increases exponentially with the dimensionality of the input data, leading to increasing sparsity. Therefore, it is important to limit the input space to prevent an unmanageably high grid space volume.

As shown in Figure 5.2, the density grid transformation consists of two steps: *Grid mapping* and *density value update*. The mapping assigns a metric data sample X_t to a grid cell g. Thereby, the location of the target grid cell is calculated based on the values of each element of X_t. Since the density space $\mathbb{G}$ is the result of dividing each dimension of the original space into h intervals, the calculation is defined as:

$$q_i = \left\lfloor \frac{X^i_t}{\Delta_i} \right\rfloor. \tag{5.5}$$

The calculation and update of the density value are explained based on a minimal example depicted in Figure 5.3. It shows two time series (A and B) consisting of 50 data points raging between 0 and 1. In each time series, 25 data points have a value of 0, and the other 25 data points have a value of 0.99. However, their temporal pattern differs. Note that the anomaly symptom recognition should distinguish anomaly types based on patterns observed in the metric data. In this example, the task would be to calculate grid cell values that allow the distinction of both time series. First, both h are set to 2, which results in a boundary at 0.5 (dotted lines) that separated the value range into two intervals (spatial discretization). Next, the temporal aggregation combines the points that are located in each interval to one representative value. Generally, this concept is known from histograms. However, histograms analyze the frequency of data points in each interval while the temporal information is lost. The relative frequency of points within each interval is calculated in the depicted example, resulting in identical histograms for both time series. We provide a temporal aggregation that encodes temporal information when aggregating the values of each interval. The bar chart visualization of the density grids shows that it is possible to distinguish both grids after aggregation.

To encode temporal information, we apply the concept of temporal decay. Thereby, a score is calculated for data samples that are mapped to a grid cell. Over time, these scores constantly decrease and asymptotically approach 0. The score is referred to as density, and the constant

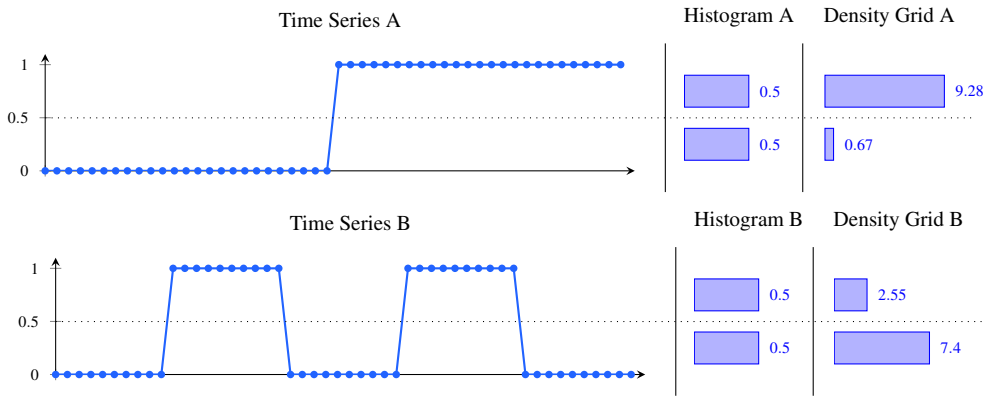


Figure 5.3: Exemplary spatial discretization of two time series together with two temporal aggregation methods. Note that the scale of the bar charts is the same within each column but differs between columns.

decrease is density decay. Based on related literature [53, 88, 134], an exponential decay is used:

$$D(t_m, t) = \lambda^{t-t_m}, \text{ with } t \geq t_m \text{ and } 0 < \lambda < 1. \tag{5.6}$$

The time t_m is the time at which a metric data sample was mapped to the grid cell. Thus, function D calculates the density value for a single metric data sample, which decreases when time t is progressing, i.e., $t_m \leq t$. Measuring the density in the moment at which the mapping is done (i.e. $t_m = t$) results in a value of 1. The rate at which the density value decrease is referred to as decay and can be controlled via the parameter λ. Examples of the decay function for different λ values is shown in Figure 5.4. It shows the exponential decrease of the density value over time. Thereby, values for λ that are closer to zero lead to faster decay while a lambda closer to 1 cause a slow decay.

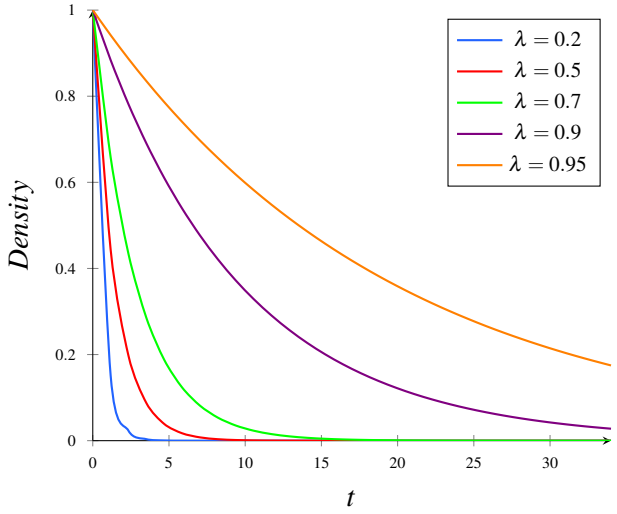


Figure 5.4: Visualization of density decay over time under varying λ.

Equation 5.6 defines the decay for one metric data sample. The overall density value δ of a grid cell at time t is the sum of all density values that were mapped to that cell so far:

$$\delta_t = \sum_{t_m \in T_m} D(t_m, t). \tag{5.7}$$

T_m represents a set of all times at which a metric data sample was mapped to this grid cell. Storing the time of every metric sample contradict the objective of temporal aggregation to achieve a compact representation. Based on Equation 5.6 and Equation 5.7, we show that the calculation of δ can be done incrementally, and thus, only the time of the last density value update is required.

Consider the current time t, the time of the previously received metric data sample t_m, and an arbitrary time between t_n. By applying the exponent product rule on Equation 5.6, it can be written as:

$$D(t_m,t) = \lambda^{t-t_m} = \lambda^{t-t_n} \cdot \lambda^{t_n-t_m} = \lambda^{t-t_n} \cdot D(t_m,t_n). \tag{5.8}$$

It means that the current density value for a single data sample can be calculated by scaling a past density value at time t_n by the factor λ^{t-t_n}. On that basis, the overall density value of a grid cell can be calculated by

$$\delta_t = \sum_{t_m \in T_m} D(t_m,t) = \sum_{t_m \in T_m} \lambda^{t-t_n} \cdot D(t_m,t_n) = \lambda^{t-t_n} \sum_{t_m \in T_m} D(t_m,t_n) = \lambda^{t-t_n} \cdot \delta_{t_n} \tag{5.9}$$

It can be seen that the density at a time t can be incrementally calculated by scaling the previously calculated density at time t_n by the factor λ^{t-t_n}. It is not required to store T_m but only the previous density value update time t_n. This time is set accordingly whenever a density value update is calculated.

Equation 5.9 calculates the density value update if no metric sample is mapped to the grid cell at time t. Whenever the update is calculated due to a metric sample being mapped to a grid cell, the case of $t = t_m$ needs to be considered. As previously shown, the density value equals 1 in this case. Therefore, Equation 5.9 must be updated as

$$\delta_t = 1 + \lambda^{t-t_n} \cdot \delta_{t_n} \tag{5.10}$$

By considering both cases, i.e., metric sample was mapped or not, we define the density value update for a grid cell as a recursive function, where t_0 is the start time of the density grid mapping:

$$\delta_t = \begin{cases} \lambda^{t-t_n} \cdot \delta_{t_n} & \text{, if no mapping} \\ 1 + \lambda^{t-t_n} \cdot \delta_{t_n} & \text{, if mapping} \end{cases} \tag{5.11}$$

$$\text{with } t_n < t, \text{ and with the initial conditions } \delta_{t_0} = 0,\ t_0 < t.$$

Referring to the example density grids in Figure 5.3, the density values were calculated with $\lambda = 0.9$

The density grid transformation is applied to achieve a compact representation of metric data. One criterion to apply for anomaly symptom recognition is a low runtime complexity. During the mapping procedure, the target interval for each dimension must be calculated to acquire the location of the target grid cell. This requires Equation 5.5 to be applied d times. After that, the density value of the grid is updated via invoking Equation 5.11 once. Neither Equation 5.5 nor Equation 5.11 has an internal higher order complexity. The overall time complexity can be expressed by using the big O notation $\mathcal{O}(d)$.

For the objective of anomaly symptom recognition, the metric data received when system components are anomalous are mapped to the grid space. Therefore, every anomaly example is represented by a set of grid cells and associated with an anomaly type (i.e., class in terms of

classification). Out of the whole grid space, only grid cells with a non-zero density are stored. Such a set of grid cells is referred to as a grid pattern ϕ. Grid patterns are representations of anomaly symptoms and, therefore, examples of an anomaly type. When assuming an available knowledge base of such patterns, the anomaly symptom recognition compares the anomaly symptom pattern to the known type patterns in the knowledge base to infer the type affiliation.

5.1.3 Grid Pattern Similarity

A notion of similarity is required to compare grid patterns. Given two example grid patterns ϕ_i and ϕ_j, we define a function $S_\phi(\phi_i, \phi_j) \in [0,1]$ to calculate their similarity, where 1 represents perfect similarity and 0 perfect dissimilarity. The intuition behind the proposed similarity concept is depicted in Figure 5.5. It shows two comparisons of grid patterns that are generated based on metric data with $d = 2$. The grid space is generated with $h = 4$. All density values are color-encoded with the respective scale shown on the right side of the picture. The left part visualizes the comparison of grid pattern ϕ_i, and ϕ_j while on the right part, ϕ_i, and ϕ_k are compared. Due to the alignment of grid cells with similar density values, the comparison of ϕ_i and ϕ_k should intuitively yield a higher similarity than ϕ_i and ϕ_j.

Based on this, the overlap of two grid patterns put on top of each other is calculated to determine their similarity. It means that a pairwise grid cell comparison is applied. Two grid cells $g_k \in \phi_i$ and $g_\ell \in \phi_j$ must fulfill the following criteria to be considered a pair:

1. Given $\mathbf{q}_k \in g_k$ and $\mathbf{q}_\ell \in g_\ell$, $\mathbf{q}_k = \mathbf{q}_\ell$: Pairs are determined based on their location. Cells with the same location tuple values are considered a pair.

2. Given $\delta_k \in g_k$ and $\delta_\ell \in g_\ell$, $\delta_k > 0 \vee \delta_\ell > 0$: At least on of the density values must be greater than zero.

The result is a set ψ containing all grid cell pairs as couples that fulfill both criteria.

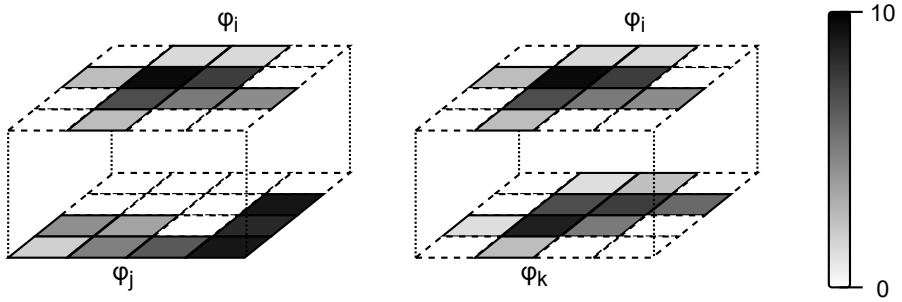


Figure 5.5: Grid pattern similarity.

The similarity of a grid cell pair is defined as the ratio of the lower density to the higher density:

$$S_g(g_k, g_\ell) = \frac{\min(\delta_k, \delta_\ell)}{\max(\delta_k, \delta_\ell)}, \text{ where } \delta_k \in g_k, \delta_\ell \in g_\ell \text{ and } (g_k, g_\ell) \in \psi. \tag{5.12}$$

Note that the denominator is never zero due to the second criterion of the grid cell pairs. The function S_g is therefore defined in the rang $[0,1]$. It is 1 (i.e. perfect similarity) when both values are equal and approaches 0 when the difference between both values increases.

Applying this on all grid cell pairs $(g_k, g_\ell) \in \psi$, the similarity function S_ϕ of two grid patterns is defined as the mean of all grid pair similarities

$$S_\phi = \frac{1}{|\psi|} \sum_{\forall (g_k, g_\ell) \in \psi} S_g(g_k, g_\ell). \tag{5.13}$$

There are two problems with a strict overlay of grid patterns. System metric data are generally noisy. Furthermore, the shape of the metric data series depends on different external and internal criteria. Considering a CPU-intensive task, the utilization of a component's CPU cores depends on the current load on the system (e.g., number of requests that need to be served in parallel), resulting in a higher or lower overall baseline. The effect of noise and a shifted baseline on the grid pattern calculation is depicted in Figure 5.6. Time series C jumps from value 0 to a baseline value of 0.59 while time series D jumps to a baseline value of 0.61. It can be seen that a slight variation of the baseline results in a mapping of data samples to another grid. Although both time series could be considered as similar, the pattern similarity defined in Equation 5.13 would yield a value of $0.\overline{3}$. The impact of noise can be observed in time series E, which jumps to a baseline of 0.60 and is affected by an additive Gaussian noise with $\mu = 0$ and $\sigma = 0.1$. This noise results in a mapping of data samples across all grid cells. The similarity between grid patterns C and E, and D and E is 0.34 and 0.25.

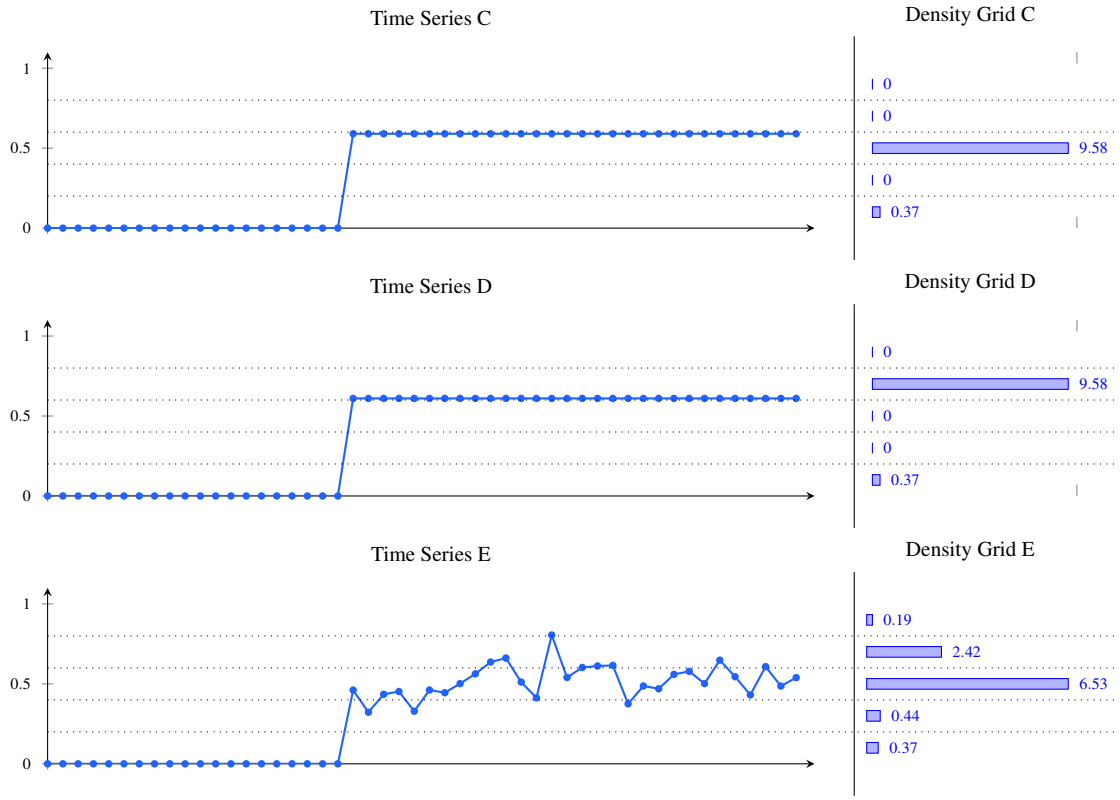


Figure 5.6: Examples of problematic cases when applying strict overlay of grid patterns.

Instead of calculating a strict overlay of two grid patterns, we relax the grid cell pair boundaries and consider a neighborhood around each cell pair. This neighborhood is considered symmetrically in both grid patterns. Thereby, the density value sum of the neighborhood cells

is used when calculating the ratio:

$$S_g(g_k, g_\ell) = \frac{\min(\sum_{\delta_{k_i} \in g_{k_i} \in \mathcal{N}(g_k)} g_{k_i}, \sum_{\delta_{\ell_i} \in g_{\ell_i} \in \mathcal{N}(g_\ell)} g_{\ell_i})}{\max(\sum_{\delta_{k_i} \in g_{k_i} \in \mathcal{N}(g_k)} g_{k_i}, \sum_{\delta_{\ell_i} \in g_{\ell_i} \in \mathcal{N}(g_\ell)} g_{\ell_i})}, \text{ where } (g_k, g_l) \in \psi. \tag{5.14}$$

The neighborhood of a grid cell $\mathcal{N}(g)$ is a set of cells that are located within a certain distance γ around g

$$\mathcal{N}(g) = \{g_i : L_1(\mathbf{q}_i, \mathbf{q}) \leq \gamma, \mathbf{q}_i \in g_i, \mathbf{q} \in g\}. \tag{5.15}$$

Due to the rectangular shape of cells and the resulting grid space structure, the L_1 norm is used as the distance metric. It can be non-formally described as the minimum amount of grid cell steps that lie between a start and destination cell. Cells that are direct neighbors have the distance of 1, i.e., one step needs to be done to get from one cell to the other. Formally, the L_1 norm is defined as

$$L_1(g_k, g_\ell) = ||\mathbf{q}_k - \mathbf{q}_\ell||_1 = \sum_{i=1}^{d} |q_{k_i} - q_{\ell_i}|, \text{ where } \mathbf{q}_k \in g_k \text{ and } \mathbf{q}_\ell \in g_\ell. \tag{5.16}$$

Since the elements of $\mathbf{q}$ are integers, the neighborhood threshold gamma is an integer as well $\gamma \in \mathbb{N}$.

The similarity calculation defined in Equation 5.14 mitigates the problems depicted in Figure 5.6. Setting the maximum distance for neighborhood selection to $\gamma = 1$, the similarity between density grid C and D becomes 1, which means that we achieved a controllable tolerance against baseline shifts. The similarity between grid patterns C and E, and D and E becomes 0.63 and 0.54, allowing configurable robustness against noise.

Two parameters are impacting the runtime complexity of the pattern similarity calculation. The highest impact is the number of Equation 5.14 invocations, which is equal to the cardinality of the cell pair set $|\psi|$. In the worst case, this cardinality is the total number of grid cells in the grid space $|\mathbb{G}^d| = h^d$. The neighborhood calculation introduces another parameter which is the sum calculation in Equation 5.14 that depends on the cardinality of the neighborhood set $|\mathcal{N}(g)|$. Both parameter are multiplicatively combined $|\mathcal{N}(g)| \cdot h^d$, resulting in a worst case big O notation of $\mathcal{O}(h^d)$.

5.2 Anomaly Type Inference

IT systems are expected to change over time. Also, certain anomaly types are rarely observable. The knowledge base of known anomaly types must be considered incomplete at all times. This assumption entails the necessity to detect unknown anomaly types. The symptom analysis is implemented as a density grid mapping method that transforms monitoring metrics of system components into grid patterns and calculates the similarity to a knowledge base of labeled patterns. A decision boundary definition is required to detect unknown anomaly types. Therefore, the anomaly type inference receives the similarity list and compares each similarity value to a boundary parameter β. If the similarity is below this parameter, the anomaly type is filtered from the list. Whenever the list happens to be empty, the anomaly is considered unknown.

The setting of the parameter β is done individually for each anomaly type based on the intra-type similarities of all available grid patterns. For each know anomaly type c, we set a

hard decision boundary by calculating the minimum intra-type pattern similarity:

$$\beta_c = \min(S_\phi(\phi_i^{(c)}, \phi_j^{(c)})), \text{ where } 1 \leq i, j \leq n_c, n_c \geq 2 \text{ and } i \neq j. \tag{5.17}$$

Thereby, n_c is the number of examples for an anomaly type c. When performing the similarity calculation for a detected anomaly represented by a grid pattern, the anomaly type is set to unknown if the similarity to all known anomaly types lies below the respective β_c value. Otherwise, the anomaly type that yields the highest similarity is set as the predicted class. Thereby, its similarity value is set as the confidence ζ.

5.3 Evaluation Environment

This section describes the evaluation environment that is used to conduct experiments. We evaluate our method based on data collected during experiments. Thereby, a commodity cluster is used to deploy a cloud computing system, where services are deployed on virtual machines. An anomaly injection system is developed to disturb the normal operation of different components and observe the resulting behavior. The evaluation environment can be used in two operation modes. First, a defined experiment can be conducted while recording monitoring data. It allows us to evaluate our methods on the collected data retrospectively. Second, it enables the deployment of the evaluation environment alongside an AIOps system to perform live testing. This evaluation environment was used to conduct experiments for several publications[3, 23, 106–108, 209–212, 232, 238]. It consists of the cloud computing system and therein deployed services. The combination of both is considered to be the SuO. In the following subsections, we provide a detailed description of the experiment environment.

5.3.1 Cloud Computing System and Monitoring

The basis of all experiments is a commodity cluster of 200 nodes. Hardware specifications of the different resource types are listed in Table 5.1. Each node has two network ports that are used to set up an external and internal network. For the internal network, a two-level tree structure is used. The leaf level consists of six Netgear FS750-T2EU switches, each having 48 100MBit ports. All 200 nodes are connected to these switches. The root level is a Netgear GS716T-200 switch, which has 16 1GBit ports. All six leaf switches are connected to it. Internet connectivity is provided via the external network. All 200 nodes are connected to an HP 5412-92G-POE+-4G v2 switch with 203 1GBit ports.

Table 5.1: Hardware specifications of commodity cluster nodes.

Resource Type	Specification	Quantity
Processor	Intel Xeon Quadcore E3-1230 V2 3.30GHz 8 Cores	1
Main Memory	Samsung M393B5173FH0 4GB 1333MHz	4
Storage	Seagate ST31000524NS 1TB 3Gb/s	3
Network	Intel 82574L 1GBit Ethernet Controller	2

A subset of commodity cluster nodes is selected to experiment. These nodes are prepared by executing custom Ansible[1] playbooks that are configuring the network interfaces and installing

[1] `https://www.ansible.com/` (last access 31 May 2021)

packages. Ansible is an orchestration tool that enabled a declarative description of the desired node state. It configures the nodes into the defined state by connecting to them via SSH and executing a set of commands.

On the prepared nodes, we deploy OpenStack[2] as a cloud computing system. After Rackspace Hosting and NASA jointly launched it in July 2010, it evolved into a widely used open-source cloud computing project [214]. It consists of a growing number of sub-projects, where each provides specific functionality to the system. The core functionality of OpenStack is to provision, manage and provide access to (mostly virtual) resources for users. OpenStack is organized as a distributed system. External systems or users access functionalities via defined APIs, while RPC over a message queue is used to realized communication between internal OpenStack components. Some sub-projects develop non-core functionalities such as monitoring solutions, orchestration tools, or deployment methods.

The Kolla[3] project is used to set up OpenStack on the commodity cluster. It manages all components that provide a core functionality as Docker[4] containers, which should allow a convenient deployment and removal of OpenStack. Thereby, Ansible is used to prepare cluster nodes, configure and place OpenStack component containers, or undo these operations when removing OpenStack. Furthermore, we utilize the orchestration project Heat[5] to provision all necessary cloud resources for experiments. It implements a domain-specific language to write template files that define resources such as virtual machines and networks or storage and describe their topology. We refer to the result of such a procedure as the virtual environment. After that, custom Ansible playbooks are used to deploy different services on the provisioned virtual machines.

In real systems, direct access to components can be limited. Considering an IaaS model, cloud providers are not allowed to access the internals of customer VMs. The network provider can monitor generic metrics but usually are not aware of the communicating applications. During our experiments, generic (i.e., not application-specific) metrics such as CPU utilization, memory allocation, or different network traffic statistics are collected. Therefore, custom monitoring agents are deployed to collect metric data from commodity cluster nodes and virtual machines. Cluster node metrics are retrieved from the */proc* file system[6]. Virtual machine metrics are acquired by accessing the *libvirt*[7] API. The data collection frequency can be set to 2Hz with a CPU utilization overhead of $< 3\%$[109]. Monitoring data samples can be stored in text files or exposed via TCP or HTTP endpoints at runtime.

5.3.2 Virtual IP Multimedia Subsystem

Inspired by the success story of cloud computing, network operators are developing solutions to run network technology on commodity hardware by utilizing virtualization. This direction is referred to as network function virtualization (NFV), whereby network software logic that was traditionally integrated into specialized equipment is decoupled and provided as virtual network functions (VNFs) [56, 113]. Besides the benefits like horizontal scaling, faster time to

[2] `https://www.openstack.org/` (last access 31 May 2021)
[3] `https://wiki.openstack.org/wiki/Kolla` (last access 31 May 2021)
[4] `https://www.docker.com/` (last access 31 May 2021)
[5] `https://wiki.openstack.org/wiki/Heat` (last access 30 May 2021)
[6] `https://man7.org/linux/man-pages/man5/proc.5.html` (last access 31 May 2021)
[7] `https://libvirt.org/` (last access 31 May 2021)

market, customized solutions, and several more, the availability requirement for VNFs is hard to achieve [113]. Due to this, we consider NFV as a relevant use case to test our methods.

The IP multimedia subsystem (IMS) is a collection of different network functions that deliver multimedia services such as IP telephony, conference calls, or rich communication services to users [44]. It is responsible for user data management, mobility management, session life cycles, and billing. Project Clearwater[8] is a virtual implementation of an IMS that was maintained as an open-source project until December 2019. After that, it became a closed source project maintained by metaswitch and commercially offered to their customers[9]. It is developed as a microservice architecture consisting of seven services designed to be deployed on cloud systems. A short description of each service is given in Table 5.2.

Table 5.2: Project Clearwater services.

Service	Description
bono	Edge proxy for clients.
sprout	Session control, authentication & authorization.
homer	XDMS: User setting storage.
dime - homestead	HSS proxy/interface.
chronos	Timing management & billing.
cassandra (Homer, Dime)	Database for HSS and XDMS.
vellum - astaire	In-memory DB. State storage.

The session initiation protocol (SIP) is used to communicate between user devices and the IMS system. We utilize the tool SIPp[10] to generate load against the Clearwater services. It can perform requests such as registration, initialization of calls with other users, calls answering, and several others. We start multiple differently configured SIPp processes to simulate a variety of user behaviors. During experiments, the load generation is executed on dedicated commodity cluster nodes to minimize the interference between load generation and the Clearwater system.

5.3.3 Video on Demand Service

The majority of the Internet traffic is generated by videos with a share of up to 60% [82, 205]. Thereby, video call applications and on-demand video streaming are the main drivers. Due to the popularity of video on demand services and their relevance for the industry, we choose it as our second use case [102, 194]. It consists of two service types: video streaming server and load balancer. The content transmission is done via the Real-Time Messaging Protocol (RTMP) [182]. A client connects to one of the load balancers, which delegates the request to one of the available video streaming servers. After that, it receives the video stream and forwards it to the client. Load balancing is realized via ha-proxy [11] while video streaming is done via nginx[12] web servers with an RTMP plugin [13]. A library of 15 videos with different lengths and respectively different resolutions is provided and used as streaming content.

[8] `https://www.projectclearwater.org/` (last access 31 May 2021)
[9] `https://www.metaswitch.com/products/clearwater-core-ims` (last access 31 May 2021)
[10] `https://sourceforge.net/projects/sipp/` (last access 31 May 2021)
[11] `http://www.haproxy.org` (last access 30 May 2021)
[12] `https://www.nginx.com` (last access 30 May 2021)
[13] `https://github.com/arut/nginx-rtmp-module` (last access 30 May 2021)

To generate load, we implement a client simulation that requests videos from defined endpoints. It is possible to adjust the selection policy of videos (random or round-robin) and set a configurable resolution preference. Client processes offer HTTP APIs, which can be used to adjust configurations at runtime or request runtime metrics like the number of requested videos, video resolutions, or the number of processed frames and pixels. This data is used to observe how well the video on demand service is operating. Again, we separate the video on demand services from the client processes by deploying them on dedicated cluster nodes to minimize interference.

5.3.4 Anomaly Injection

Anomalies occur irregularly in functional systems. However, control and reproducibility are required to evaluate and compare fault tolerance methods. A widely applied solution is the synthetic injection of anomalies into systems that run real load [59, 62, 64, 74, 104, 154, 174, 208, 228, 236, 240] It enables the execution of reproducible experiments. Thereby, either the system configurations are adjusted, or processes start to simulate anomalous behavior while the system runs real load. Furthermore, it typically allows to revert the injection and bring the system back into a normal state.

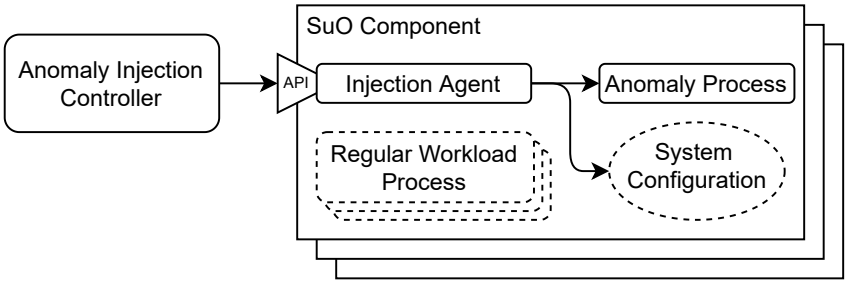


Figure 5.7: Agent-based anomaly injection.

We implement an agent-based anomaly injection system [14]. Its architecture is depicted in Figure 5.7. Injector agents are deployed on all relevant SuO components, i.e., on virtual machines or the commodity cluster nodes, while an anomaly injection controller runs on another dedicated cluster node. The agents expose an HTTP API that can be used to start or stop anomalies. Internally, they use two tools to realize anomalies. The stress-ng tool[15] is used to start processes that affect resources like CPU, memory, or disk I/O. The tcconfig library[16] configures the traffic shaping on network interfaces which is used to increase latencies, limit bandwidth or enable packet drop. Components can become unreachable from outside when certain anomalies are injected (e.g., high packet drop percentage). Therefore, the agents enable a configurable fall-back timer. On expiry, the respectively injected anomalies are reverted by the injection agents. A controller allows defining anomalies and injection schedules in order to run experiments. Anomaly intensities, duration, and temporal progress can be configured. The configuration enables the randomization of configuration parameters either via normal or

[14] Available at `https://github.com/dos-group/distributed-anomaly-injection` (last access 30 May 2021)

[15] `https://kernel.ubuntu.com/~cking/stress-ng/` (last access 30 May 2021)

[16] `https://github.com/thombashi/tcconfig` (last access 30 May 2021)

equal distribution. It is possible to draw intensities, durations, and other parameters from these distributions instead of setting them to a fixed value.

Table 5.3: Overview of anomalies.

Computation Resource Anomalies	
CPU	Perform operations on the CPU to achieve a configure utilization.
Memory	Allocate and eventually release a defined amount of memory.
Disk	Perform read and write operations on the disk.
Network	Send network traffic to or receive it from a defined destination.
Network Shaping Anomalies	
Latency	Delay ingress or egress packets.
Bandwidth	Limit the throughput.
Packet Loss	Drop packets with a configurable probability.
Packet Duplication	Duplicate a configurable percentage of egress packets.

An overview of the anomaly portfolio is listed in Table 5.3 Based on experiments that were conducted in related publications [62, 74, 154, 174, 228, 236, 240], we define three general types of anomalies that affect component resources: hogging, leakage, and fluctuation. Therefore, example injections could simulate a memory leak, CPU hog, network congestion, or fluctuating disk utilization. Table 5.3 also shows the network shaping anomalies. Thereby, anomalies like a slow or unreliable network can be simulated.

Since the start and end times of the injected anomalies are known, it is possible to label the data set after the experiment is finished. Samples within an injection period are labeled by the anomaly type that was injected. Samples outside of the injection periods are labeled as normal. These labels are the ground truth for the evaluations.

5.4 Evaluation

In this section, we evaluate our method. First, the evaluation scores used to test our method are described. After that, we describe two experiments conducted to collect metric data. These data are used to evaluate the anomaly symptom recognition method. Next, we evaluate the ability of our model to recognize anomaly type-specific symptoms and infer the anomaly type. Finally, we test whether the model can detect previously unseen anomaly types as "unknown". Our work is compared to two baseline methods.

5.4.1 Evaluation Scores

The anomaly symptom recognition is defined as a classification problem. Multiple anomaly types entail multiple classes, which results in a multi-class classification. We utilize a multi-class confusion matrix to define the evaluation scores for anomaly symptom recognition.

The multi-class confusion matrix is shown in Table 5.4. The rows represent class predictions, while columns are the actual classes. Given an example prediction c_i and the ground truth c_j, it is possible to assign it into the respective cell c_i, j. Based on this, we want to measure the

Table 5.4: Multi-class confusion matrix.

		Actual			
		Class_1	Class_2	...	Class_K
Predicted	Class_1	$c_{1,1}$	$c_{1,2}$	...	$c_{1,K}$
	Class_2	$c_{2,1}$	$c_{2,2}$	...	$c_{2,K}$
	...	...	...	...	...
	Class_K	$c_{K,1}$	$c_{K,2}$	...	$c_{K,K}$

fraction of correctly classified examples, which is referred to as *accuracy*:

$$Acc = \frac{1}{\sum_{i=1}^{K}\sum_{j=1}^{K} c_{i,j}} \sum_{k=1}^{K} c_{k,k}. \tag{5.18}$$

It calculates the ratio of the matrix diagonal sum, i.e., all elements that are correctly predicted, to the sum of all matrix cells, i.e., all predicted examples.

Furthermore, we use *precision*, *recall* and *F1 score* to evaluate our methods. The recall is defined respectively for each Class_i as the ration of correct predictions to all predictions of this certain class:

$$rec(c_{i,i}) = \frac{c_{i,i}}{\sum_{j=1}^{K} c_{j,i}}. \tag{5.19}$$

It can be intuitively described as the ability to not classify Class_i as another class. The precision is defined respectively for each Class_i as the ration of correct predictions of that class to the number of other class' mispredictions as Class_i:

$$pre(c_{i,i}) = \frac{c_{i,i}}{\sum_{j=1}^{K} c_{i,j}}. \tag{5.20}$$

It can be intuitively described as the ability to not misclassify other classes as Class_i. The F1 score is the harmonic mean between precision and recall:

$$f1(c_{i,i}) = \frac{2 \cdot pre(c_{i,i}) \cdot rec(c_{i,i})}{pre(c_{i,i}) + rec(c_{i,i})}. \tag{5.21}$$

Furthermore, we define the overall recall, precision, and F1 score by calculating the weighted average for all classes:

$$Sc = \sum_{i=1}^{K} w_i \cdot sc(c_{i,i}), \tag{5.22}$$

where Sc can be substituted with $\{Pre, Rec, F1\}$ and sc with $\{pre, rec, f1\}$ with the constraint that both substitution must select the same set index. The weight is the relative frequency of the respective class and can be calculated via $w_i = \frac{\sum_{\ell=1}^{K} c_{i,\ell}}{\sum_{j=1}^{K}\sum_{k=1}^{K} c_{j,k}}$.

5.4.2 Experiment

We use the experiment environment described in Section 5.3 to conduct two experiments, which are conducted on 25 commodity cluster nodes. For both experiments OpenStack Stein[17] is deployed on 21 nodes. Three nodes are load balanced with haproxy [18] and host the OpenStack network and controller services. Six nodes are used as storage, and another eleven as compute nodes. Out of the remaining five nodes four are reserved for load generation and one hosts the anomaly injection controller.

We deploy the virtual IMS Clearwater in virtual machines hosted on the eleven compute nodes in the first experiment. The number of components for each Clearwater service is listed in Table 5.5. The VMs are provided with one virtual CPU, 2GB RAM, and 40GB disk space. They operate with Ubuntu 16.04.3 LTS under Linux kernel version 4.4.0-128-generic. For simulating user traffic, one SIPp process is started on each node separated for workload generation. They are configured to jointly generate between 900 and 1200 user registrations and call initiations every second. All monitoring agents are configured to collect system metric data at a frequency of 2Hz. We collect the following metrics: CPU utilization in percent, the fraction of allocated memory in percent, the sum of disk read and write volume in bytes, and the sum of in- and out-going network traffic in bytes.

Table 5.5: Number of VMs for each Clearwater service.

Component	Bono	Sprout	Homer	Homestead	Chronos	Cassandra	Astaire
Num. of VMs	6	6	4	4	5	3	5

In the second experiment, we deploy the video on demand service. We deploy six VMs as service load balancers and another six VMs as video streaming servers on the eleven compute nodes. Each VM is provided with two virtual CPUs, 4GB RAM, and 80GB disk space. Again, Ubuntu 16.04.3 LTS under Linux kernel version 4.4.0-128-generic is used. The load system load is generated via four processes running on each of the reserved user load generation nodes. They simulate client access whereby between 100 and 150 clients should simultaneously request video streams. After finishing a video, a simulated client waits between 10 seconds and 5 minutes to select the next video. The video and its resolution are randomly chosen.

In order to evaluate the anomaly symptom recognition, several anomalies were injected into the respective service VMs. All anomalies together with their description and parametrization are listed in Table 5.6 To test whether our approach can recognize different anomaly occurrences of the same type, we inject each anomaly type several times into the same service. For Clearwater, we excluded Cassandra as an injection target. On all other Clearwater and video on demand services, each anomaly was injected 20 times. This resulted in six injection targets and overall 960 injections for the Clearwater experiment and two injection targets, and overall 320 injections for the video on demand experiment. The injection targets, e.g., a concrete Bono or video server VM, were selected randomly. The duration of every injection was set to 4 minutes, and 1 minute recovery period between the injections is defined. We use the start and end times of the anomaly injection for labeling. All metric series samples within this time range are labeled with the respective anomaly type. Additionally, we consider 25 normal samples before the respectively first anomalous sample to capture the transition from normal to an

[17] `https://releases.openstack.org/stein` (last access 30 May 2021)
[18] `http://www.haproxy.org/` (last access 30 May 2021)

anomaly state. Both experiments have an initial two hours without injections. The first and second experiments have an overall duration of 82 and 28.7 hours.

Table 5.6: Description and parametrization of injected anomalies.

	Description	
Anomaly	**Clearwater**	**Video on Demand**
CPU overutilization	Utilize all cores at 90 - 100%.	
CPU hogging	Step-wise increase the CPU utilization. Increase by 1% every 2 seconds until 90-100%.	
Memory leak	Step-wise allocate memory. x MB every y seconds until z MB.	
	x=3, y=1, z=500	x=6, y=1, z=1000
Memory overutilization	Allocate x MB of memory.	
	x=500	x=1000
Disk IO	Start 3 processes that write / read data to / from disk.	
Packet Loss	Drop 15 % of incoming and outgoing packets on all interfaces.	
Throttled Bandwidth	Limit bandwidth to 50 Kbps.	
Network Overutilization	Repeatedly start downloading a large file from the internet.	

5.4.3 Evaluating Anomaly Symptom Recognition

For the evaluation, we use the monitoring metric series that were labeled as anomalous. We test the ability of the anomaly symptom recognition to infer the anomaly type based on the symptom patterns in the metric data. Therefore, the available 20 examples of each injected anomaly are split in a training and test set. This is done by randomly selecting a fraction of examples for training while the remaining examples are left for testing. The training set is used to generate anomaly symptom-specific grid patterns. After that, the anomaly examples in the test set are transformed into grid patterns and compared to the patterns that were generated from the training set. The comparison is made individually for each service, i.e., a test pattern generated from Bono data is compared only with Bono training patterns. For each anomaly type, the highest similarity is used. The list of anomaly similarities is passed to the confidence estimation module, which filters entries bases on the β values. The interval parameter to discretize the input series is set to $h = 20$, and the decay is set to $\lambda = 0.9$. The neighborhood distance threshold γ is determined based on the available test data. It is set to minimize the average intra-class similarity and maximize the average inter-class similarity between the training patterns by searching the value range $1 \leq \gamma \leq \frac{h}{2}$.

The grid pattern (GP) model is compared to two baselines. For both, the time series of the training set is stored and used for comparison with the time series of the test set. The first baseline is calculating the average Euclidean distance between each data point of two time series. A varying number of samples is handled via truncation to the shorter of both series. The second baseline calculates the distance between two time series via dynamic time warping (DTW) [26]. DTW is considered a strong baseline in many state-of-the-art publications on time series analysis [40, 80] while the Euclidean model is often used for comparison with DTW [72, 127]. The DTW model has a hyperparameter ω, which is the search window through the warping matrix. We set this parameter the same as the γ parameter for GP, based on the available training examples. The anomaly symptom recognition for the baselines is performed

by the same procedure as for density grid patterns. Considering an anomaly series that should be type matched, we use the smallest distance to all available examples of each anomaly. This list of distances is passed to the confidence estimation module, which filters entries bases on the β values. The β values are determined as described in Section 5.2, but for small distance instead of high similarity values.

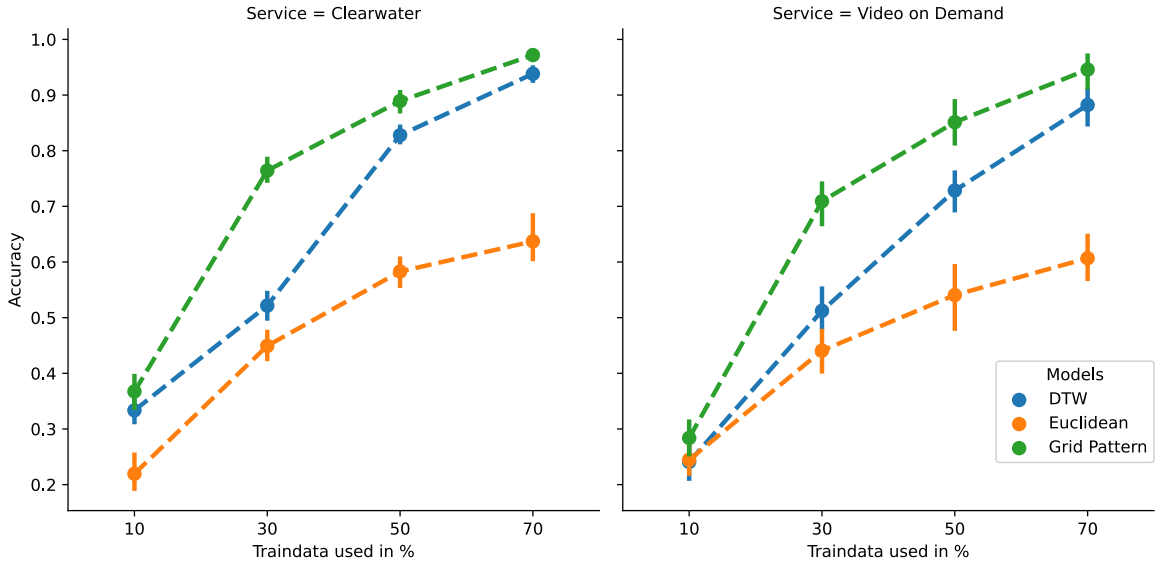


Figure 5.8: Accuracy results for different training / test data splits.

First, we evaluate the ability of the models to accurately match observed symptoms with existing anomaly type symptoms to infer the anomaly type. This is expressed by the accuracy, which is depicted in Figure 5.8. A detailed list of results for each service component can be found in Section A.1. The two plots show respective results for the two services: Clearwater is shown on the left, and video on demand is shown on the right. We analyze different splits between training and test data. Thereby, 10%, 30%, 50%, or 70% of the examples were used as training data, while the remaining examples are test data. To mitigate the bias of a concrete split, we apply k-fold cross-validation with $k = 20$. The accuracy values are averaged over all folds and service components. The error bars above and below indicate the 0.95 confidence interval.

The DTW and GP models outperform the Euclidean distance model. It can be explained due to the inability of the point-wise comparison to tolerate noise, time shifts, or symptom shape variations. Comparing the DTW and GP models, GP yields a higher mean accuracy for all splits. However, both models yield close accuracy values for the 30 % and 70 % splits. For Clearwater, six examples (30 % split) of each anomaly type are required for the GP model to reaches an accuracy value of 0.76. Increasing it to 10 and 14 examples (70 % and 50 % split) results in accuracy values of 0.89 and 0.97. Equivalently, the DTW model achieves accuracy values of 0.52, 0.83, and 0.94. Similar trajectories are observable for video on demand. The GP model achieves accuracies of 0.71, 0.85, and 0.94 for 30%, 50% and 70% splits compared to 0.51, 0.73, and 0.94 achieved by the DTW model. Generally, the GP model can achieve higher scores with less amount of training data than the DTW model for both service scenarios.

Besides the accuracy, we want to analyze the recall, precision, and F1 score. These are

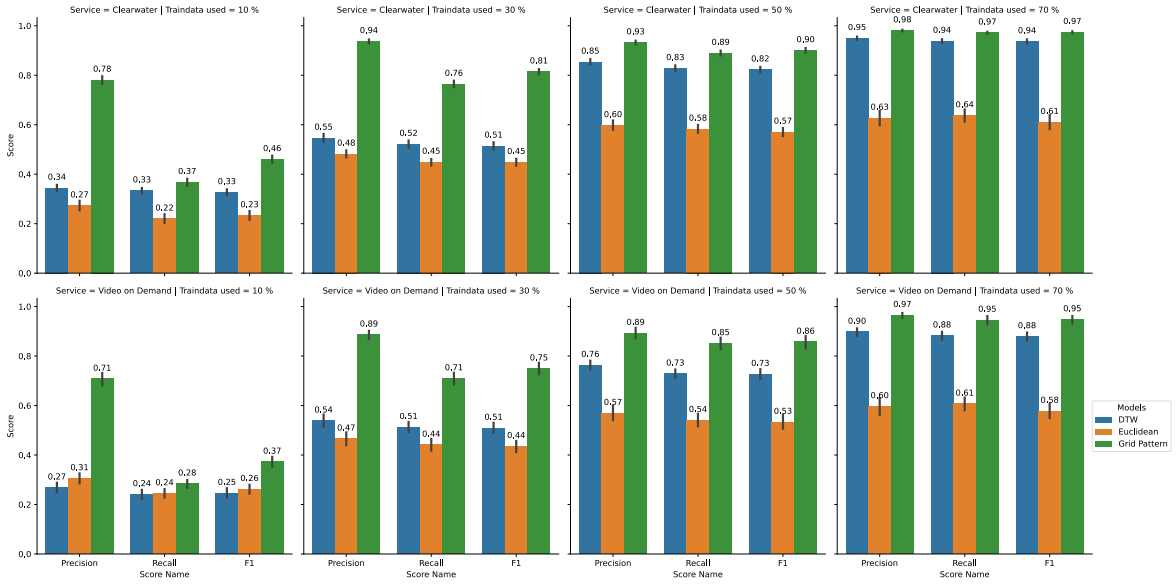
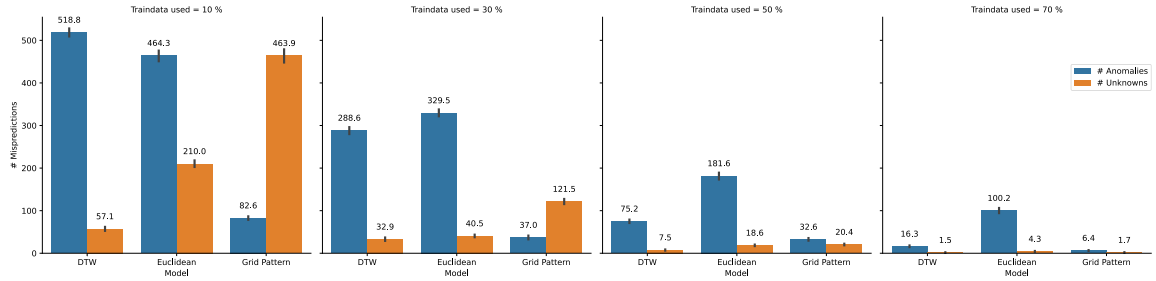


Figure 5.9: Precision, recall, and F1 scores.

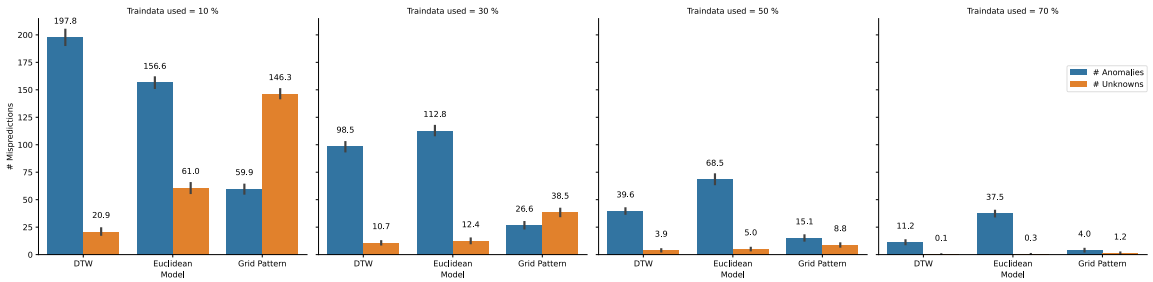
depicted in Figure 5.9. The plots are organized in a grid with four columns and two rows. Each column contains the result for a specific split (left to right 10%, 30%, 50%, and 70%). The top and the bottom row contain the results of Clearwater and video on demand. The bars are grouped by the score type, i.e., precision, recall, and F1 score, where each bar represents a respective model. Again, increasing score values can be observed with an increasing amount of available training data. From the 30% split and beyond, the GP and DTW models outperform the Euclidean distance model across all scores. Also, scores for the GP model are generally increasing faster than the DTW model scores but end up in similar ranges at the 70% split.

The larger precision scores for the GP model, especially in the lower splits, are apparent. For 10% and 30% splits, the GP precisions are 2 and 1.5 times higher than the DTW precision values. Note that the scores are weighted averages over the respective scores for each class, where the weights are the relative frequency of each class. Although anomaly examples within the test set can be classified as "unknown", no "unknown" class examples exist because no "unknown" anomalies were injected during the experiment. The weight for the "unknown" class is therefore 0. The "unknown" predictions are considered in the recall calculation but are nullified in the calculation of the precision since no "unknown" class can be mispredicted. The precision of the "unknown" class itself is nullified due to the zero-weight. Both aspects result in high precision scores if most mispredicted examples are predicted as "unknown".

This supposition is confirmed when analyzing the incorrectly predicted examples (see Figure 5.10a for Clearwater and Figure 5.10b for video on demand). The two plots show the distribution of incorrectly predicted examples between the "unknown" class (orange bar) and all other classes (blue bar). The error bars represent 0.95 confidence intervals. Note that the y-axes are differently scaled for each service due to the different amount of available examples, i.e., all anomalies are injected 20 times into all services, but Clearwater has six and video on demand two service components. Again, the columns represent the split from 10% until 70%, and both bars are grouped based on the model. By comparing the 10% distributions for Clearwater, it is evident that the GP model yields the majority of mispredictions as "unknown". In



(a) Distribution of incorrectly predicted classes for Clearwater.



(b) Distribution of incorrectly predicted classes for video on demand.

contrast to that, the majority of DTW mispredictions are other classes than "unknown". As expected, the amount of incorrect predictions decreases with increasing training data split percentage. However, the ratio between "unknown" and other class mispredictions stays in favor of other classes for the DTW model. In contrast to that, GP initially yields "unknown" mispredictions but changes after the 30% split to misclassify more examples as other classes than "unknown". These misclassification distributions for the DTW model are because the DTW is created to find short distances between time series. Instead of yielding "unknown" predictions, it incorrectly predicts examples as other existing anomaly types instead of "unknown". This behavior indicates a sparsity of regions within the input space that are not representing any class. The behavior of the GP model can be explained by tight decision boundaries around each class. Examples outside of these boundaries are classified as "unknown". The boundaries are adjusted when more training examples are available. However, some examples appear in regions of other classes. The amount of these examples is decreasing slower than the amount of "unknown" misclassifications, which indicates the existence of examples that are generally difficult to classify.

The anomaly type matches should be used to automatically select operations to resolve anomalies. If a model misclassifies an anomaly as "unknown", no operations can be automatically selected. A human operator is requested to resolve the anomaly. If a model incorrectly predicts an anomaly as some other anomaly type, operations that probably will not resolve the anomaly are selected. The execution of such operations can aggravate the anomaly, leaving the component in a state that is even harder to recover. From a practical perspective, a model that can admit its inability to perform a correct anomaly symptom recognition should be favored over a model that mispredicts examples as other anomaly types.

5.4.4 Evaluating Unknown Anomaly Type Recognition

Distributed systems are constantly changing, which potentially introduces novel anomalies. The knowledge base of anomaly types must be assumed as incomplete at all times. Besides recognizing known anomaly symptoms, an anomaly symptom recognition method should be able to identify a symptom pattern as unknown if the respective anomaly type never occurred before. Therefore, we evaluate the accuracy of the GP and the two baseline models to classify metric anomaly data as "unknown" if they do not occur in the training set.

We apply the same model parametrization as in Subsection 5.4.3. The metric series that are labeled as anomalous is used. The available 20 examples of each anomaly injected into each service are randomly split into a training and test set. This splitting is done respectively for each anomaly, i.e., identical amounts of training and test examples are available for each anomaly after the split is applied. For this evaluation, we define a constraint whereby the training set contains all but one anomaly type. This means that we exclude one anomaly type from the training set and move all examples of this excluded type to the test set. The task of the models is to predict the excluded anomaly type examples as "unknown". We repeatedly apply this procedure for each anomaly type while setting the training data fractions to 10%, 30%, 50%, and 70%. Ten folds were executed for each split to analyze the dependence on the selection of training examples, i.e., for each fold, respective training examples were randomly chosen.

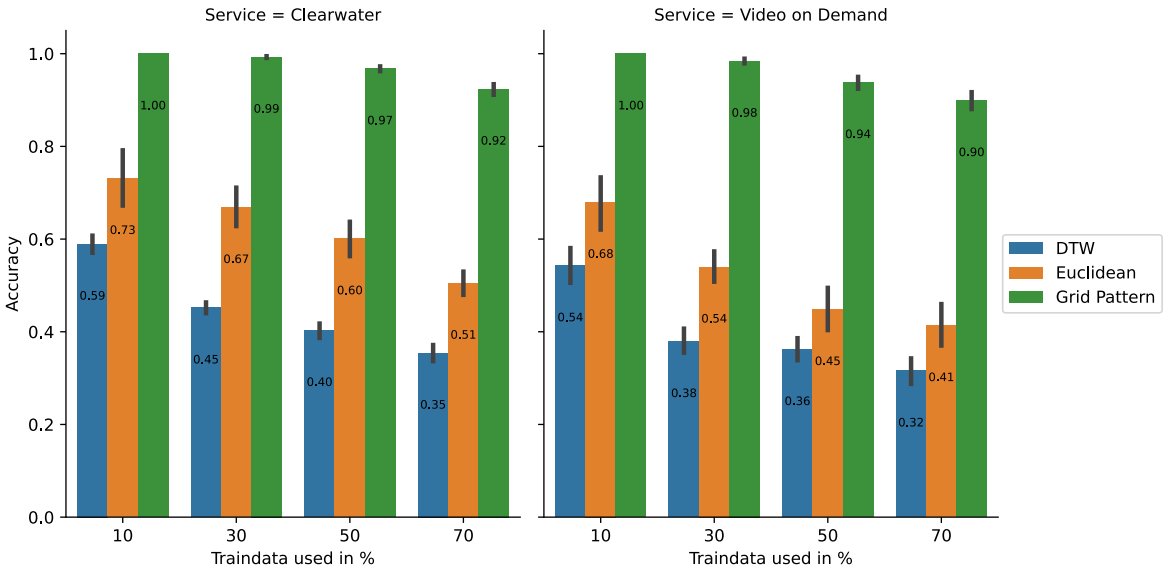


Figure 5.11: Accuracy of "unknown" anomaly prediction.

The results are depicted in Figure 5.11. A detailed list of results for each service component can be found in Section A.2. Again, the results for Clearwater and video on demand are respectively shown in the two bar plots. The bars are grouped by the splits, and each bar in a group represents the mean accuracy value of the respective model. Error bars are showing the 0.95 confidence intervals. It can be observed that the accuracy generally decreases as more training examples are available. This is due to the initial sparsity of decision regions in the input space if few examples are available. When more examples are observed, the decision regions grow and eventually cover areas of the "unknown" anomaly. The GP model's accuracy is the highest

on all splits and across the two services. The DTW model is designed to tolerate variances and shifts in the time series, which leads to less decision region sparsity in the input data space. This results in smaller distance values than the calculated β values, appearing in a non-empty result list for "unknown" test examples. The accuracy values for the Euclidean distance model are usually higher than DTW but lower than the GP accuracies. It has less tolerance to noise and intra-class variations than the other models, resulting in a higher dependence on the split, which is indicated by the longer error bars. For Clearwater and video on demand, GP yields a 1.0 accuracies for 10% splits, which is expected since only a few training examples are available to define the decision regions. The accuracy decreases when more training examples are available and reaches 0.97 and 0.92 on 50% and 70% splits for Clearwater and 0.94 and 0.90 for video on demand. We conclude that the GP model can define tight decision regions and recognize unseen anomaly types as "unknown".

Chapter 6

Recurrent Neural Network Model

Analyzing metric data to identify specific symptom patterns of anomaly types is difficult. It is especially difficult when system components execute heterogeneous workloads (e.g., a compute node hosting virtual machines that contain different services). For such components, the metric data time series exhibit complex temporal relationships and stochasticity [180]. This chapter proposes a method that realizes the recognition of anomaly type symptoms via a recurrent neural network (RNN) based on gated recurrent units (GRUs). It is designed to capture complex dependencies in sequential data in the presence of noise and intra-type variation. However, the ability of GRU RNNs to capture such complex dependencies makes them prone to overfitting, especially when few training examples are available [207]. Therefore, a method for time series augmentation is proposed that stabilizes the training procedure of the model and mitigates the overfitting problem.

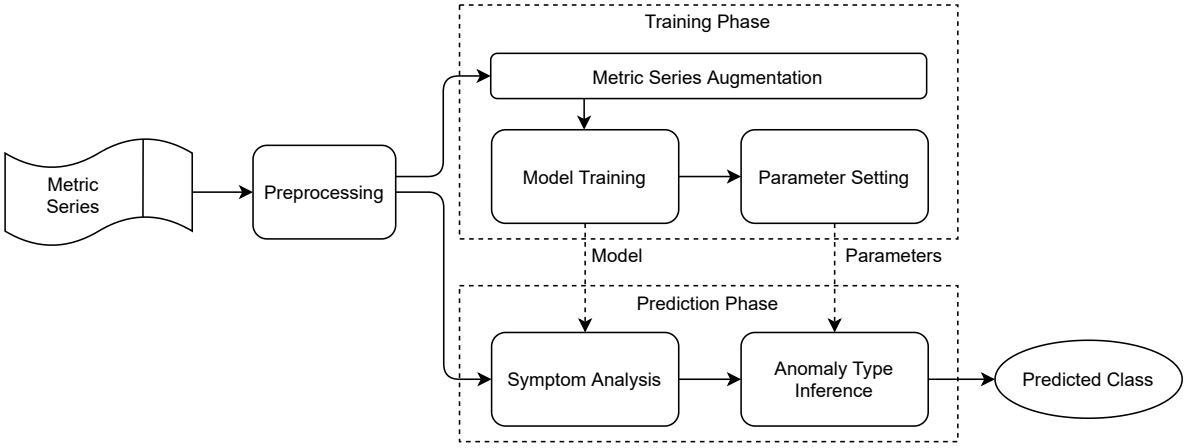


Figure 6.1: General workflow of the proposed method.

The workflow of our method is depicted in Figure 6.1. Metric series are preprocessed first. After that, the method is split into two phases: training and prediction. During training, the metric series augmentation generates metric series data based on available training examples. Thereby, the symptom analysis model, i.e., the GRU RNN, is trained first. After that, model prediction results on augmented examples are used to set the parameters of the anomaly type inference. Next, the trained model, together with the parameters, is deployed for prediction. The GRU RNN yields prediction results for each received monitoring series sample while the

anomaly type inference analyses the prediction sequences and infers the anomaly type. The anomaly type options include the known anomaly types and "unknown" representing the recognition of yet unknown symptom patterns.

6.1 Anomaly Symptom Analysis

To match anomaly types based with specific symptom patterns in metric data, we consider the temporally ordered sequence of samples $X_{t_0:t_c} = \{X_{t_0}, X_{t_1}, \ldots, X_{t_c}\}$ at time t_c. This means that a set of samples from the time of anomaly detection t_0 up until t_c are considered to perform the anomaly symptom recognition. The symptom analysis is applied on each X_t, resulting in a prediction function $f_C(X_t, \theta) : \mathbb{R}^d \mapsto \mathbf{y}_t$. Thereby, θ are parameters that should be adjusted based on available class examples of anomaly types and an objective function. We use a GRU RNN to model the function f_C. The prediction results $\mathbf{y}_t$ contain a set of class prediction targets C_k, resulting in $|\mathbf{y}_t| = K_k$. Each element $y_i \in \mathbf{y}_t$ is a probability value $0 \leq y_i \leq 1$ that indicates the association to a certain class. The y_i values can be expressed as conditional probabilities $y_i = P(c_i|X_{t_0:t})$. Note that the GRU RNN considers the current metric series sample X_t and a compact representation of all processed samples before t. This allows to condition the prediction on the whole series $X_{t_0:t_c}$. We model $\mathbf{y}_t$ as a posterior joint probability which means that all element values must sum up to 1.

6.1.1 Model Architecture

After detecting an anomaly, the anomaly symptom recognition constantly receives monitoring metric samples whenever they are available. With each received sample, the symptom analysis calculates the association probabilities to all known anomaly types. Therefore, the model must be applied to a sequence of monitoring series samples. We propose a sequence-to-sequence prediction modeling that yields scores at each time step t. Thereby, a metric sample X_t and a compact representation of all received samples up to time t is used to calculate the prediction. The architecture of our model is depicted in Figure 6.2. In the following, we describe each model component in detail.

Preprocessing

The metric data samples are rescaled to a range of $(0,1)$. This rescaling is done within the ingest control but is shown here for the sake of completeness. It is important to adjust the model parameters θ in a well-conditioned manner, which is crucial for the parameter values to converge [129]. We apply rescaling individually on each series via

$$\hat{X}_t^i = \frac{X_t^i - X_{min}^i}{X_{max}^i - X_{min}^i}. \tag{6.1}$$

Most system component resource metrics' upper and lower boundaries are well-known (e.g., number of CPU cores and their frequency or maximum available memory). However, for some metrics like latencies between network endpoints, the number of context switches, or packet errors, it is challenging to decide on upper or lower boundaries. If the boundaries are not known, X_{min}^i and X_{max}^i are determined from the available training examples and used during prediction.

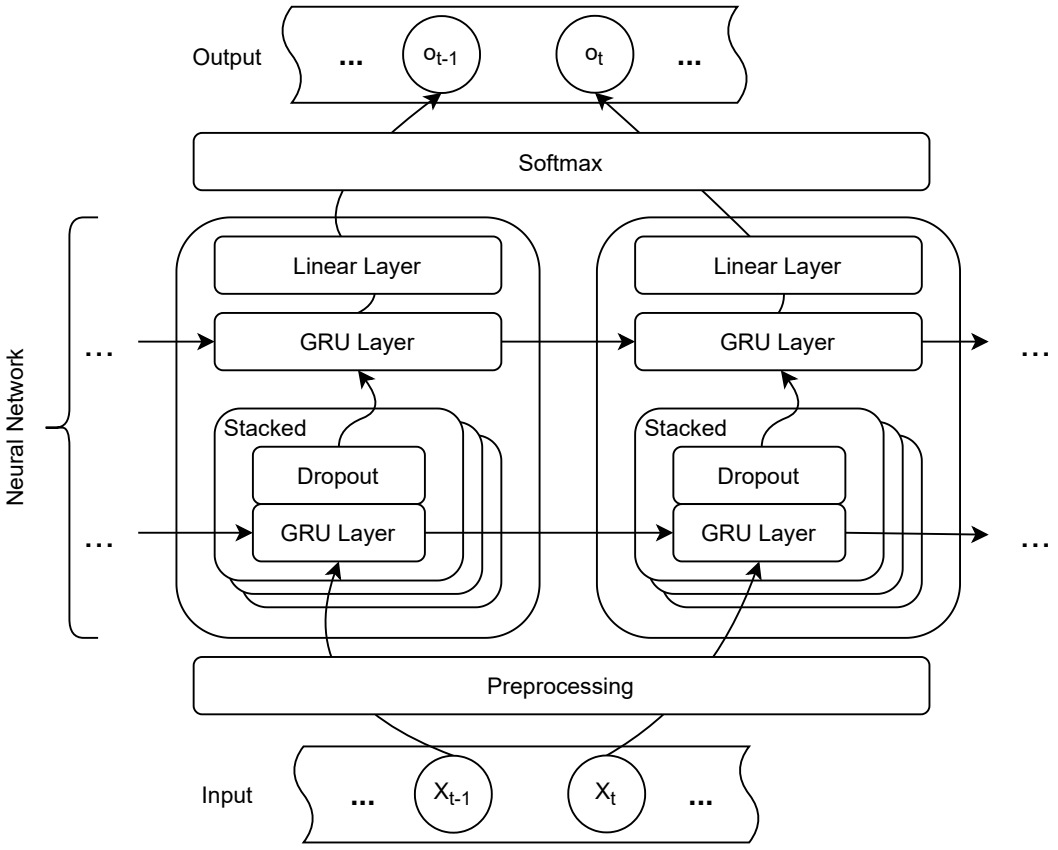


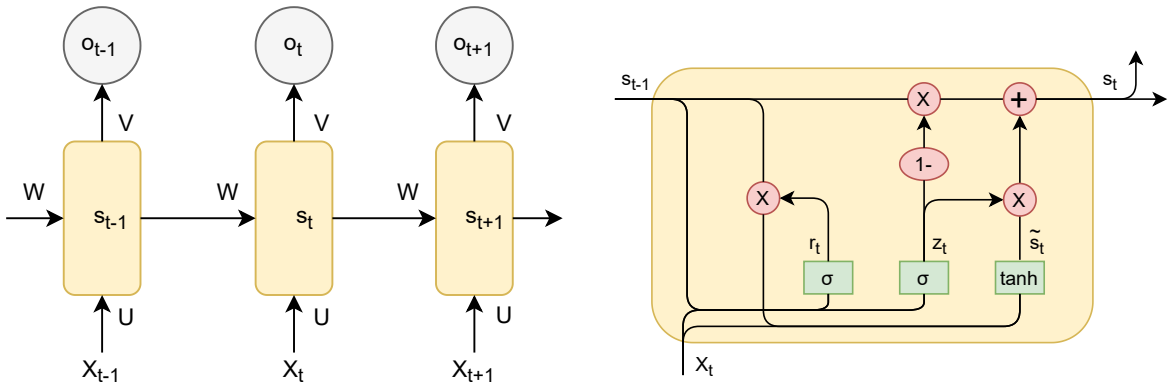
Figure 6.2: Sequential model architecture for anomaly symptom recognition.

Recurrent Neural Network Structure

The neural network with parameters θ is the central component of the symptom analysis model. It must retain the temporal information of the metric data series by considering the sequential dependencies between consecutive samples. We utilize a recurrent neural network that has sequential connections between neuron outputs [199]. This conditioning is realized by maintaining an internal state, which is a compact representation of preceding samples. A general RNN structure is depicted in Figure 6.3a. At a time step t the model receives a metric data sample X_t and the previous state s_{t-1}. The state contains information about all metric data samples up to t. RNNs can handle sequential data, where sequences can have varying lengths, which fits our requirement of constantly received metric data samples until an anomaly is resolved.

The basic RNNs suffer from the issue of vanishing gradients [117]. Common solutions for this problem are either LSTM (Long Short-Term Memory) [118] or GRU (Gated Recurrent Unit) [57] cells. Empirical evaluation shows that both cell types perform similarly on many datasets [58]. Since a GRU cell has one parameter less than an LSTM cell, we use GRU cells[1]. Figure 6.3b shows the internals of an GRU cell. It requires the previous state value s_{t-1} and a metric data sample X_t as input. Its internal structure routes the values through different gates to eventually calculate the state s_t. For GRU cells, the state s_t is also the output o_t. We show the

[1]Note that neural networks usually consist of hundreds or thousands cells, which results in a respective scale of the one parameter less.



(a) General RNN structure. (b) GRU cell.

Figure 6.3: RNN and GRU cell structure.

formal definitions and explain the intuition behind each gate. The update gate z_t is defines as

$$z_t = \sigma(UX_t + Ws_{t-1}). \tag{6.2}$$

Thereby, the input sample and state values are multiplied with their respective weights. These weights are part of the model parameters θ and adjusted during model training. Intuitively, the update gate controls how the current information contained in X_t is combined with information from past samples that are compactly represented by s_{t-1}. The reset gate r_t is computed similarly but uses different weights for s_{t-1}

$$r_t = \sigma(UX_t + Vs_{t-1}). \tag{6.3}$$

Intuitively, it controls the model's decision to forget past information. The internal memory $\tilde{s}_t$ is defined as

$$\tilde{s}_t = \tanh(UX_t + r_t \circ Vs_{t-1}). \tag{6.4}$$

It combines the information contained in the current metric sample and the previous state scaled by the reset gate. The scaling is done via the Hadamard product of the two vectors. If the reset gate contained only zero-values, the current state would contain recent information only. The output s_t is a combination of both the previous state and the current internal state

$$s_t = z_t \circ s_{t-1} + (1 - z_t) \circ \tilde{s}_t. \tag{6.5}$$

Directly scaling the previous state s_{t-1} without any weight multiplication serves as a shortcut for the back-propagation and mitigates the problem of a vanishing gradient. Furthermore, when values of z_t are close to one, it allows information from previous samples to be passed directly to the next time step, enabling the use of past information to make predictions. Contrary, z_t values close to zero enable the model to use the most recent information to make predictions. Routing information through these gates enables GRU cells to consider short- and long-term dependencies.

In our model, we use GRU layers, a set of GRU cells connected with either the input metric samples or the outputs of the previous layer's GRU cells. The architecture of the neural network is depicted in Figure 6.2. By unfolding the neural network, it is visually shown that the incorporation of the previous GRU states allows the model to condition the current prediction

on a representation of preceding time series samples and the current sample. This way, the salient information from the past samples can impact the prediction at each time t and identify anomaly type-specific symptom patterns in the metric series. The first part of the neural network consists of a combination of a GRU layer with dropout. This combination can be arbitrarily stacked, resulting in a configurable amount of GRU+dropout layers. Dropout is a regularization mechanism to mitigate overfitting [221]. It randomly disables a defined fraction of connections between layers, i.e., withholds part of the information from the respective model layer. Dropout is applied only during model training. Throughout our experiments, we set the dropout to 0.3. The neural network model is terminated with a combination of a GRU and linear layer. The linear layer adjusts the output dimensionality to K_k outputs, where each element in the output vector is a prior probability value representing the associational of the samples to an anomaly type.

Calculating Posterior Probability via Softmax

The output of the linear layer is a vector $\mathbf{y}_t$ with K_k elements, where each element value is the prior probability at time t. It denotes that the currently observed metric data series is a manifestation of a specific anomaly type. This means that each value in this vector is associated with one of the K_k classes. We utilize the softmax normalization [37] to calculate the posterior probability values. Thereby, the values are scaled to a range of $(0,1)$ and should sum up to 1, i.e., $\forall y_i \in \mathbf{y_t} : 0 \leq y_i \leq 1 \wedge \sum_i y_i = 1$. To simplify the notation we omit the time index t for the y_i elements. The values y_i are calculated by

$$y_i = \frac{e^{y_i}}{\sum_{j=1}^{K_k} e^{y_j}}. \tag{6.6}$$

It is applied on each element in $\mathbf{y}_t$ yields a probability distribution over all K_k classes. The higher the value, the stronger the association with the respective anomaly type.

6.1.2 Model Training

The objective of the model is to combine an input sample X_t with the state s_{t-1} and transform them to posterior probability values that represent associations with the respective class prediction targets. Considering the function $\mathbf{y}_t = f_C(X_t, \theta)$ and the fact that X_t is given, the parameters θ should be adjusted during the training of the model. The parameters of neural networks are adjusted via stochastic gradient decent[196] and backpropagation [147]. It requires available examples, i.e., metric series, for which the class label, i.e., anomaly type, is known and an error function that quantifies the correctness of the prediction. It allows to gradually adjust the θ values of the neural network via

$$\theta_{i+1} = \theta_i - \alpha \frac{\partial E(f_C(X_t, \theta_i), \mathbf{y}_t^*)}{\partial \theta_i}. \tag{6.7}$$

The network yields a prediction for an example X_t based on the current parameters θ. The availability of the ground truth $\mathbf{y}_t^*$ allows calculating the error E between the prediction and the ground truth. Calculating the partial derivative with respect to θ_i gives the gradient, which defines how the respective values of θ_i must be changed to reduce the error. The parameter α

controls the change of the parameters to prevent a highly fluctuating gradient. The result is an updated set of parameters θ_{i+1}. Since neural networks combine several linear and non-linear functions, the chain rule is applied to calculate the partial derivatives. The chain rule application is referred to as backpropagation. It is computationally expensive to execute gradient decent calculations. Therefore, averaging the error of a sample set and jointly calculating the backpropagation instead of doing this for each sample is a method to reduce the overhead. The set of samples is thereby referred to as *batch*, and the number of samples in a batch is the *batch size*. A neural network must process the available training data several times until the parameter values converge. Processing a number of batches selected from the available training examples is referred to as training epoch.

During training, a set of metric series examples with known class affinity is available. The affinity is represented by the ground truth vector $\mathbf{y}_t^*$ that contains the value 1 for the class with which the metric series sample X_t should be associated while all other values are set to zero. The cross-entropy loss is utilized to measure the error between the ground truth and the prediction:

$$E = -\sum_{i}^{K_k} y_{i,c}^* \log(y_{i,c}). \tag{6.8}$$

Having multiple classes as prediction targets, the predictions y_i for which the ground truth value is 1 are considered. These are indicated by c. All others are nullified due to the zero-multiplication by $y_{i,c}^*$ and, thus, do not need to be considered in the summation. Considering a single $y_{i,c}^*$ and $y_{i,c}$ pair, the cross entropy loss yields is zero if the predicted probability $y_{i,c}$ is 1. Note that this is expected since it matches the ground truth probability $y_{i,c}^*$, which is also 1. The progress of an increasing divergence between the expected and actual probability is depicted in Figure 6.4. It can be seen that the loss approaches infinity if the predicted loss approaches 0, although 1 is expected.

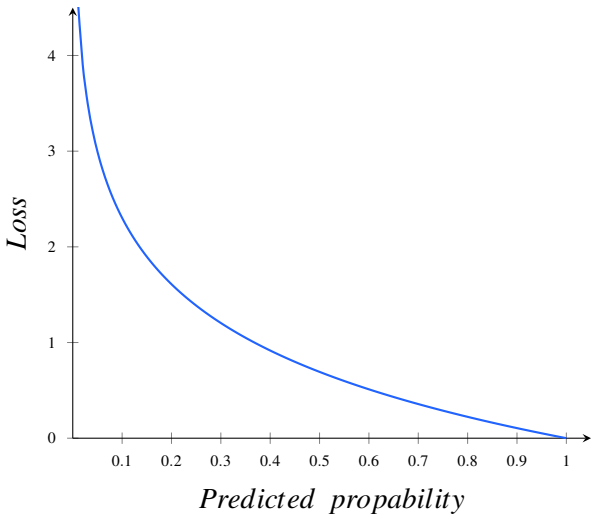


Figure 6.4: Visualization of the cross entropy loss for the ground truth value 1.

The GRU RNN should capture the sequential dependencies between metric series samples. During training, we select a metric series that should be associated with an anomaly type and provide the samples $X_{t_o:t_c}$ in sequential order to the model. After one series is processed, we

reset the internal state of the GRU cells to randomly initialized values. The input series must not be truncated or padded to a uniform size since the GRU RNN is used as a sequence to sequence model. Therefore, the whole metric series of an anomaly example is used as a batch, whereby the model should output an anomaly class prediction for each sample in the series. The backpropagation is computed after processing the complete series. The model is constantly trained for 300 epochs with a learning rate of $\alpha = 10^{-3}$.

6.2 Anomaly Type Inference

The anomaly type selector receives the $\mathbf{y}_t$ results from the symptom analysis module. The supervised training of the RNN does not allow to directly predict a probability value for the "unknown" class. No examples are available for anomaly types that were not observed yet. Furthermore, the anomaly type selector receives a prediction for each observed monitoring data sample at time t resulting in multiple prediction results for the duration of an anomaly. These results should be aggregated to yield one anomaly type eventually. Based on this, the selector has two tasks. First, it should analyze the received results and infer whether the observed anomaly does not match any known anomaly types. If this is the case, the observed anomaly type should be stated as "unknown". Second, the observed results over time should be aggregated to eventually yield one anomaly type decision together with a value that represents the confidence ζ.

The "unknown" class represents all areas in the input data space that are not associated with known anomaly types. Since such anomalies were not observed yet, it impossible to train the model explicitly on such examples. Therefore, postprocessing of the posterior probability values should infer this information. The intuition behind this approach is that if the model cannot assign a high probability value to one of the classes but similar probabilities to several classes, the observed symptoms indicate a yet unknown anomaly type. Similar probability values across multiple classes result in an increasingly uniform posterior distribution. We utilize the Gini coefficient[231] to measure the uniformity of the predicted probability values. It is defined as

$$G(\mathbf{y}_t) = \frac{\sum_{y_i \in \mathbf{y}_t} \sum_{y_j \in \mathbf{y}_t} |y_i - y_j|}{2(K_k)^2 \bar{y}_t}, \tag{6.9}$$

where $\bar{y}_t = \frac{1}{K_k} \sum_{y_i \in \mathbf{y}_t} y_i$. It represents the average of element-wise differences for each existing element pair, normalized by the average of all elements. The elements are the predicted probability values at time t. The range of G is $(0,1)$ with 0 representing an uniform distribution. G approaching 1 represents an increasing non-uniformity. Based on this, we interpret the Gini coefficient values that are closer to 0 as an indicator that the current metric data are representing a yet unknown anomaly type.

The next step is to decide on a class $\hat{y}_t$ based on the symptom analysis results $\mathbf{y}_t$ and the Gini coefficient. Whether to set the result to "unknown" is decided based on a threshold value ε for G. If the result is not interpreted as "unknown", the class with the highest probability value is selected as the predicted class. This is defined as

$$\hat{\mathbf{y}}_t = \begin{cases} c_u & \text{, if } G(\mathbf{y}_t) \leq \varepsilon \\ \arg\max_{C_k}(\mathbf{y}_t) & \text{, otherwise.} \end{cases} \tag{6.10}$$

Thereby $\hat{\mathbf{y}}_t$ represents the prediction result at time t, which can be a known anomaly type, the normal state, or a yet unknown anomaly type. Following the 1-of-K coding scheme, where $K = K_k + 1$, $\hat{\mathbf{y}}$ is a vector where each element is associated with a class. In $\hat{\mathbf{y}}_t$, one element has the value 1 while all others are 0.

The anomaly symptom recognition results should be used to automatically select operations to resolve an anomaly. A decision on an anomaly type initializes the selection process. Therefore, it is unreasonable to yield an anomaly type match after each received metric data sample, i.e., forward all $\hat{\mathbf{y}}_t$. The second task of the anomaly type inference is to aggregate multiple prediction results from the symptom analysis and eventually yield a final anomaly symptom recognition decision consisting of a class and a confidence value. This is done after receiving τ prediction results. First, we calculate the sum of all prediction vectors

$$\hat{\mathbf{y}} = \sum_{t_0 \leq t \leq \tau} \hat{\mathbf{y}}_t. \tag{6.11}$$

The class decision is done by $y = \arg\max_C(\hat{\mathbf{y}})$, which is effectively a majority voting. The confidence value for the result is defined as the relative frequency of the predicted class y. If i_c is the index of the predicted class in $\hat{\mathbf{y}}$, the confidence ζ is formally defined as

$$\zeta = \frac{\hat{y}_{i_c}}{\sum_{\hat{y}_j \in \hat{\mathbf{y}}} \hat{y}_j}, \text{ where } \hat{y}_{i_c} \in \hat{\mathbf{y}}. \tag{6.12}$$

The intuition is that the prediction result can be trusted if the model frequently yields the same class prediction. A prediction that fluctuates across different classes is assumed to be less trustworthy.

Both parameter ε and τ are determined during the training phase after the training of the RNN model. Thereby, the metric series augmentation generates a number of examples for classes that represent anomaly types. For each example, the model yields predictions $\mathbf{y}_t$. The parameter ε is determined first. For all $\mathbf{y}_t$, the Gini coefficient $G_t(\mathbf{y}_t)$ is calculated. The threshold ε is defined individually for each class as the average Gini coefficient for all $\mathbf{y}_t$ predictions and all augmented examples of this class. After that, the parameter τ is calculated individually for each class. It is set such as for the anomaly type inference to yield a correct decision with the minimum amount of $\hat{\mathbf{y}}_t$ predictions, emphasizing a timely but correct decision. Therefore, the smallest t with the highest number of correct predictions is set as τ.

6.3 Metric Series Augmentation

Metric data collected from IT system components generally contain variations due to internal and external factors that affect the processing. These variations affect the anomaly type-specific symptom patterns, resulting in variations between the examples and general noise. An anomaly symptom recognition model should be able to cope with intra-type variation and tolerate noise.

Neural networks in general, and thus RNNs, can learn complex dependencies within the input data. On the one hand, it allows the proposed RNN to identify complex symptom patterns of anomaly types, but on the other hand, it is prone to learn an exact or close to the exact representation of the observed metric series. The latter aspect is referred to as overfitting. Thereby, noise within the training data is learned as part of the classification information, which deteriorates the ability of the model to make predictions on unseen examples [230]. Overfitting occurs

when the model contains more parameters than justified by the available training data. From this statement, two mitigation strategies can be inferred. Either the amount of model parameters is treated as a hyperparameter and must be adjusted based on the available training data, or the training data is manipulated to prevent the model from learning the exact representation of available examples. Former requires to split available examples into a training and validation set, whereby the model is trained with different amounts of parameters and validated to identify an appropriate setting [28]. The splitting of available data is problematic if only a few examples are available. Latter is referred to as augmentation. Thereby, synthetic examples are generated that follow the distribution of the available training data. They resemble them, such as allowing the model to learn the representation but differ in a way that prevents the model from learning the exact representation, i.e., contained noise. Furthermore, an augmentation method can manipulate existing examples to better generalize the model on unseen examples.

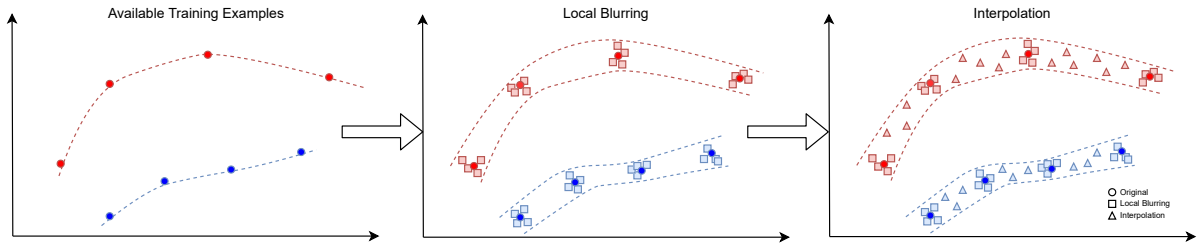


Figure 6.5: Conceptual visualization of the proposed metric data augmentation method.

We propose an augmentation method for metric data to generate synthetic metric series based on available examples. It consists of two steps that are conceptually depicted in Figure 6.5. The plots contain training examples (original and augmented) from two different classes. No augmentation is performed on the data in the left plot. These are the available examples to train the model. The dotted line represents a trajectory within the input space that indicates the decision region's location for the respective class. We refer to the first augmentation step as local blurring. Thereby, the noise values of an example are changed to prevent the model from learning an exact representation of this noise. The second augmentation step is referred to as interpolation. Two neighboring locally blurred examples are selected as references. Based on them, an augmented example between these references is generated. This should allow the RNN model to assign decision regions that are not covered by the real examples to the respective class. Note that the pictured example is an oversimplified visualization. Applying local blurring and interpolation to metric series with multiple dimensions and containing multiple samples is a non-trivial task.

6.3.1 Local Blurring

The local blurring should prevent the model from learning the general noise in the training examples by applying variations of that noise. Since noise can depend on the current state of a SuO component, the local blurring is done individually for each available training example. Applying noise variation must be done cautiously. Minor noise variations will not prevent the model from overfitting, while very high variations will hide the signal (i.e., anomaly symptom patterns), making it impossible for the model to identify it. Therefore, we infer information

about the noise from the respective metric series examples. This inference is made in two steps. First, an initial Box-Cox transformation [36] is applied to stabilize the variance of a metric series. This step is required since the second step assumes a stationary series, which is usually not the case for monitoring metric data from a dynamic IT system. In the second step, a season and trend decomposition based on Loess (STL) [197] is applied to infer the residuals of the metric series. We are interested in the local noise of the series and set the period parameter of STL to 3. The residuals represent the noise of the metric series and thus, can be modified to create a synthetic example that contains the original anomaly symptom patterns but with changed residuals. Changing residuals is realized by drawing samples from the extracted residual values. Since residuals are autocorrelated, a tapered block bootstrap [181] is applied to draw a sequence of residuals that preserves the distribution of the original residual values. The STL decomposition is reversed by combining the trend and seasonality component with the sampled residuals. After that, the inverse Box-Cox transformation is applied to acquire the original variance of the metric series.

Given a metric series $X_{t_0:t_c}$ that contains symptoms of an anomaly type, the local blurring is applied respectively for each series dimension $X^i_{t_0:t_c}$. Rectangles represent the synthetic examples resulting from the local blurring in Figure 6.5.

6.3.2 Interpolation

The combination of variating symptom patterns and the limited availability of examples poses a challenge for anomaly symptom recognition models. In Figure 6.5, this is visualized by gaps between available example points. These gaps result in decision areas representing anomaly types but are not covered by the available examples. Based on this, we propose to create synthetic metric series in these uncovered areas via interpolation between two neighboring examples. The interpolation between samples required a distance notion D. We define a metric series $X^{(1)}$ and its nearest neighbor series $X^{(2)}$ based on D, which are both of the same class, as the basis. An interpolated synthetic series example $X^{(1\text{-}2)}$ is defined via the distance to both basis series

$$D(X^{(1)},X^{(2)}) = D(X^{(1\text{-}2)},X^{(1)}) + D(X^{(1\text{-}2)},X^{(2)}). \tag{6.13}$$

Assuming a relative distance of 1 between $X^{(1)}$ and $X^{(2)}$, the distance between $X^{(1)}$ and $X^{(1\text{-}2)}$ is w_1 and between $X^{(2)}$ and $X^{(1\text{-}2)}$ is w_2. Setting w_1 and w_2 accordingly allows to control the location of the interpolated sample between the two basis samples, e.g., weights of $w_1 = 0.9$ and $w_2 = 0.1$ would result in a series that is located close to $X^{(1)}$ and further away from $X^{(2)}$. We utilize dynamic time warping (DTW) to calculate the distance D between two metric series.

Since DTW is an algorithm to find the closest distance of two series by searching for a path through the warping matrix, it is not trivially possible to interpolate between two series. The interpolation is realized by applying DTW barycenter averaging (DBA) [185]. It is an optimization problem defined as

$$\arg\min_{\overline{X}^i} \sum_{X^i \in \mathscr{X}^i} D^2(\overline{X}^i, X^i). \tag{6.14}$$

DBA is defined for univariate series and thus, must be applied on each metric series dimension i. The series $\overline{X}^i$ is the result and subject to optimization. All samples in $\overline{X}^i$ should be set such as to minimize the squared DTW distance between $\overline{X}^i$ and a set of reference series $\mathscr{X}^i$. The result is considered as an average over the series contained in $\mathscr{X}^i$.

The generation of an interpolated synthetic metric series example for a particular class is realized as follows. First, we select a random metric series example $X^{(1)}$ and its nearest neighbor $X^{(2)}$ based on the DTW distance. Second, we define a value n_1 for $X^{(1)}$ and $n_2 = n_{max} - n_1$ for $X^{(2)}$, where n_1 is a sample from a uniform distribution $n_1\ U(1, n_{max} - 1)$. Throughout our experiment, we set n_{max} to 10. Third, the local blurring is applied n_1 times to each dimension i of $X^{(1)}$ and n_2 times to each dimension i of $X^{(2)}$. The result of the local blurring for each dimension i is defined as $\mathscr{X}^i$. Fourth, DBA is applied on each $\mathscr{X}^i$, resulting in a synthetic metric series example $\overline{X}$. Thereby, the values n_1 and n_2 serve implicitly as weights $w_1 = \frac{n_1}{n_1+n_2}$ and $w_2 = \frac{n_2}{n_1+n_2}$. An n_1 value that is close to n_{max} results in a $\mathscr{X}^i$ that contains more locally blurred examples of $X^{i,(1)}$ which effectively leads to $\overline{X}^i$ to be closer to $X^{i,(1)}$ than to $X^{i,(2)}$. Note that the n_1 value is used for all dimensions i. This equality results in the same relative distance of all dimensions of $X^{(1)}$ and $X^{(2)}$. The synthetic metric series $\overline{X}$ is the result of our metric series augmentation method. The augmented examples are used to train the RNN GRU model and configure the parameters of the anomaly type inference. Triangles represent the synthetic examples resulting from the interpolation in Figure 6.5.

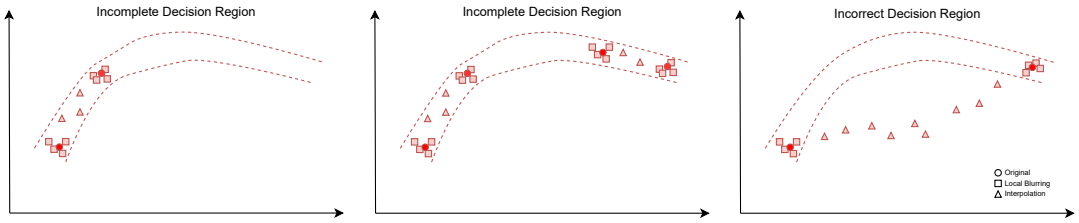


Figure 6.6: Conceptual visualization of two possible problems with sporadic decision region sub-sampling.

Due to the interpolation, this augmentation method requires examples representing a reasonable sub-sampling of the decision region in the input space to yield good augmentations. Two problems that occur when the input space is sporadically sub-sampled are conceptually depicted in Figure 6.6. It shows a decision region of a class between two dotted lines, which is non-linear. As depicted in the left-most visualizations, only two examples for the class are available, allowing only a sub-region to be sub-sampled. The augmentation method is not able to extrapolate distant decision regions. In the middle visualization, four examples are available, which are located in the outer area of the decision region. The nearest neighbor selection prevents the sub-sampling of the region between the two outer examples. In the right-most visualization, the examples are located distant from each other. The non-linearity of the decision region's trajectory leads to a sub-sampling of a decision region representing the shortest path between the two examples but is different from the actual decision region.

The central part of the local blurring is the STL decomposition, and the interpolation of metric series is achieved via DBA. Therefore, we refer to this augmentation method as STL-DBA augmentation.

6.4 Evaluation

In the following, we evaluate the anomaly symptom recognition method whereby we employ the RNN model with GRU cells to realize the symptom analysis, apply the described anomaly

type inference, and use the metric series augmentation. First, a detailed explanation of the conducted experiment is given. It is executed on the previously presented evaluation environment, whereby we focus on system components that execute heterogeneous workloads and inject different anomaly types therein. Base on the collected metric data, we analyze the effect of our metric data augmentation method when the RNN model is trained.

6.4.1 Experiment

We want to evaluate the proposed method's ability to identify anomaly types within components that concurrently execute heterogeneous workloads. Therefore, anomalies are injected on compute nodes that host VMs wherein different services are executed. We use the experiment environment described in Section 5.3 to conduct the experiment. The experiment is conducted on 27 commodity cluster nodes. OpenStack Stein[2] is deployed on 21 nodes. Three nodes are load balanced with haproxy [3] and host the OpenStack network and controller services, six are used as storage, and eleven as compute nodes. Out of the remaining seven nodes, one is reserved for the anomaly injection controller, and the other six for load generation.

Table 6.1: Number of VMs for each Clearwater and video on demand service.

	Clearwater						
Service	bono	sprout	homer	homestaed	chronos	cassandra	astaire
# VMs	6	6	4	4	5	3	5
	Video on Demand						
Service	lb	video server					
# VMs	6	6					

We jointly deploy both services, Clearwater, and video on demand. The number of VMs for each Clearwater and video on demand service is listed in Table 6.1, whereby VMs were randomly placed on the available compute nodes. Such deployment represents an execution of multiple heterogeneous workloads on each compute node. The Clearwater VMs are provided with one virtual CPU, 2GB RAM, and 40GB disk space, while video on demand VMs run with two virtual CPUs, 4GB RAM, and 80GB disk space. All VMs operate with Ubuntu 16.04.3 LTS under Linux kernel version 4.4.0-128-generic. We execute a SIPp process respectively on two and four video client simulation processes respectively on the other four cluster nodes separated for workload execution. Each SIPp process is configured to generate between 600 and 900 user registrations and call initiations every second. The video client processes are simulating 40 to 60 users that request videos. After finishing a video stream, a simulated user waits for 10 seconds to 5 minutes to select the next video. The video and its resolution were randomly selected. The monitoring agents are configured to collect system metric data at a frequency of 2Hz. All metrics that were collected throughout the experiment are listed in Table 6.2. The metrics are measured either in bytes, percent %, or a counting value #.

To evaluate the anomaly symptom recognition method, we inject anomalies into the compute nodes. The anomalies that were injected during this experiment are listed in Table 6.3. Each anomaly type was injected 20 times into a randomly selected compute node, resulting in

[2] `https://releases.openstack.org/stein` (last access 06 May 2021)
[3] `http://www.haproxy.org/` (last access 30 May 2021)

Table 6.2: Metrics that are collected during the experiment.

Resource	Metrics	Unit
CPU	cpu utilization	%
Memory	allocated memory	%
Disk	written data	bytes
	read data	bytes
	I/O operations	#
	used disk volume	%
Network	rx volume	bytes
	tx volume	bytes
	rx packets	#
	tx packets	#
	errors	#

180 injections. The duration of every injection was set to four minutes, after which one minute recovery period without injections was defined. We use the start and end times of the anomaly injection. All metric series samples within this time range are labeled with the injected anomaly type. Additionally, we consider 25 normal samples before the respectively first anomalous sample to capture the transition from normal to an anomaly state. The experiment has an initial 2 hour period without injections, resulting in an overall experiment duration of 17 hours.

Table 6.3: Description and parametrization of anomalies.

Anomaly	Compute Nodes
CPU overutilization	Utilize all cores at 90 - 100%.
CPU hogging	Step-wise increase the CPU utilization. Increase by 1% every 2 seconds until 90-100%.
Memory leak	Step-wise allocate memory. 20 MB every second until 3000 MB.
Memory overutilization	Allocate 2500 MB of memory.
Disk IO	Start 3 processes that write / read data to / from disk.
Packet Loss	Drop 5 % of incoming and outgoing packets on all interfaces.
Throttled Bandwidth	Limit bandwidth to 1500 Kbps.
Network Overutilization	Repeatedly start downloading a large file from the internet.

6.4.2 Effect of Metric Series Augmentation

The anomaly symptom recognition method should be evaluated as depicted in Figure 6.1. It includes the metric series augmentation used to generate augmented training examples based on a set of available training data. The augmentation should prevent the GRU model from overfitting. Furthermore, augmented examples are used to set the parameters of the anomaly type inference. Before analyzing the anomaly symptom recognition method and comparing it

to baselines, we investigate the effect of the augmentation on the training process. The accuracy of the proposed anomaly symptom recognition method that was trained based on either augmented or original examples is tested. Beside the proposed STL-DBA augmentation from Section 6.3 we utilize a jitter augmentation as proposed by [229] or [226]. Thereby, a small random value is added to each sample of the time series. The random value is drawn from a normal distribution with zero mean and a configurable standard deviation. We follow the recommendation of Um et al. [229] and set the standard deviation $\sigma = 0.03$. The Jitter augmentation is applied respectively to each metric series.

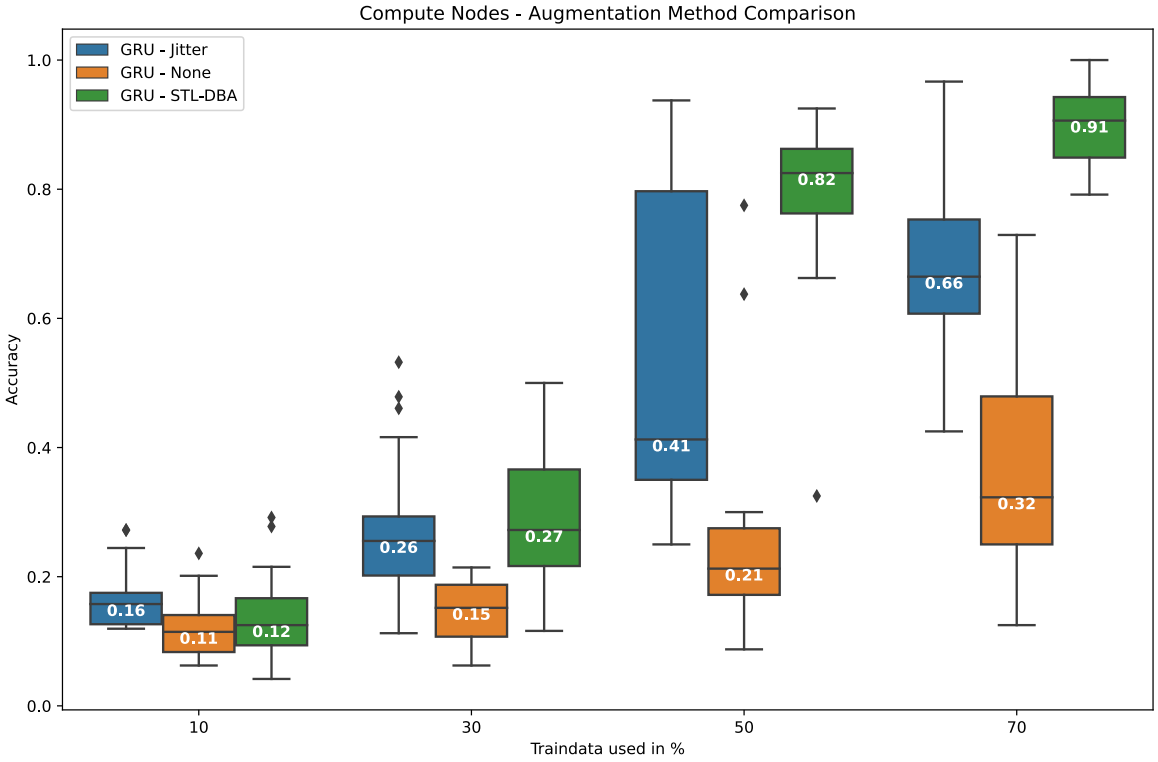


Figure 6.7: Anomaly symptom recognition results based on different metric series augmentation methods.

The available 20 examples of each anomaly type injected into the compute nodes are randomly split in a train and test set. The split is done respectively for each anomaly, i.e., the same amounts of training and test examples are available for each anomaly after the split is applied. After that, the training data are either used directly or are the basis to generate augmented examples. We train the GRU model and set the anomaly type inference parameters. For the setting of the anomaly type inference parameters, we utilize 20 augmented examples. When executing the evaluation without augmentation, all available training examples are used to set the parameters. The evaluation is executed for four different training, and test data splits, whereby 10 %, 30 %, 50 %, and 70 % of the available examples are used for training while the remaining are used as test data. For each of the split, random examples are selected and used for training. The evaluation for each split is repeated 20 times, each time with a random training set selection.

The evaluation results for the different metric series augmentation methods are depicted in Figure 6.7. The plot shows the accuracy values for the anomaly symptom recognition, whereby

the GRU model and the anomaly type inference parameters are adjusted based on metric series examples. The three colors represent the two different augmentation methods and the case where no augmentation was used. The x-axis depicts the different training, and test data splits. One notable aspect observed during the experiments is the high variation between the folds, whereby the model yields sporadically high accuracy results. A bar plot is used to visualize this. The line within the bar and the value represent the median, while the lower and upper boundaries of the bars are the first and third quartile. The distance between them is the interquartile range (IRQ) and is used to define outliers. All observations outside the $1.5 * IRQ$ range are outliers and marked separately as points in the plot. The lower and upper whiskers represent the minimum and maximum value that are not outliers.

The combination of the overfitting and the heterogeneous workloads resulting in high intra-type variation for the anomaly metric series prevents the model from detecting anomaly type-specific symptoms based on 10 % and 30 % of available training data. These splits translate to two and six metric series examples, which is not enough to match anomaly types reliably. However, it can be seen that both augmentation methods are mitigating the overfitting of the GRU model. For the 30 % split, the STL-GRU augmentation results in a median accuracy of 0.27 compared to 0.15 when no augmentation is used and 0.26 for Jitter augmentation. Three outliers of the Jitter augmentation reach values above 0.4, which is not the case when no augmentation is used. The 50 % split, i.e., having ten examples available for each anomaly type, is a reasonable representation of the anomaly type decision regions and allows the STL-DBA augmentation to yield a median accuracy of 0.82. The lower outliers indicate that there are splits, for which STL-DBA does not sub-sample the decision region well enough. The median accuracy for the Jitter augmentation is 0.41 and for no augmentation 0.21. However, a very high variance can be observed, which indicates a high dependence on the training examples selected for the respective fold. For Jitter augmentation, some constellations yield accuracy results of above 0.8 while more than half of the accuracy values lie below 0.4. Using 70 % of the available data for training, the median accuracy value for the STL-DBA augmentation improves to 0.91 and does not yield any lower bound outliers. The median values for the Jitter augmentation and when no augmentation is used improve to 0.66 and 0.32. However, the high dependence on the selected training examples remains, which results in a wide IQR when no augmentation is used and in long whiskers for Jitter augmentation.

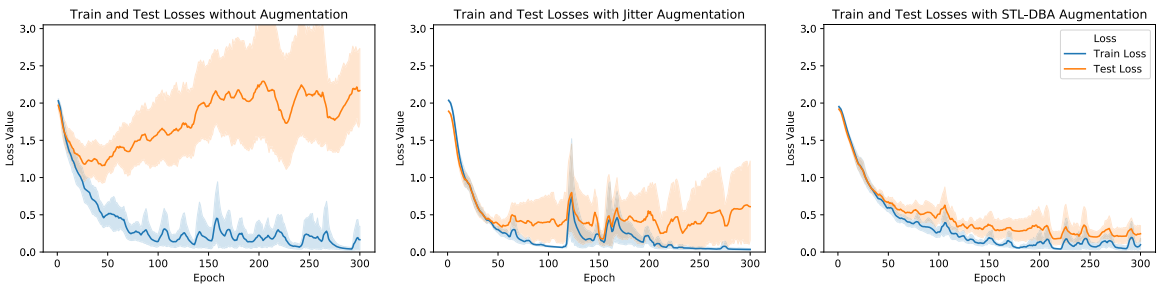


Figure 6.8: Loss values for training and test examples.

To analyze the extent to which the observed results are due to the overfitting of the GRU model, we analyze the training procedure of the model. Therefore the loss values for all 20 folds of the 70 % are observed across all 300 epochs. During training, the average loss value for each epoch is determined. After each epoch, we switch the model into the prediction mode, calculate

a prediction for each test example metric series, and determine the average loss value. This procedure results in a series of training and test loss values for each epoch. To remove initially high loss, we start from epoch 3. The results are depicted in Figure 6.8, which shows the loss values when no augmentation is used (left plot), for the Jitter augmentation (middle plot), and the STL-DBA augmentation (right plot). The x-axis represents the epochs, while the y-axis shows the loss values. The thick line represents the mean value, while the transparent area around it is the standard deviation. When no augmentation is utilized, it can be seen that after an initial joint decrease, the loss values for the training and test examples are strongly diverging. The model is learning to apply the anomaly symptom recognition on the training examples but fails to do so on unseen metric series, which fits the definition of overfitting. The test data loss values are closer to training data loss values for the Jitter augmentation, indicating a mitigated overfitting effect. However, between epoch 50 and 100, the test loss is not converging with the training loss. The application of the STL-DBA metric series augmentation results in a test data loss that closely follows the training data loss, resulting in a well-generalized model. Combining the accuracy observations and loss value progresses, we conclude that the proposed STL-DBA metric series augmentation method can mitigate the overfitting problem and enable a well-generalized GRU model if the available examples are sub-sampling the decision region well enough.

6.4.3 Evaluating Anomaly Symptom Recognition

We evaluate the ability of the anomaly symptom recognition to determine the anomaly type based on metric series when the heterogeneous workload is executed on the components. Based on the augmentation evaluation from Subsection 6.4.2, we utilize the STL-DBA augmentation to train the symptom analysis model and set the anomaly type inference parameters. We refer to our model as GRU in the following. The evaluation results are compared to the GP, DTW, and Euclidean distance models. Since these models have limitations regarding the input space dimensionality d, we use a partly aggregated subset of the collected metric data, which are: CPU utilization in percent, the fraction of allocated memory in percent, the sum of disk read and write data volume in bytes, and the sum of rx and tx network traffic volume in bytes. The same parametrization as in Subsection 5.4.3 is used for the GP, DTW, and Euclidean distance models. The augmentation is applied only for the GRU RNN model training. All other models utilize the available training examples. Again, we use the available 20 examples of each anomaly type injected into the compute nodes. The 20 examples of each anomaly are randomly split into a training and test dataset. We analyze our method for different amounts of available training examples, where 10 %, 30 %, 50 %, and 70 % are used as training data, and the remaining examples are left for testing. The evaluation is executed 20 times for each split.

The accuracy results are depicted in Figure 6.9. The plot visualizes the trajectories of achieved accuracy values with increasing training data split. Accuracy values are shown on the y-axis, while the training data splits are marked on the x-axis. Respective colors represent the four models. The dots are the mean accuracy values, while the error bars represent the 0.95 confidence interval calculated for all anomalies and all folds. It can be observed that the GRU model generally yields lower accuracy results for the 10 % split. For GP, DTW, and Euclidean distance models, the accuracy ranges between 0.18 and 0.21 while the accuracy for GRU is 0.14. For the 30 % split, the highest accuracy of 0.45 is achieved by the DG model while the GRU, DTW, and Euclidean distance model's accuracies are 0.29, 0.33, and 0.28. For the

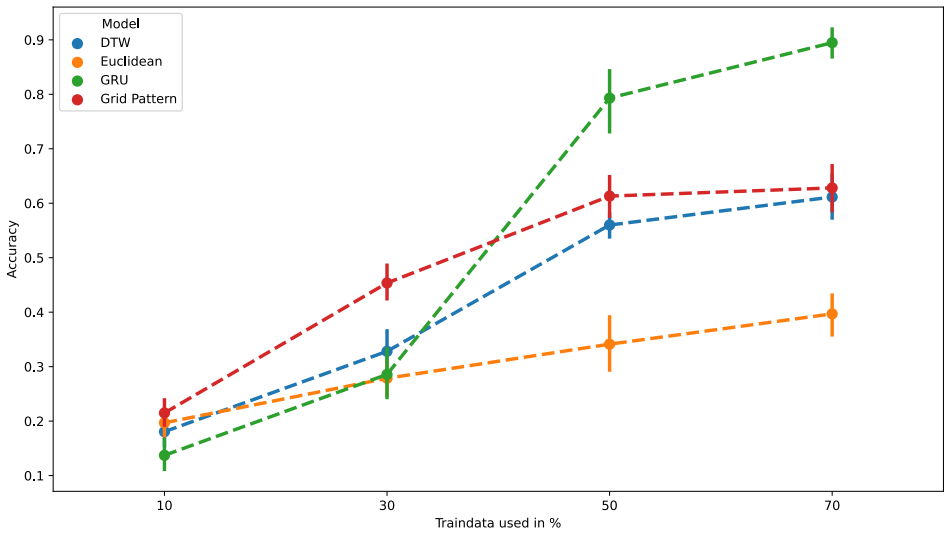


Figure 6.9: Accuracy results for different train / test data splits.

50 % split, the GRU accuracy jumps to 0.79. A similar increase was observed during the previous evaluation in Subsection 6.4.2. This increase indicates that the intra-type variation of the anomalies injected in this experiment necessitates around ten training examples to reasonably sub-sample the decision regions in the input space, which allows the STL-DBA augmentation to produce representative interpolations. The accuracy values for the GP, DTW, and Euclidean distance models respectively increase to 0.61, 0.56, and 0.35. A slight increase of the GP and DTW model's accuracies for the 70 % split to 0.63 and 0.61 indicates a saturation. Additional training data are unlikely to further increase the accuracy values for these models. The intra-class variation limits the ability of the saturated models to improve further their ability to match anomaly types, while the GRU model in combination with the metric data augmentation can improve further. The slow increase of the Euclidean distance model prevents it from reaching a mean accuracy of above 0.4. The GRU model achieves the highest accuracy value of 0.9.

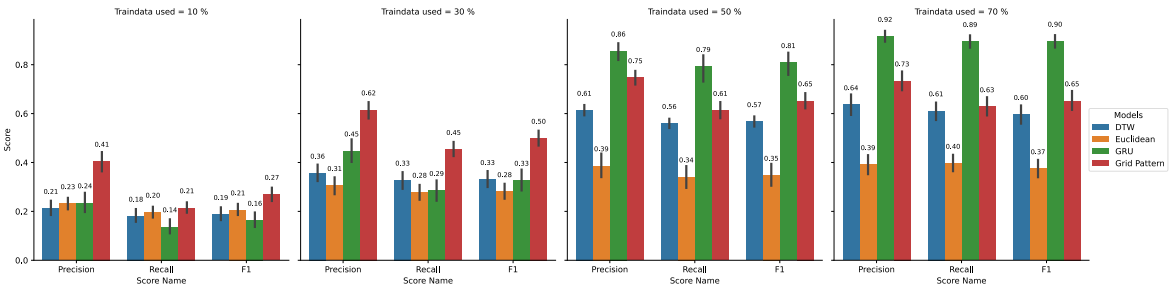


Figure 6.10: Precision, recall, and F1 scores.

We further analyze the precision, recall, and F1 scores, which are depicted in Figure 6.10. From left to right, the four plots represent the 10 %, 30 %, 50 %, and 70 % splits. Each bar color represents a model. The respective score type groups the bars. The height of the bar is defined by the respective mean score value, while the error bars are 0.95 confidence intervals. As expected, the scores for all models increase with the increasing availability of

training examples. For the 10 % and 30 % splits, a comparably high precision value can be seen for the GP model. The GRU model yields a comparable precision to the DTW and Euclidean distance models, while the recall and F1 score are notably lower. For the 30 % split, the GRU model achieves a higher precision compared to the DTW and Euclidean distance models and a comparable recall and F1 score. At the 50 % split, the GRU scores outperform all other models reaching a precision, recall, and F1 score of 0.86, 0.79, and 0.81 compared to the GP model with 0.75, 0.61, and 0.65. The same steep increase of the GRU scores between 30 % and 50 % split was previously observed for the accuracy values. For the 70 % split, the GP, DTW, and Euclidean distance models reach a saturation, whereby only a slight change in the scores can be observed compared to the 50 % split. The scores for the GRU model further increase to a precision of 0.92, a recall of 0.89, and an F1 of 0.90. The overfitting of the GRU model when only a few training examples are available leads to low evaluation scores. Additional training examples enable a better sub-sampling of the decision regions of each class and thus allow the augmentation method to generate examples within the decision regions. The variation of the examples through augmentation mitigates the GRU overfitting problem for 50 % and 70% splits.

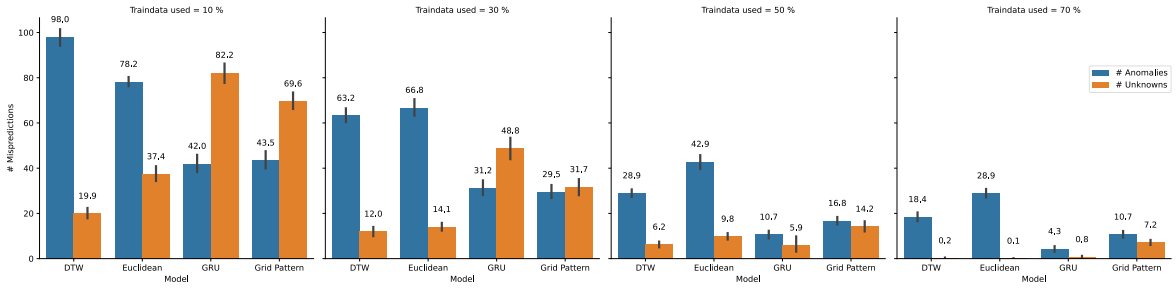


Figure 6.11: Distribution of incorrectly predicted classes.

A model should rather misclassify anomaly types as unknown instead of predicting an incorrect anomaly type. From a practical perspective, the models' ability to admit their inability to match an anomaly type is essential. It is better to redirect the problem to a human expert instead of automatically select operations that will not resolve the anomaly. Therefore, we observe the distribution of misclassifications between the "unknown" class and all other classes and present the result in Figure 6.11. The four plots represent the four different splits. The two bars are grouped by the model while the colors show the average total number of incorrectly predicted examples respectively for "unknown" and all other classes mispredictions. The error bars represent the 0.95 confidence interval. For the 10 % and 30 % splits, most GRU and GP model mispredictions are the "unknown" class. In contrast to that, the DTW and Euclidean distance models mispredict most examples as other anomaly type classes. The ration between "unknown" class and all other anomaly type classes changes after the 30 % split for GP and GRU. This change indicates an adjustment of the decision regions, which reduces the "unknown" misclassifications faster. For the 70 % split, all models except GP yield an average total number of other anomaly type mispredictions lower than 1.0. The "unknown" misprediction value of 7.2 for the GP model indicates that the scores could be further increased by fine-tuning the β parameter for each class. The GRU model yields an average total number of 5.1 mispredictions.

In Subsection 5.4.3 we state that the high precision values of the GP model can be translated

to a majority of "unknown" mispredictions. However, the inverse is not possible, represented by most "unknown" mispredictions for the GRU model but a precision value of 0.24. The precision is the ratio between the correct predictions of a class divided by the number of other classes' mispredictions as this class. This entails that the precision value for a class can be low if either few examples are correctly predicted, or many examples of other classes are incorrectly predicted as this class. Due to the nullification of the "unknown" class, the latter results in high precision values for GP. In the case of GRU, the small number of correct predictions for certain anomaly types (e.g., packet loss yields and memory leak yield precisions of 0.03 and 0.1 for the 10 % split) results in a lower overall precision value.

6.4.4 Evaluating Unknown Anomaly Type Recognition

We test the ability of the proposed model to recognize previously unseen anomaly types as "unknown", which is important for the extension of the anomaly symptom recognition knowledge base. The same partly aggregated subset of the collected metric data as described in Subsection 6.4.3 is used, whereof the monitoring metric series that are labeled as anomalous is selected. The available anomaly type examples are split into a training and test dataset. We exclude anomaly types, one at a time, from the training set for this evaluation and move them to the test set. The task of the models is to predict the metric series that represent the excluded anomaly type as "unknown". Each anomaly type is excluded ten times from the training data, while 10 %, 30 %, 50 %, and 70 % of randomly selected examples of the non-excluded anomaly types are selected for the training dataset.

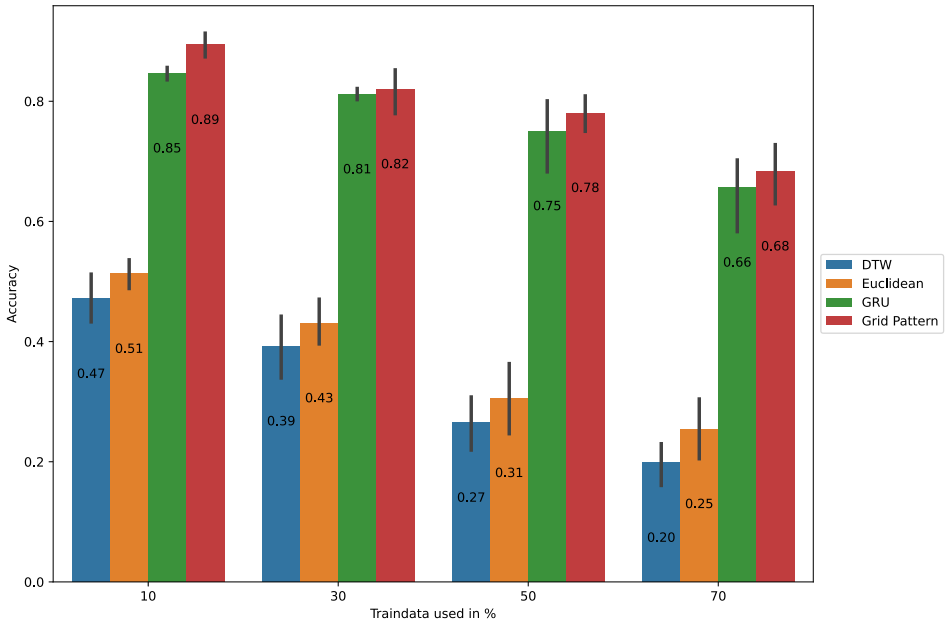


Figure 6.12: Accuracy result for predicting the "unknown" class.

We analyze the model's ability to recognize the excluded anomalies by calculating the accuracy. For this, all 20 metric series examples of the excluded anomaly type are presented to the model with the desired outcome for it to classify them as "unknown". The results are depicted in Figure 6.12. Each of the four bars represents the accuracy value of the respective model,

while the bars are grouped by the split percentage of used training data. The height of the bar shows the mean accuracy, while the error whiskers are the 0.95 confidence intervals. Classifying previously unseen anomaly types as "unknown" requires setting tight decision boundaries around the decision regions. It can be seen that it is challenging for models to set tight decision boundaries in the presence of homogeneous workload and the resulting high intra-class variations. The DTW and Euclidean distance models are yielding lower results than the GP and the GRU models. The GP model yields constantly higher accuracy than the GRU model. When few training examples are used, the sparsity of the decision regions allows the GRU and GP to achieve accuracy values of 0.85 and 0.89 for the 10 % split and accuracy values of 0.81 and 0.82 for the 30 % split. For the 50 % and 70 % split, the accuracy for the GRU model decreases to 0.75 and 0.66. The high variance indicates a dependence on the selected training examples. The GP model's accuracy is 0.78 and 0.68 for the 50 % and 70 % splits with a smaller variance than the GRU model.

The augmentation that is used to train the GRU model results in a sub-sampling of the decision region. Whether the generated samples fall into the real decision region has an effect on how the decision boundaries are set during model training. A higher number of available training examples results in decision regions of known classes that overlap with the unknown class decision region with the effect of misclassifying them. It can be seen as a trade-off. More available examples result in higher accuracy for known anomaly type classes but entail lower accuracy values when recognizing unknown anomaly types.

Chapter 7

Integration into an AIOps Platform

This chapter is concerned with a prototypical implementation of a AIOps system that contains our presented ReCoMe engine and the anomaly type recognition methods. It should enable the automatic selection of operations to resolve anomalies. In the previous sections, we show the ability of our methods to recognize symptom patterns in metric data and infer the anomaly type. However, an anomaly symptom recognition alone cannot enable an automatic selection of operation to resolve anomalies. It requires other methods like system component monitoring, anomaly detection, root cause analysis, and execution of operations. These are provided by a AIOps platform named ZerOps (short for zero-touch operation), wherein we integrate an implementation of the ReCoMe engine, which contains our anomaly symptom recognition methods. ZerOps builds upon Bitflow, a data stream processing framework. We first describe both systems, which are published in [105] and [106]. Besides the ability to recognize symptoms and infer anomaly types, the ReCoMe engine requires two other modules. The ingest control serves as an interface. It manages the ingest of metric data and anomaly detection alarms, delegates the root cause analysis task and initiates the anomaly symptom recognition. Further, it can relay the metric data to another ReCoMe engine if no operation can be selected for the inferred anomaly type to resolve the anomaly. It also contains the operation selection, including the interaction with human operators. Finally, ZerOps allows flexible deployment of the AIOps modules. We describe the concept of *in-situ* deployment and discuss different deployment concepts of our ReCoMe engine. For evaluation, we deploy ZerOps in a cloud and simulated fog computing environment and test the ability of our solution to infer anomaly types.

7.1 Bitflow and ZerOps

We implement the ReCoMe engine into the ZerOps platform. Initially proposed as a self-stabilization pipeline for network function virtualization infrastructures [109], we develop ZerOps into an AIOps platform for cloud and fog computing environments. At its core, it is based on a data stream processing framework called Bitflow [1]. The architecture of Bitflow is depicted in Figure 7.1. At its core, it consists of a stream processing runtime and a Bitflowscript compiler. The runtime provides data processing operations which are referred to as *operators*. The runtime enables the execution of operators, organizes the routing of the data, and provides input and output interfaces. Furthermore, it manages all libraries that are required to execute the

[1] `https://github.com/bitflow-stream/` (last access 20 May 2021)

operators. An operator itself processes incoming data and forwards them to the next operator. They can be combined as a directed acyclic graph (DAG). The definition and parametrization of operators, input sources, and output sinks together with their combination to DAGs is expressed in a lightweight domain-specific language (DSL) named Bitflowscript. Such a script can be passed to the runtime as an argument, whereupon data are read or received from defined sources, processed by the operator of the DAG, and transmitted or provides at an output interface. Thereby, the Bitflowscript compiler checks the syntax, identifies potential conflicts, and notifies the user if defined step implementations are missing. A set of data operations, such as filtering, normalization, and aggregation, is implemented by default. More advanced operators, such as machine learning algorithms can be added through a plugin mechanism.

The Bitflowscript compiler and the runtime are deployed as Docker [2] containers. We refer to a running container as a processing step. It is possible to define processing steps as input data sources or output sinks. By doing so, the steps can be combined into a data processing pipeline. Aside from processing steps, it is possible to define files, databases, and network endpoints as sources and sinks.

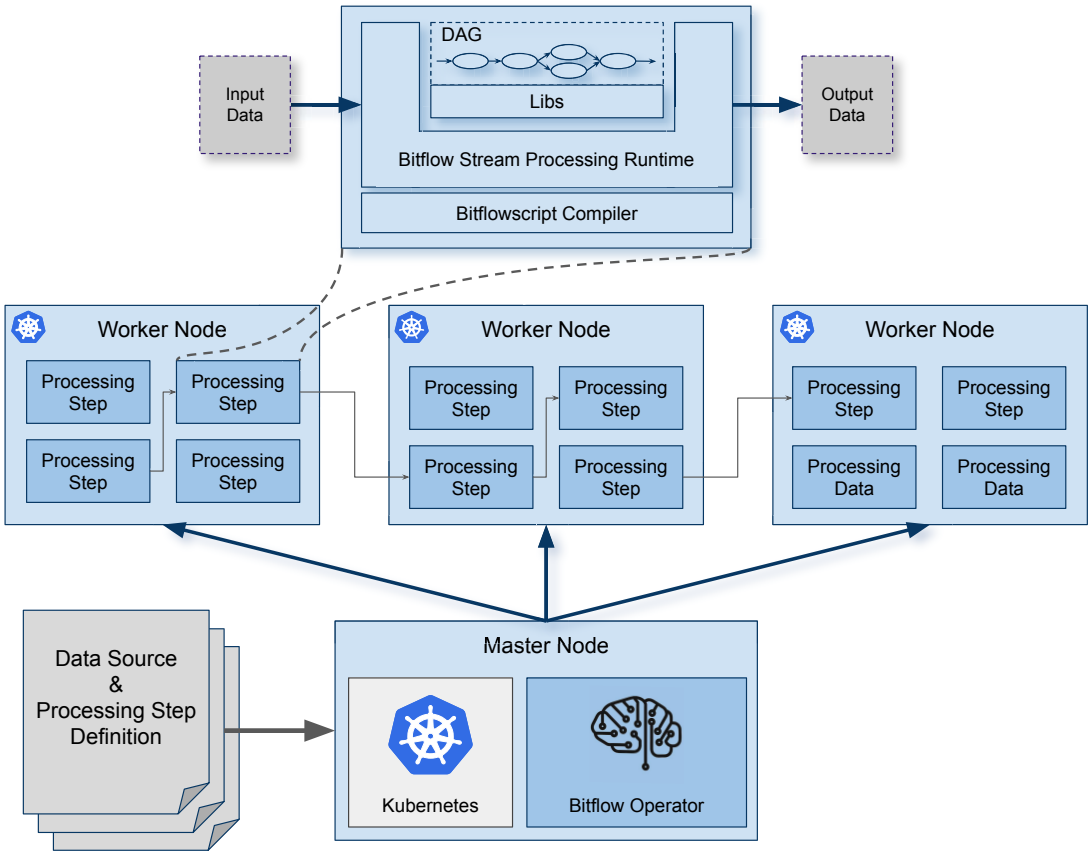


Figure 7.1: Bitflow data stream processing framework (adjusted from [106]).

The management of the processing steps is realized with Kubernetes [3]. Kubernetes is used for resource management, scheduling, and placement of processing steps. A Bitflow operator

[2] `https://www.docker.com/` (last access 20 May 2021)
[3] `https://kubernetes.io/` (last access 20 May 2021)

is realized with the Kubernetes operator pattern [4]. Thereby, the processing steps are managed as custom resources [5]. Two resource types are defined: data input and processing step. The coupling of data inputs and processing steps is realized via labels. It is possible to assign labels to data inputs and processing steps and define matching rules based on these labels. The Bitflow operator spawns the instances of the custom resources based on the defined labels and matching rules and couples them as a data processing pipeline. Following the declarative object definition principle of Kubernetes, processing steps are defined via a customized Kubernetes object description files (ODF) - usually in YAML or JSON format. They contain the definition of processing steps and data sources, which are external sources like files, databases or monitoring systems, or existing processing steps. The Kubernetes ODF also contains the default configuration parameters for containers, which allow to define resource and placement constraints, adjust port and volume mappings, and many others. These configurations allow controlling how processing steps are places and executed on worker nodes. It can be seen that such a setup allows the definition of AIOps methods like anomaly detection, anomaly symptom recognition, root cause analysis, or operation execution as processing steps. The connection of processing steps enables the deployment of a closed-loop AIOps system, whereby it is possible to control how many resources the steps are allowed to consume and how they should be placed across the worker nodes.

The ZerOps platform is implemented around Biflow. It includes several components to realize the AIOps system. The system architecture is depicted in Figure 7.2. ZerOps utilizes the Bitflow collector, which implements interfaces to different system data sources to acquire monitoring data from SuO components. They transform the monitoring data into a unified Bitflow-internal format and provide them as a data source to data processing steps. Furthermore, ZerOps provides an interface to the meta-information of the SuO components, such as the placement of virtual machines, IP addresses, or network topology. This information is required by the Bitflow operator for data processing step placement or by the operation selection module to select operations that can be executed on a specific component. ZerOps extends the Bitflow stream processing runtime and implements interfaces to a model repository and an event bus. Former is used to load and store models, e.g., for anomaly detection or anomaly symptom recognition. A version policy is implemented for models, which allows retrieving previous model parameters. It is used to trace the models' past decisions if parameters changes in the meantime. Latter is a communication interface between processing steps to transmit and receive notifications.

The Bitflow operator places and starts the data processing steps on worker nodes. The anomaly detection, RCA, and all ReCoMe modules are implemented as data processing steps. They receive monitoring series data and apply their analyses. The operation selection is a special processing step. It does not process the monitoring series but operates on events. If anomalies are detected, it expects to receive information about the anomaly type, which allows an automatic selection of an operation. StackStorm [6] is utilized as an external operation executor to separate the selection of operations from their implementation. Human operators are needed whenever an automatic operation selection cannot be made. They are required to de-

[4] `https://kubernetes.io/docs/concepts/extend-kubernetes/operator/` (last access 20 May 2021)

[5] `https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/` (last access 20 May 2021)

[6] `https://stackstorm.com/` (last access 20 May 2021)

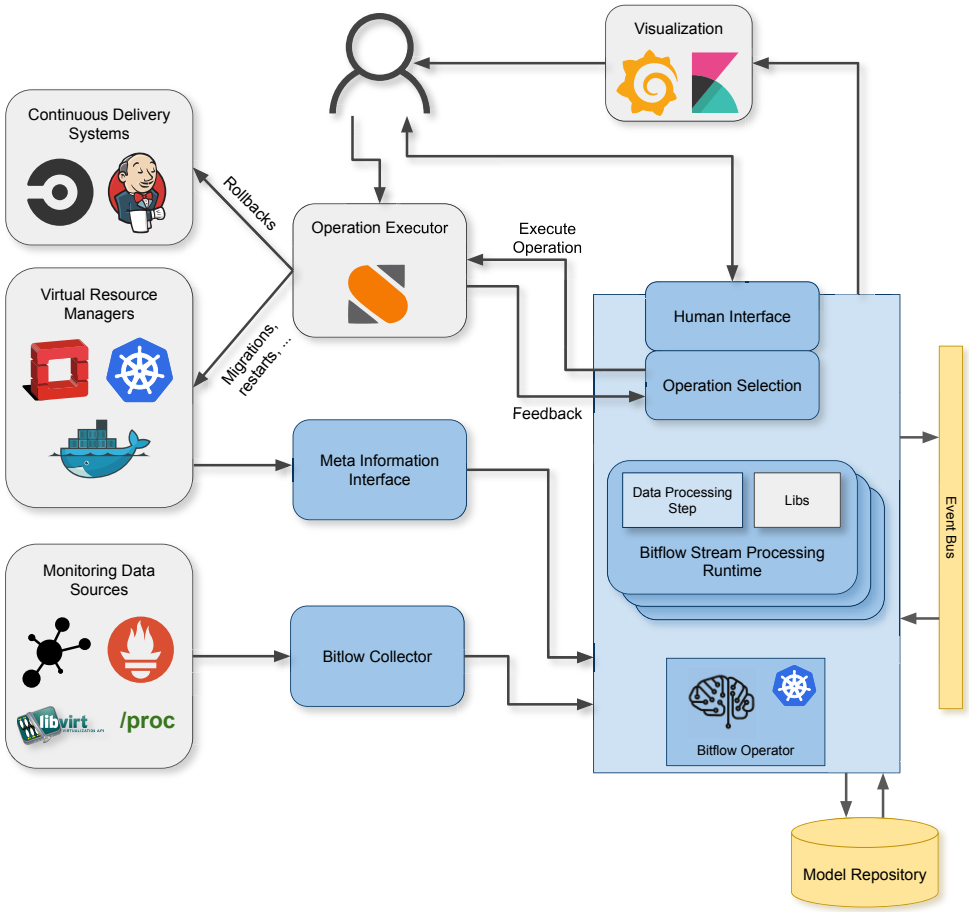


Figure 7.2: Overview of the ZerOps platform.

fine operations and adding them into the ReCoMe operation knowledge base. For this, ZerOps provides a human interface where these entries can be done. The ZerOps provides interfaces to visualize metric data or analysis results via Kibana and Grafana. It provides an overview of the states of anomalous SuO components and analysis method results to human operators.

7.2 ReCoMe Engine Implementation Details

The interaction between the modules of the ReCoMe engine is a complex and dynamic process with interfaces to external entities. The engine receives alarms from external anomaly detection, queries metric data of components reported to be anomalous, identifies anomaly type-specific symptom patterns, interacts with human experts, and selects feasible operations to resolve the anomalies. This section provides details that are important for implementing the ReCoMe engine and its integration into ZerOps.

The ingest control module is the interface between the anomaly detection and the ReCoMe engine. It receives $N \mapsto A$ and $A \mapsto N$ alarm messages via the event bus of ZerOps. Furthermore, it communicates to an external RCA module whenever several SuO components are reported to be anomalous to identify the root cause. Whenever several SuO components are reported to be anomalous, an external RCA module is requested to identify the root cause component. Based

on this, it selects one anomalous SuO component to be resolved by the subsequent ReCoMe modules. First, the RC component is remediated. It requests metric data for this component from the Bitflow collector and keeps receiving metric data samples whenever available until an $A \mapsto N$ alarm occurs. These data are forwarded to the anomaly symptom recognition, which should infer the anomaly type based on the observed symptom patterns. If other components are still anomalous after the remediation of the RC, it initiated the symptom analysis for them as well. The ingest control can be configured to delegate the anomaly symptom recognition to another ReCoMe engine. In this case, it buffers the received metric series samples and receives feedback messages from the operation selection module. If the "unknown" anomaly type is inferred or if an automatically selected operation could not resolve the anomaly, the ingest control forwards the buffered metric data to another ReCoMe engine. There, the anomaly symptom recognition is initiated.

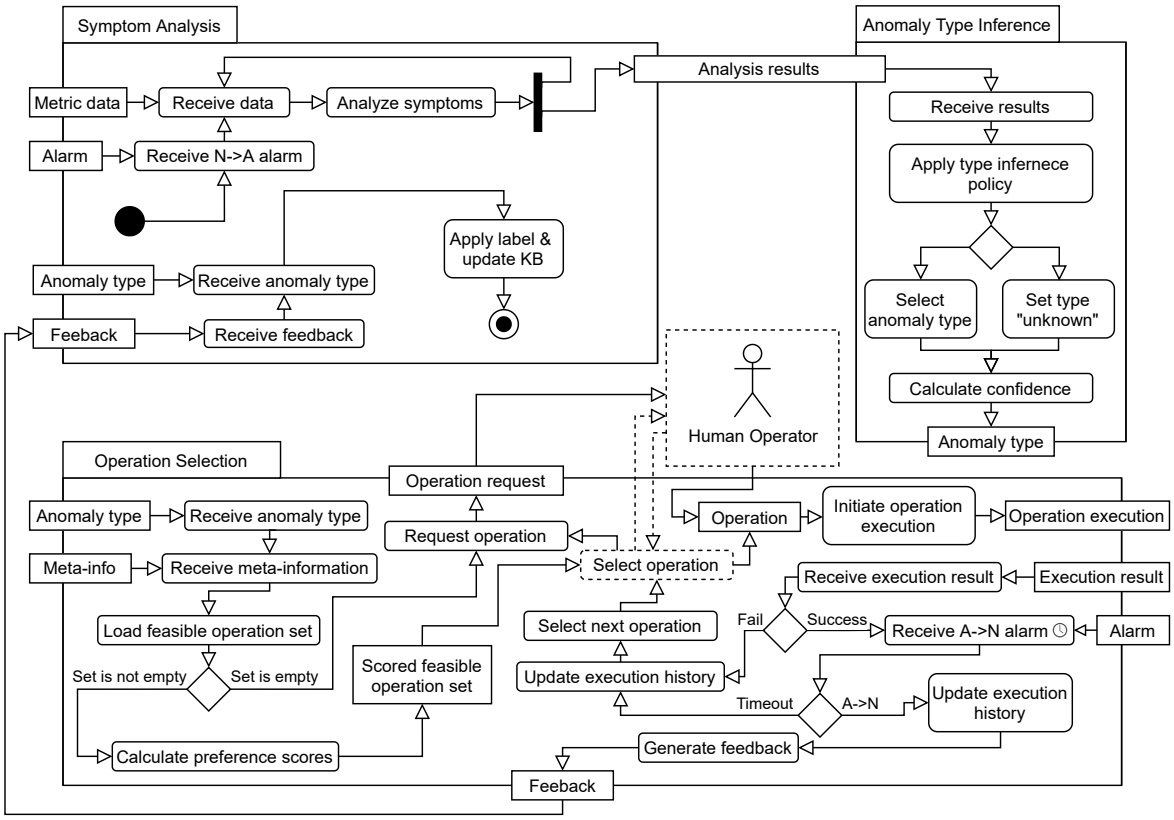


Figure 7.3: ReCoMe process flow.

The internals of the symptom analysis, anomaly type inference, and operation selection modules are explained based on the process flow diagram in Figure 7.3. The symptom analysis receives the metric data series of an anomalous component and tries to recognize anomaly type-specific symptom patterns. This analysis is applied whenever a metric series sample is received. The symptom analysis results represent associations of anomaly-type symptoms within the observed metric series with the symptom patterns from previously observed anomalies. These symptom analysis results are sent via the event bus to the anomaly type inference module, which applies a type inference policy. This inference policy application either results in predicting an anomaly type from the result list or setting the anomaly type as "unknown". After that, a

confidence value is assigned to the selected anomaly type. The anomaly type and the confidence value are sent via the event bus to the symptom analysis and operations selection modules.

Based on the anomaly type and meta-information about the anomalous component, the operation selection module should automatically select feasible operations to resolve the ongoing anomaly. The anomaly type and meta-information are used to query the knowledge base of available operations. First, operation knowledge base entries that contain the received anomaly type in their anomaly type column are selected. After that, only entries for which the component meta-information match the defined properties in the component selector column. The outcome is a set of feasible operations. This set can be an empty set if no matches are found. A human operator is involved in case an empty set is encountered, who must define an operation to resolve the ongoing anomaly. Otherwise, each operation's execution history entries are loaded, and a preference score is calculated for each operation. It results in a set of feasible operations descendingly sorted by their preference score. Details about the preference score calculation are described in Subsection 7.2.1 After that, an operation should be selected from the sorted list. Selecting a concrete operation from this set highly depends on the SuO use case and the preferences of human experts. It is possible to propose the list as a recommendation and leave the selection to a human expert or select the operation with the highest preference score.

After selecting an operation, the operation selection module initiates the execution through the external operation executor. The outcome of the execution is expected to be reported. This outcome indicates whether the operation was successfully executed, i.e., no errors occurred when executing the respective commands of the operation. If the remediation operation execution fails, the outcome is added to the remediation execution knowledge base, and another operation is requested. This can involve a human expert. Note that the reported execution result does not contain any statement whether the anomaly was successfully resolved. If the operation execution was successful, an $A \mapsto N$ alarm is awaited. This process is coupled with a timeout. If the executed operation cannot resolve the anomaly, an $A \mapsto N$ alarm is never received, and the timeout occurs. This again entails storing the results in the operation execution knowledge base, whereafter another operation is selected. If no timeout occurs, the receipt of an $A \mapsto N$ alarm indicates the transition of the SuO component from an anomalous to a normal state. In this case, the executed operation is assumed to successfully resolved the anomaly. The execution result is stored in the operation execution knowledge base, and respective feedback for the symptom analysis module is generated.

The feedback message should enhance the symptom analysis for future occurrences of this anomaly type. The content of the feedback message can be separated into three cases:

1. The predicted anomaly type is unknown. In this case, the human expert is expected to provide an anomaly type label when an operation is compiled. This label is added to the feedback message.

2. The predicted anomaly type is incorrect. Therefore, the selected operations will probably not resolve the anomaly, and a human expert is notified at some point. It is expected that the expert corrects the misprediction and defines or selects an operation to resolve the ongoing anomaly. This information is added to the feedback message.

3. The predicted anomaly type is correct. In this case, the automatically selected operation can resolve the ongoing anomaly. Still, it is possible to configure the operation selection

to request a confirmation for the predicted anomaly type during the operation selection. This information is respectively included in the feedback message.

The symptom analysis receives the feedback message and the predicted anomaly type. Based on the information in the feedback message, it applies the label on the analyzed symptom patterns and accordingly updates the knowledge base. The feedback message is only received if the anomaly is resolved, i.e., an $A \mapsto N$ alarm is received. Therefore, the symptom analysis terminates after applying the labeling of the symptom patterns and storing them into the knowledge base. At this point, it can be invoked again by the ingest control if anomalies occur.

7.2.1 Preference Score Calculation

Based on the anomaly symptom recognition result, the operation selection module retrieves a set of feasible operations to resolve the ongoing anomaly. It further calculates a preference score for each operation in order to rank them. This preference score is a weighted sum of numerical attributes that are available for each operation. The concept is show in Table 7.1. The weights and all operations are listed as rows, while the different attributes and the preference score are arranged as columns. Calculating the preference score vector Σ can be expressed as a matrix multiplication

$$\Sigma = A\mathbf{w}, \tag{7.1}$$

where A is the matrix of all attribute values from $a_{1,1}$ until $a_{N,M}$, $\mathbf{w} = (w_1, \ldots, w_M)^T$ is the weight vector, and $\Sigma = (\Sigma_1, \ldots, \Sigma_N)^T$ is a vector containing all preference scores.

Table 7.1: Preference score calculation via weighted sum model.

	Attribute_1	Attribute_2	...	Attribute_M	Preference Score
Weights	w_1	w_2	...	w_M	
Operation_1	$a_{1,1}$	$a_{1,2}$	...	$a_{1,M}$	Σ_1
Operation_2	$a_{2,1}$	$a_{2,2}$	...	$a_{2,M}$	Σ_2
...	...	...	...	...	...
Operation_N	$a_{N,1}$	$a_{N,2}$	...	$a_{N,M}$	Σ_N

The attributes represent the numerical properties of the operations. The respective weights can be interpreted as the importance of the attributes. A higher weight will let the attribute contribute more to the preference score than a lower weight. Attributes can have a negative or a positive impact on the preference score. If a fast operation execution time should be preferred, the execution time of an operation should reduce the preference if the value increases. A high confidence value should usually increase the preference score. Therefore, the weights are signed. A negative sign implies a negative impact of the attribute value on the overall score, while a positive sign represents a positive impact. We define a limited set of attributes that are used for the preference score calculation. For this set of attributes, the user defines weights representing the preference towards specific attributes for the preference score calculation. Human involvement is required since it highly depends on the use case which operations should be preferred. The following attributes are defined:

1. The confidence value for the prediction anomaly type. This value is the same for all operations.

2. The average execution duration in seconds, based on the execution duration property of the operation execution knowledge base.

3. The ratio of successful executions of this operation to its overall number of executions. This value is calculated based on the "successfully executed" property of the operation execution knowledge base.

4. The ratio of executions that successfully resolved the anomaly for this operation to its overall number of executions. This value is calculated based on the "successfully remediated anomaly" property within the operation execution knowledge base.

After calculating these attributes for each operation of the feasible set, each attribute is respectively normalized via

$$a_{i,j} = \begin{cases} \frac{a_{i,j} - a_{i,min}}{a_{i,max} - a_{i,min}} & \text{, if } a_{i,max} \neq a_{i,min} \\ 1 & \text{, otherwise.} \end{cases} \tag{7.2}$$

where $a_{j,min}$ and $a_{j,max}$ are the minimum and maximum values of the respective column j. If all attribute values are equal, we assign 1 to prevent devision by zero. The normalization is required to make the attributes comparable. Ratios are defined between 0 and 1 while the execution time in seconds can arbitrary grow, which would result in an unproportionally high impact of the execution time on the preference score.

7.2.2 Integration into ZerOps

We integrate the ReCoMe engine module into ZerOps to realize the closed-loop control. The architecture is depicted in Figure 7.4. It shows a compute node that hosts two VMs as the SuO. All components in the diagram that are marked by the docker symbol are realized as containers. Dotted lines are used to mark components that are not the explicit focus of this thesis but are utilized to realize the closed-loop AIOps system. The Bitflow collector containers are deployed on each compute node and collects runtime metrics from the compute node itself and all VMs that it hosts. The metrics are respectively provided via endpoints as data input sources. In this example, the Bitflow collector provides three sources for the compute node and the two VMs. Whenever additional VMs are started on that compute nodes or if existing VMs are terminated or migrated, it respectively adds or removes endpoints. Bitflow can provide TCP endpoints, where a different port is used for each VM and compute node, or HTTP endpoints where the hostname of the compute nodes or the libvirt identifier of the VMs is used as URL paths. The collector allows configuring a buffer to store a certain amount of recent metric samples for each endpoint. The Bitflow operator checks the collector endpoints and adjusts the metric data processing pipeline. Due to the adding and removal of endpoints, the analysis pipelines are accordingly adjusted. It enables ZerOps to configure the data analyses according to the dynamics of the SuO. By utilizing the label matching configuration of the Bitflow operator, it is possible to define a specific mapping of data sources to processing steps. It is possible to spawn a one-to-one mapping or use a selector that connects a processing step to several data sources. The first processing step is anomaly detection, whereby one anomaly detection processing step is spawned for each Bitflow collector data source.

The modules of the ReCoMe engine are deployed as three containers. Due to their strong interdependence, the symptom analysis and the anomaly type inference are deployed in the

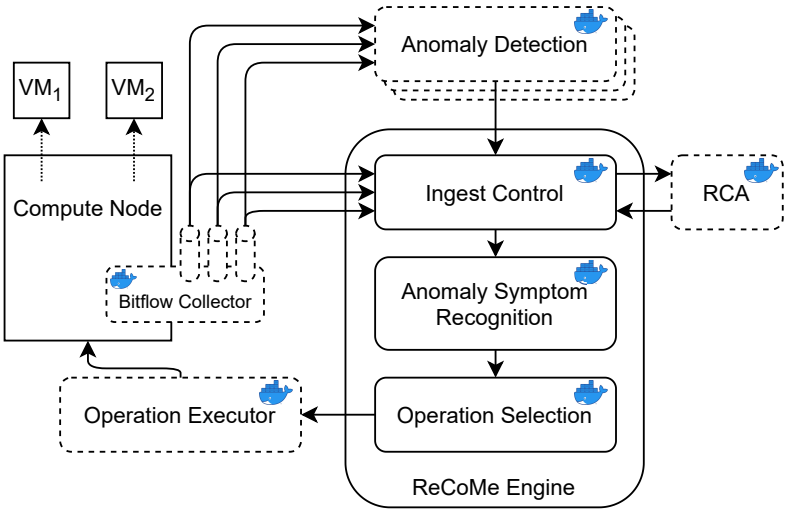


Figure 7.4: Closed-loop AIOps system containing the ReCoMe engine.

same container. The first module of the ReCoMe engine is the ingest control. It is configured to receive alarms from all anomaly detection processing steps. Furthermore, it is connected to an RCA module, requested whenever $N \mapsto A$ alarms from multiple system components are received. We deploy an ideal root cause analysis throughout our experiments that utilizes the ground truth to identify the root cause. The ingest control initializes the automated operation selection to resolve an anomalous component. Thereby, it requests all buffered metric series samples from the respective Bitflow collector endpoint and transmits them to the anomaly symptom recognition, which applies the symptom analysis and anomaly type inference. The results are transmitted to the operation selection, which eventually initializes the operation execution through the executor.

7.3 Deployment Strategies

ZerOps utilizes the Biflow operator to manage the monitoring data analysis methods as processing steps, which are deployed as Docker containers. The execution of the processing steps requires computational resources. We want to analyze two deployment concept variants for the AIOps system. One possibility is to provide dedicated nodes in a data center used to run all processing steps of the AIOps system. The second possibility is to co-locate processing steps with the regular workload on the compute nodes.

The two deployment concept variants are depicted in Figure 7.5. Concept 1 is shown on the left part as a set of compute nodes reserved for regular workload execution and another set of nodes where the AIOps system is deployed. A possibility is to select a fraction of available nodes - e.g., 5 % - for this. The advantage of this variant is that there is no interference between regular and AIOps workload. Also, the nodes' hard- and software specifications can be specialized to allow efficient and fast execution of the metric data processing steps. The disadvantage of it is that monitoring data need to be sent from compute nodes to the dedicated AIOps node. For a SuO that is located in a single data center, this aspect might be neglectable. However, when considering cloud computing systems that are spread across several data centers or fog environments, the monitoring metrics' transmission should be reduced. Especially for the lat-

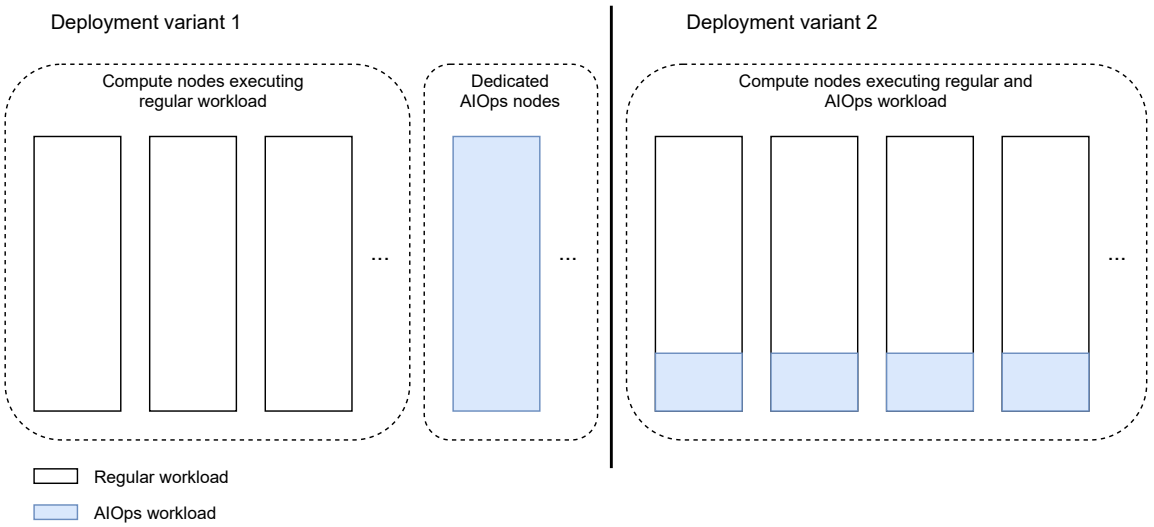


Figure 7.5: Two deployment concept variants.

ter, a major objective is to process data near their origin to prevent transmission via a carrier network.

Therefore, deployment concept 2 is considered. In Figure 7.5 it is depicted on the right side as a set of compute nodes that execute both regular and AIOps workload. We refer to it as *in-situ* (Latin for *in place*) data analysis [106]. A similar resource fraction assignment like for variant 1 can be done by reserving a fraction of available computation resources - e.g., 5 % - on each compute node for AIOps workload execution. Under ideal assumptions, the total amount of reserved resources for AIOps is the same for variants 1 and 2. However, variant 2 allows a natural scaling when the system is growing. No additional dedicated AIOps nodes are required. The reservation of computation resources can be adjusted based on the current requirements, and unneeded resources can be utilized by regular workload. This variant is streamlined with the computation paradigms introduced by edge and fog computing, where data should be analyzed near the source. With variant 2, we emphasize the placement of the monitoring data analysis steps on the compute node where the metric data are collected. Analyzing data close to their source also entails low latencies and locality, which are essential when dealing with imminent system component failures or when the network is unreliable between the fog and edge nodes and the dedicated AIOps nodes. Disadvantages are the additional placement logic requirement and the potential inference with a regular workload. A management unit that schedules the execution of AIOps workload is needed, which is the Bitflow operator. Furthermore, it utilizes Docker's ability to limit access to specific resources. Docker realizes the resource limitation via the "cgroups" feature of the Linux kernel [160]. Although it is possible to limit the utilization of CPU, memory, disk, and network resources for container processes, the executed workload and AIOps analysis processes still have inferences in the deeper parts of the system due to scheduling, the different CPU caching layers, and many others.

A mixture of both variants is possible. Whether analysis steps can be in-situ executed depends on their computational resource requirements. Processes requiring much memory (e.g., due to many model parameters) or executing complex CPU-intensive calculations might not fit the reserved fraction of computation resources. If metric data from several remote sources must be sent to a single processing step, an in-situ approach is also not ideal. Therefore, we suggest

a mixed deployment concept to benefit from the advantages of both variants while mitigating the respective disadvantages. Analysis steps can be deployed in-situ if

1. their resource requirements comply with the reserved fraction on the node where it should be placed, and

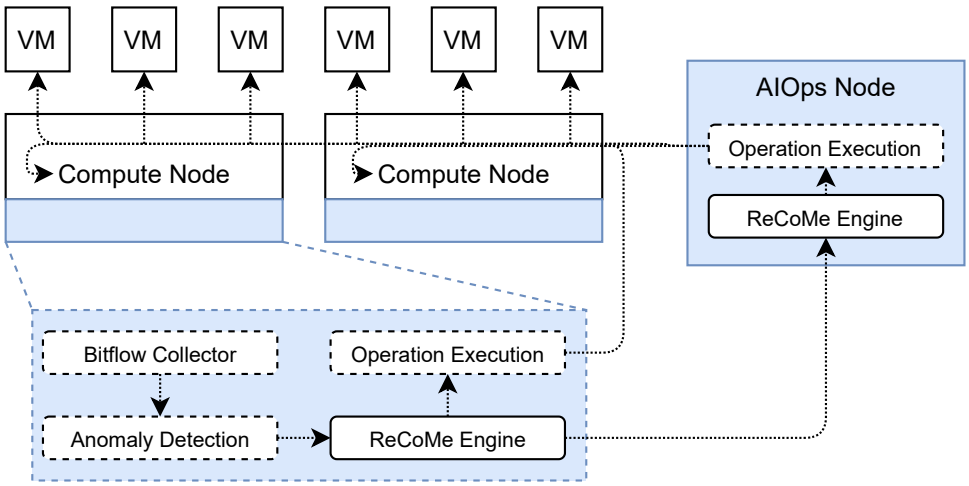
2. they do not require metric data from components located outside of the node on which they are placed.

The former is a hard requirement, while the latter can be relaxed by moving AIOps analysis steps as close as possible to the data sources.

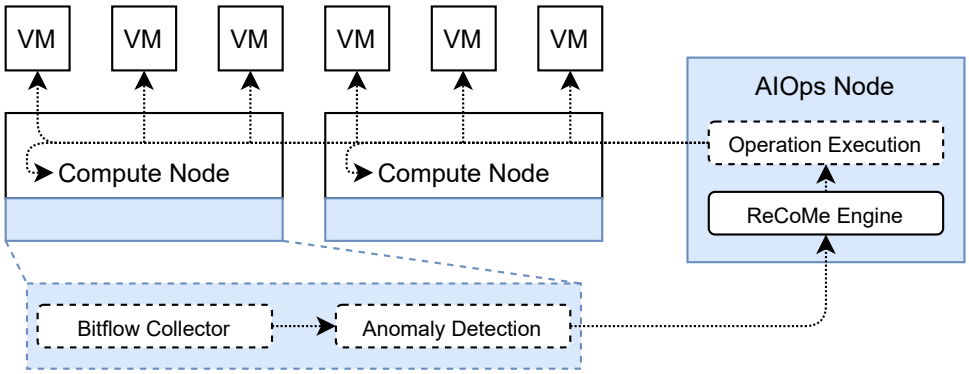
As presented in Subsection 7.2.2, the closed-loop control consists of different analysis steps. This modular design allows deploying steps either in-situ or on dedicated AIOps nodes. In the following, we present two deployment concept of our AIOps system. Both are depicted in Figure 7.6. As an example, we assume two compute nodes that are hosting virtual machines. Figure 7.6a and Figure 7.6b show two possible distributions of the processing steps between the reserved fraction of resources for in-situ analysis and a dedicated AIOps node. Note that the ReCoMe engine analysis steps are represented as a grouped ReCoMe entity in the diagrams, but indeed are three separate processing steps.

In Figure 7.6a, the Bitflow collector and the anomaly detection analysis step are deployed in-situ. Thereby, the metric data of the respective node and the individual VMs are observed and analyzed by the anomaly detection. Thereby, the anomaly detection can be interpreted as a filter. It does not raise alarms for normal component states, and monitoring data are not sent beyond the compute node. Since anomalies are expected to occur infrequently, this is the case for most of the collected metric data volume. Therefore, this deployment reduces the overhead caused by transmitting monitoring data to a central AIOps node. If the anomaly detection raises alarms, the ingest control of the ReCoMe engine requests a fraction of recent monitoring data of the anomalous SuO component. At the end of ReCeMe's process of analyzing metric data, an operation execution is eventually initialized. The operation execution module remotely executed the operation on the respective components. This variant assumes a reliable network connection between the compute nodes and the AIOps node. The transmission of data to the AIOps node and the remote operation execution might not be possible if the network connectivity is causing the anomaly. System components that are geographically distributed, as is the case in fog and edge computing, require options to resolve anomalies even if the network is unreliable.

In Figure 7.6b, the whole AIOps system is deployed in-situ. This enables autonomy for the respective compute nodes and must not rely on the network between workload and AIOps nodes. Considering a geographic distribution of nodes, the in-situ deployment of the whole AIOps system reduces the latency. A fast local reaction to detected anomalies is possible. The biggest challenge of this deployment concept is the aspect of limited resource availability. It demands a high resource efficiency from the utilized monitoring, anomaly detection, and ReCoMe processing steps. This requirement effectively excludes the utilization of complex models. Since anomaly detection is analyzing all monitoring data, this is a common assumption for anomaly detection models[177, 210, 212, 234, 243]. However, a strict resource limitation is challenging, especially for the anomaly symptom recognition of the ReCoMe engine, which should analyze metric series and identify anomaly types based on symptom patterns. Furthermore, anomalies can throttle the internal processes of a compute node. An over-utilization of



(a) Deployment concept 1: The ReCoMe engine is deployed on a dedicated AIOps node.



(b) Deployment concept 2: Two ReCoMe engines are deployed in-situ and on a dedicated AIOps node.

Figure 7.6: Two deployment concepts.

available computation resources is an example. Due to the potential internal unresponsiveness, operation execution from external nodes must be done to resolve the anomaly. Based on these considerations, we propose a fall-back strategy. Another ReCoMe engine instance, together with an operation execution module, is located on a dedicated AIOps node. Whenever the anomaly symptom recognition cannot be done via a resource-efficient local model, the problem is delegated to the ReCoMe engine running on the AIOps node. There, it is possible to employ models of higher complexity to perform the pattern analysis. Furthermore, the operation execution module can execute remediation operations if a compute node cannot perform the remediation locally. This also enables the handling of propagating anomalies, where components beyond one compute node are affected, by relaying the responsibility to the ReCoMe engine running on the dedicated AIOps node.

The effect and intensity of advantages and disadvantages of both deployment concepts highly depend on the use case and the employed AIOps analysis step implementations. With the proposed design, we aim to allow a flexible and modular AIOps system setup.

7.4 Evaluation

We conduct experiments to evaluate the AIOps system containing our ReCoMe engine. Therefore, we first analyze the runtime of the two anomaly symptom recognition methods to infer implications for analysis step placement. It should be investigated which data processing can be executed in-situ. After that, we describe the experimental design for deployment concept testing. The evaluation environment is extended to allow a simulation of fog environments. Thereby, the compute nodes are separated into groups that represent fog and cloud computing environments. Anomaly injection experiments are conducted there. We test the AIOps system by applying the two deployment concepts on the cloud and fog computing environments and observe the system's ability to automatically infer anomaly types and automatically select operations to resolve the anomaly.

7.4.1 Runtime Evaluation

The deployment concepts of the AIOps analysis steps highly depend on their runtime efficiency. We consider two phases when investigating the analysis step runtime: the training and prediction phase. During training, all available anomaly metric series are used to set up the model. This is either done by the density grid transformation for the density grid pattern model or by tuning the GRU RNN model parameters. After that, the model can be employed for prediction, where incoming series are searched for symptom patterns to infer the anomaly type. We conduct runtime measurements for both models during the training and prediction phase. Throughout all measurements, the datasets from the experiments described in Subsection 5.4.2 and Subsection 6.4.1 are used.

Both anomaly symptom recognition methods and the metric series augmentation are implemented in Python [7]. During our experiments, Python version 3.7.6 [8] is used. The density grid model is a custom implementation. The GRU RNN is realized with PyTorch version 1.7.0 [183]. For the metric series augmentation the libraries statsmodels version 0.12.1 [9], tslearn version 0.5.0.2 [10], and recombinator version 0.0.5 [11] are used. All model parameters are set as defined in Chapter 5 and Chapter 6. The time measurement is realized via a performance counter. Since the symptom pattern recognition methods are implemented in Python, Python's *perf_counter* is used. It returns a value in fractional seconds of the operating system's internal clock. Thereby, the time value reference is not defined, which means that the difference between the return values of two *perf_counter* function calls must be calculated to acquire a valid duration [12].

First, we measure the time that is needed to set up the respective prediction model. The measurements are performed based on 14 available training examples (corresponds to a 70 % split) for each component and anomaly type. There are nine components (the compute nodes, six Clearwarter, and two video on demand services) and eight anomaly types. A model is

[7] `https://www.python.org/` (last access 25 May 2021)

[8] `https://www.python.org/downloads/release/python-376/` (last access 25 May 2021)

[9] `https://www.statsmodels.org/v0.12.1/` (last access 25 May 2021)

[10] `https://github.com/tslearn-team/tslearn/` (last access 25 May 2021)

[11] `https://github.com/InvestmentSystems/recombinator` (last access 25 May 2021)

[12] `https://docs.python.org/3/library/time.html#time.perf_counter` (last access 25 May 2021)

Table 7.2: Runtime results of metric data processing steps during the training phase.

Data Processing Step	Time in [s]		
	Median	**Mean**	**Standard deviation**
Density Grid Pattern Model Grid Transformation	1.94	1.24	0.11
Metric Data Augmentation	1854.24	1903.98	134.65
GRU Model trained on CPU	7639.37	7621.45	240.29
GRU Model trained on GPU	1549.97	1548.63	11.28

trained for each component to distinguish the eight anomaly types. We measure the time until all models are trained. The training of each model is executed sequentially. While the grid transformation is performed on a regular cluster node, we measure the time to train a GRU model on a cluster node's CPU and a GPU of an AIOps node. Furthermore, the GRU model is trained based on augmented anomaly type examples. Thus, the time needed to generate 500 augmented examples is measured as well. Each of the four data processing steps is executed ten times and we calculate the mean, median, and standard deviation in seconds. The results are listed in Table 7.2. As expected, the training of the GRU model requires significantly more time than the grid transformation. The difference lies within three orders of magnitude. Our density grid model requires 1.24 seconds to process all available anomaly training examples. The GRU model is trained for 300 epochs, which is the number of iterations over the available training examples. It requires approximately 2 hours to train it on a CPU and 25 minutes on a GPU. Furthermore, it takes approximately 30 minutes to generate 500 augmented training examples. Depending on the availability of computational resources, the augmentation and model training can be executed in parallel with an initial offset for the training to generate the first batch of examples. Notably, the standard deviation for the GRU training on the GPU is smaller than the one for CPU training. This difference is explained by the fact that the GPU is exclusively reserved for model training calculations. The CPU, on the other hand, is concurrently accessed by all running processes (at least the operating system and the docker container management) that are running beside the model training process.

Table 7.3: Runtime results of metric data processing steps during the prediction phase.

Data Processing Step	Time in [ms]		
	Median	**Mean**	**Standard deviation**
Density Grid Pattern Model Grid Transformation	11.00	11.77	4.34
Density Grid Pattern Model Grid Pattern Comparison	3.70	11.41	-
GRU Model on CPU	8209.73	8312.97	326.16
GRU Model on GPU	9690.13	9686.75	80.93

Second, we investigate the prediction time. Thereby, the time a model needs to process one metric series and predict the anomaly type is measured. Again, the models are trained on 14 available examples of each anomaly type (corresponds to a 70 % split) for every component. After that, the remaining six examples are used for prediction. Each example is processed ten times resulting in 60 measurements for each anomaly and component. The density grid model

predictions are executed on the CPU of a regular cluster node. Thereby, the series is first transformed into a density grid and eventually compared to the available density grids to calculate the similarities. We respectively measure the time for both data processing steps. For the overall prediction time, they need to be added. The prediction time of the GRU model is measured when executed on a CPU of a regular cluster node and a GPU of an AIOps node. Thereby, no sample batching is done. Every sample is processed individually. No metric series augmentation is applied during the prediction phase. The results are listed in Table 7.3. Note that the mean, median, and standard deviation are calculated in milliseconds. The prediction time of the GRU model is two orders of magnitude higher than the density grid prediction time. The mean time for the grid transformation of a single series is 11.77 milliseconds. For the grid pattern comparison, a mean time of 11.41 milliseconds is required, resulting in 23.18 milliseconds for density grid model prediction time. The mean prediction time for the GRU model on CPU and GPU is 8.2 and 9.7 seconds. The faster prediction on the CPU can be explained by the missing sample batching. Transferring each sample to GPU and back results in additional overhead and is eventually reflected in a longer prediction time for an anomaly series. The effect regarding the higher standard deviation on the CPU compared to GPU can be observed again. The grid pattern comparison times are heavily skewed, as indicated by the difference between the mean and median. The calculated standard deviation is 23.79 milliseconds, resulting in negative time values if we assume a normal distribution. Therefore, we further investigate the reason for the skew.

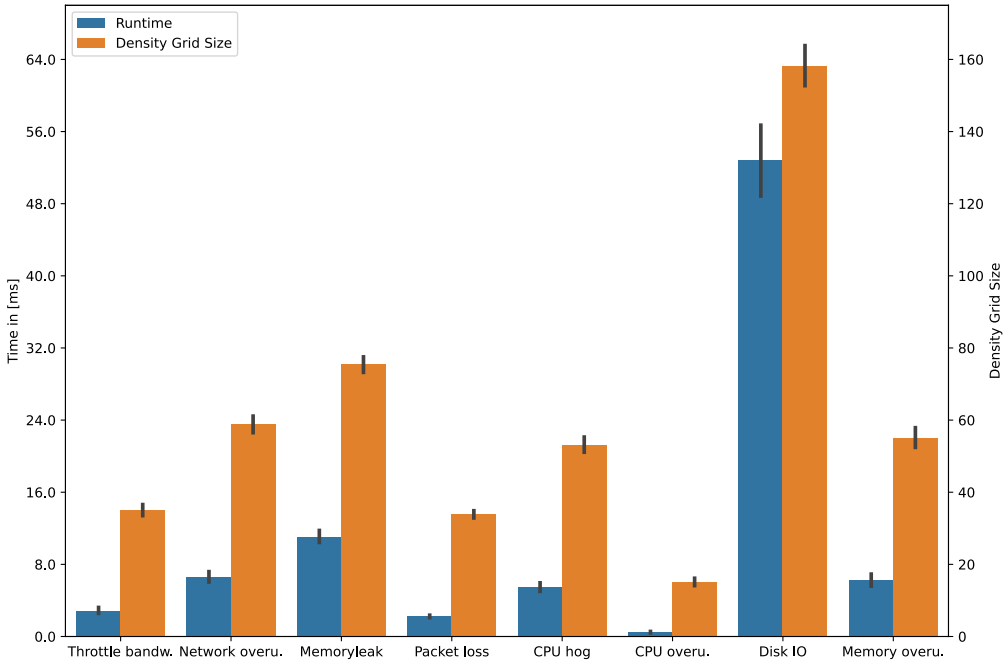


Figure 7.7: Runtime results of density grid comparison for each anomaly type.

We analyze the grid pattern comparison runtime depending on the anomaly type, which is visualized in Figure 7.7. The x-axis depicts all eight anomaly types. The blue bars represent the measured runtime values for the density grid comparison, while the orange bars show the density grid sizes, which is the number of non-zero cells in the density grid pattern. The left and right y-axes are scaled to show the values of the blue and orange bars, respectively. A 0.95

confidence interval is shown as error bars on top of each bar. A high difference in the runtime values between anomaly types can be observed. Calculating the grid similarities for the CPU overutilization anomaly requires a mean value of 0.47 milliseconds, while the calculation for the disk IO anomaly takes a mean runtime of 52.9 milliseconds. These high differences cause the skewness of the distribution if collectively considering all anomaly types. We further investigate the relationship between the density grid size and the density grid comparison runtime, revealing a Pearson correlation coefficient of 0.97. Therefore, a high correlation between the amount of non-zero grid cells and the grid pattern comparison runtime exists. This complies with the theoretical runtime analysis of the density grid model from Subsection 5.1.3. Anomalies that affect multiple computation resources and cause high variation result in higher numbers of non-zero grid cells, leading to higher grid pattern comparison runtimes.

The runtime analysis result has implications regarding the placement of the anomaly symptom recognition methods. The GRU model has a high computation resource requirement for the training phase. We argue that an in-situ execution of GRU model training, where the availability of computing resources is further limited, is not feasible. They should be trained on dedicated nodes. Furthermore, the time that is needed to train such models poses a limitation for frequent retraining. Thereby, a trade-off must be made between incorporating recent information and the high resource utilization during training. The runtime for the density grid model's training and prediction is several orders of magnitude lower compared to the GRU model. Consequently, the density grid model training of and prediction can be executed in-situ due to its efficiency.

7.4.2 Experimental Environment

We utilize the evaluation environment describe in Section 5.3 but adjust it to simulate the properties of a fog computing system. Furthermore, we add one node to the system equipped with graphics processing unit (GPU) hardware to accelerate the RNN model training. The hardware specifications of this node can be found in Table 7.4. We refer to it as *GPU node*.

Table 7.4: Hardware specifications of compute node for RNN model training.

Resource Type	Specification	Quantity
Processor	Intel Xeon Silver 4208 2.1GHz 8 Cores	1
Graphical Processor Unit	NVIDIA Quadro RTX 5000 16GB GDDR6	2
Main Memory	DDR4 16GB 2400MHz	6
Storage	Micron 2200 M.2 NVMe 1TB 3GB/s	2
Network	Intel X550 10GBit Ethernet Controller	2

The experiment is conducted on 17 commodity cluster nodes and one GPU node. We deploy OpenStack Stein[13] as a cloud resource management system on 12 nodes. Three of them are load balanced with haproxy [14] and used as OpenStack network and controller hosts, while nine compute nodes host the virtual machines. Two AIOps system nodes are reserved for running the ZerOps platform services exclusively. One of them is a regular commodity cluster node, while the other is the GPU node. Three nodes contain the processes that simulate client access, and one is used as a host for the anomaly injection controller.

[13] `https://releases.openstack.org/stein` (last access 06 May 2021)
[14] `http://www.haproxy.org/` (last access 30 May 2021)

Since the nodes are located in a LAN, connected via a maximum of three hops over switches, they do not meet the requirement of a fog environment. We utilize the traffic shaping tool tcconfig [15] which allows configuring a limited bandwidth, unreliable network, or increased latency between arbitrary endpoints. The tool is used to configure latencies between nodes to simulate a fog-like environment. The setup is depicted in Figure 7.8. A cloud environment group with five compute nodes and two fog environment groups consisting of respectively two compute nodes provide the computational resources. OpenStack's host aggregate and availability zone functionalities are used to realize the compute node grouping [16]. All outgoing packets of respective groups are delayed to simulate geographical distance. It results in an increased latency between the respective node groups. The packet delay is only applied between compute nodes of different groups but not between nodes of the same group. The delay is set to 100 ms between the client nodes and the fog groups and 150 ms between the client nodes and the cloud group. Furthermore, a delay of 50 ms is configured between the two fog groups and between the fog groups and the cloud group. The cloud group contains the OpenStack controller and network and the AIOps nodes.

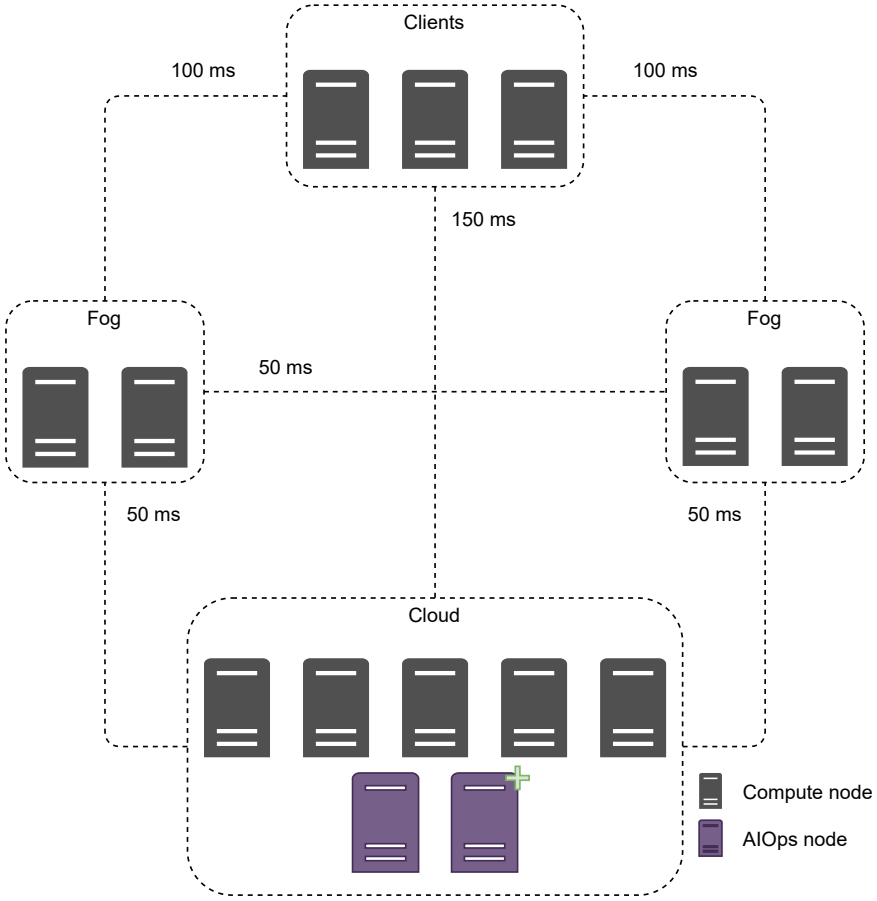


Figure 7.8: Simulated fog environment.

Both service use cases are deployed on the available compute nodes. For Clearwater, we

[15] https://github.com/thombashi/tcconfig (last access 22 May 2021)

[16] https://docs.openstack.org/nova/latest/admin/availability-zones.html (last access 22 May 2021)

deploy the bono VMs within each fog group on respectively one compute node. All other Clearwater service VMs are deployed on the compute nodes within the cloud. The numbers of Clearwater service VMs are listed in Table 7.5. For the video on demand, two load balancer VMs and one video server VM are deployed on the other compute node within each fog group. Furthermore, three video servers are deployed on the compute nodes of the cloud group. The Clearwater VMs are provided with one virtual CPU, 2GB RAM, and 40GB disk space, while video on demand VMs run with two virtual CPUs, 4GB RAM, and 80GB disk space. All VMs operate with Ubuntu 16.04.3 LTS under Linux kernel version 4.4.0-128-generic. Two SIPp processes are executed on one node of the client group to generate load against Clearwater. They are configured to generate between 300 and 500 user registrations and call initiations every second. The other two nodes of the client group execute two video client simulation processes that simulate between 30 to 50 video requesting users. After finishing a video stream, a simulated user waits for 10 seconds to 5 minutes to select the next video. The video and its resolution were randomly selected. The monitoring agents are configured to collect system metric data at a frequency of 1Hz. All metrics that were collected throughout the experiment are listed in Table 7.6. The metrics are measured either in bytes, percent %, or a counting value #.

Table 7.5: Number of VMs for each Clearwater service.

Component	bono	sprout	homer	homestead	chronos	cassandra	astaire
Num. of VMs	4	4	2	2	3	3	3

Table 7.6: Metrics that are monitored during the experiment.

Resource	**Metrics**	**Unit**
CPU	cpu utilization	%
Memory	allocated memory	%
Disk	written data	bytes
	read data	bytes
	I/O operations	#
	used disk volume	%
Network	rx volume	bytes
	tx volume	bytes
	rx packets	#
	tx packets	#

In this experiment, the AIOps closed-loop control should be evaluated. We want to test the system's ability to recognize anomaly types automatically select operations to resolve the anomaly. Therefore anomalies were injected into the VMs and compute nodes. A list of injected anomalies with their parametrization is listed in Table 7.7. Every anomaly was injected 15 times into each Clearwater and video on demand service (i.e., bono, sprout, or video server) respectively for each group. Furthermore, they were injected respectively 15 times into the three compute node groups. The duration of every injection was set to three minutes, after which one minute recovery period without injections was defined. The experiment has an initial 2 hour period without injections, resulting in an overall experiment duration of 80 hours.

Table 7.7: Description and parametrization of anomalies.

	Description		
Anomaly	**Clearwater VMs**	**VoD VMs**	**Compute Nodes**
CPU overutilization	Utilize all cores at 90 - 100%.		
CPU hogging	Step-wise increase the CPU utilization. Increase by 1 % every second until 100 %.		
Memory leak	Step-wise allocate memory. x MB every y seconds until z MB.		
	x=4, y=1, z=600	x=6, y=1, z=1000	x=20, y=1, z=3000
Memory overutilization	Allocate x MB of memory.		
	x=600	x=1000	x=3000
Disk IO	Start 3 processes that write / read data to / from disk.		
Network Overutilization	Repeatedly start downloading a large file from the internet.		

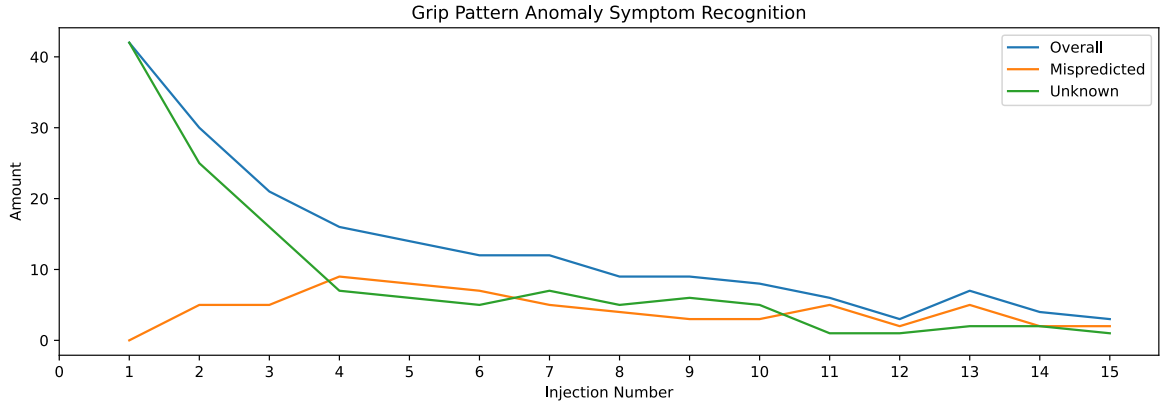
ZerOps utilizes the Bitflow operator to connect metric data sources and processing steps into a closed-loop AIOps system. We deploy OpenStack and the service use cases, run the anomaly injection experiment and store the metric data locally on the respective compute nodes. The data storage is done to prevent a repeated execution of the experiment Thereby, the origin (respective VM or the compute node) of the metric data is preserved by storing the metric data into separate files. After that, we remove OpenStack and deploy Kubernetes version 1.19 [17] and ZerOps, whereby the compute and AIOps nodes are defined as Kubernetes worker nodes. We provide 10 % of the available computation resources in each compute node for the AIOps processes. No resource limitations are applied on the AIOps nodes. The Biflow collector is configured to read the locally stored files and provide metric data endpoints. It buffers 45 metric samples, which can be requested by the ReCoMe engine's ingest control if an anomaly is detected. Throughout our experiments, we use an ideal anomaly detection based on the ground truth.

7.4.3 AIOps in Cloud Environments

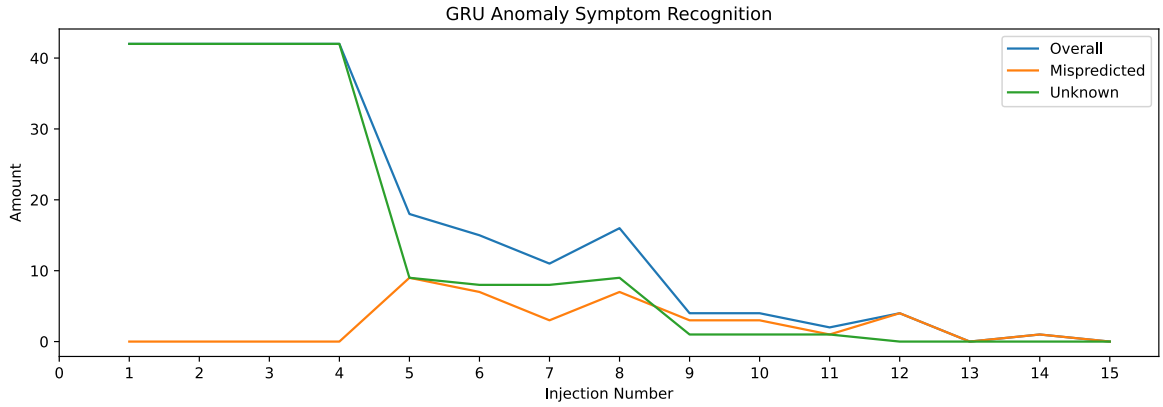
The cloud group represents the compute nodes that are located in the same data center. Carrier networks are not involved when data samples are transmitted between the compute and AIOps nodes. Therefore, we utilize deployment concept 1, whereby the Bitflow collector and anomaly detection containers are deployed in-situ, while the ReCoMe engine and the operation executor are located on a AIOps node. The Bitflow collector is configured to read the metric data files at a frequency of 1Hz, the original sampling frequency when the data were collected. For comparison, two ReCoMe engines are deployed respectively utilizing the density grid and GRU models for anomaly symptom recognition. The engines are deployed separately on the two available AIOps nodes. Whenever anomalies are detected, the ingest control requests the buffered samples from the Bitflow collector and forwards them to the anomaly symptom recognition. The GP anomaly symptom recognition is updating its symptom pattern knowledge base after each observed anomaly. Since the retraining of the GRU model requires more time and computational overhead, we employ retraining after observing each anomaly type 4, 8, and 12 times. The GRU model is using all recorded metrics while the grid pattern model utilizes a

[17] `https://v1-19.docs.kubernetes.io/` (last access 22 May 2021)

partly aggregated subset of the collected metric data, which are: CPU utilization in percent, a fraction of allocated memory in percent, the sum of disk read and write data volume in bytes, and the sum of rx and tx network traffic volume in bytes. We assume an empty knowledge base in the beginning and thus, consider the cold start problem. After the anomaly type is predicted, the results are sent to the operation selection. In this experiment, we assume a single operation for each anomaly type to resolve it. If the correct anomaly type is predicted, the operation selection chooses the respective operation and sends this information to the operation executor. The executor holds the ground truth, compares it with the selected operation, and sends respective feedback to the operation selection. If the "unknown" type is predicted, no operation is selected. The operation selection module logs the amount of "unknown" anomaly type predictions, correct and incorrect operation selections.



(a) Grid pattern anomaly symptom recognition model.



(b) GRU anomaly symptom recognition model.

Figure 7.9: Mispredictions results for the ReCoMe engine using the two different anomaly symptom recognition models.

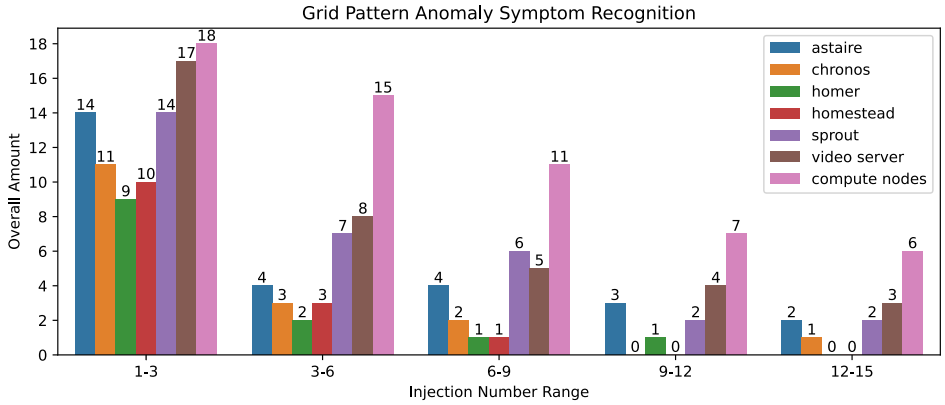
First, we evaluate the number of mispredictions and "unknown" predictions depending on the amount of occurred examples per anomaly type. The results are depicted in Figure 7.9, whereby Figure 7.9a shows the results for the ReCoMe engine with the density grid anomaly symptom recognition and Figure 7.9b contains the results of the ReCoMe engine with the GRU anomaly symptom recognition. The x-axis represents the injection number. Each tick means

that all six anomaly types were injected for the respective amount. The y-axis depicts the amount of mispredicted examples (orange line), "unknown" predictions (green line), and the sum of both (blue line). Five Clearwater, one video on demand, and the compute nodes result in seven injection targets for the six anomaly types, resulting in 42 injected anomalies per x-axis tick. The plot shows for how many of these 42 injections, operations could not be selected automatically due to an unsuccessful anomaly symptom recognition. Therefore, the blue line can be interpreted as the amount of human involvement. The orange line represents incorrect operation selection while the green line shows human involvement due to false recognition of known anomaly type as "unknown".

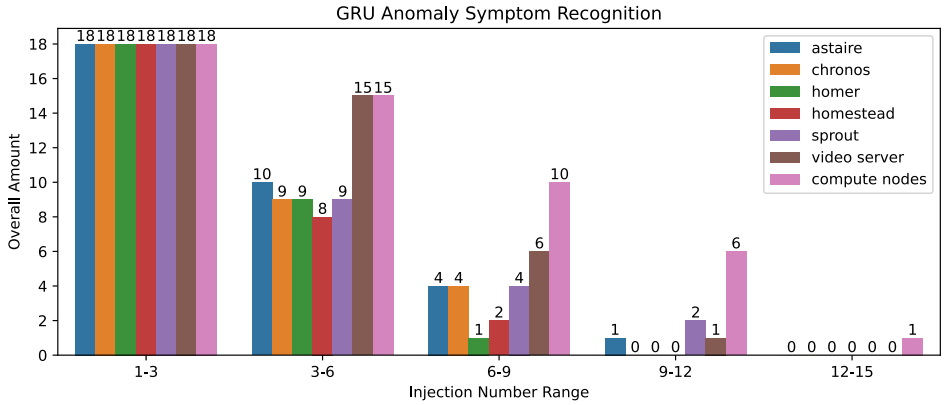
The injection number starts at 1, which means that the knowledge base is empty, i.e., representing the cold start problem. Thereby, the anomaly type inference receives an empty result list from symptom analysis and sets the predicted anomaly type as "unknown". Therefore, all 42 injected anomalies are predicted as "unknown". The grid pattern model is updated after every anomaly is received once, resulting in an ability to predict anomaly types at the second injection number already. For the second, third, and fourth injection, the mispredictions drop to 30, 21, and 16 whereof 25, 16, 7 are predicted as "unknown". The first training of the GRU model is applied after the fourth injection. Until then, all anomaly types are predicted as "unknown", which represents the requirement of human experts to handle these anomalies and thereby create a training dataset for the model. The employment of the first trained model can be seen as a drop in the overall misprediction curve. The trained model is used until injection number 8, resulting in between 11 and 18 overall mispredictions, whereof between 8 and 9 are predicted as "unknown". In comparison to that, the grid pattern model's predictions between injection numbers 5 and 8 are ranging between 12 and 16 with between 5 and 7 "unknown" predictions. Another GRU retraining is done after injection number 8, again resulting in a misprediction drop. Between injection numbers 9 and 12, they range from 2 to 4, compared to 3 to 9 mispredictions of the density grid model. The final retraining of the GRU model is employed after the 12th injection number, whereafter only one misprediction occurs between injection numbers 13 and 15. The number of mispredictions saturate for the density grid model and remains between 3 and 7 mispredictions.

It can be observed that the GRU model requires a more extended warm-up period. The training after the fourth injection number renders it incapable of performing predictions prior to that. Contrary to that, the fast knowledge base update of the grid pattern model allows it to be employed for prediction after the very first occurrences of anomalies. After the fourth injection number, both models yield a similar misprediction amount. After the eighth injection number, the GRU model outperforms that density grid model. The GRU model's ability to capture complex sequential dependencies in the monitoring data requires more training to come into effect but allows the model to improve and yield one misprediction after the last retraining. The density grid model's fast knowledge base extension allows it to outperform it with few available training examples. However, its limitations cause a saturation after the 11th injection number preventing it to improve further.

Next, we investigate the distribution of the mispredictions across the different components. In this experiment, five Clearwater services and the video servers are deployed on VMs running on cloud group compute nodes. The results are depicted in Figure 7.10, where Figure 7.10a shows the grid pattern and Figure 7.10b the GRU model results. We aggregate the mispredictions over ranges of three injection numbers and group the bars based on these ranges. Each bar represents a service (the five Clearwater services and the video server) or the compute nodes of



(a) Grid pattern anomaly symptom recognition model.



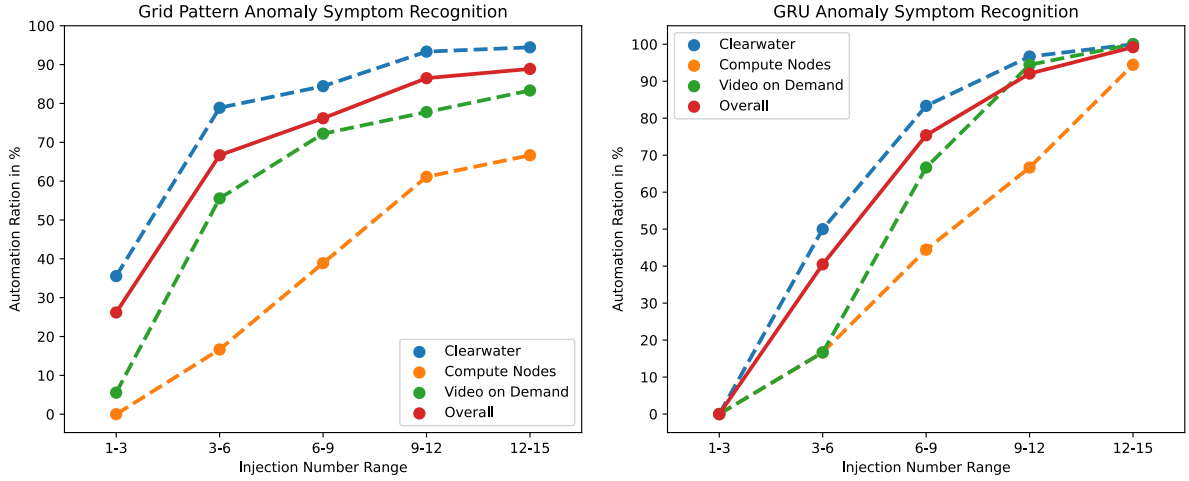
(b) GRU anomaly symptom recognition model.

Figure 7.10: Distribution of the mispredictions across the different components for the two anomaly symptom recognition models.

the cloud group. No distinction between mispredictions and "unknown" predictions is made. Note that the bono and load balancer services are deployed on the fog compute nodes and are not represented here. Six anomalies are injected on each of the components represented by a bar. The aggregation over three injection numbers results in a maximum bar height of 18.

For the GRU model, the 18 mispredictions at the 1-3 range are due to the first training of the model after the fourth injection number. For the grid pattern model, it can be seen that it performs better on Clearwater services. This is generally the case also for the higher injection ranges. The same effect can be observed for the GRU model. Clearwater employs a microservice architecture with a comparably fine-grained separation of responsibilities across the different services. This decoupling results in an overall homogeneous workload and reduces the complexity of the observed anomaly type-specific symptoms. More precisely, the intra-type variation for Clearwater services is smaller than for components that execute heterogeneous workload, such as the compute nodes hosting VMs with different services deployed within. The recognition of anomaly types injected on the video servers is increasingly difficult. The video server reads the locally stored video data, applies the encoding and buffering, and transmits them via the load balancer to the requesting client. Different videos and resolutions result in

a broader workload spectrum and increase the intra-type variation of anomaly type symptoms. In most of the depicted injection number ranges, this results in higher misprediction amounts than the Clearwater services. The compute nodes represent the highest workload variation. They host different VMs, and the co-location of the VMs differs between the compute nodes, resulting in the parallel execution of different workloads. Due to the high intra-type variation of anomaly symptoms, it is hard for models to assign observed symptom patterns to anomaly types. It requires more examples of each anomaly type to improve the prediction performance and achieve lower numbers of mispredictions. Even after observing more than ten examples of each anomaly type, the density grid model mispredicts between 6 and 7 examples for the last two ranges. In contrast to that, the GRU model mispredicts only one injected anomaly example in the 12-15 injection number range.



(a) Grid pattern anomaly symptom recognition model. (b) GRU anomaly symptom recognition model.

Figure 7.11: Percentage of cases for which an operation can be automatically selected.

The amount of correctly predicted anomaly types can be interpreted as the number of cases where operations can be automatically selected to resolve the ongoing anomaly. Therefore, we calculated the percentage of cases where the anomaly symptom recognition correctly predicts the anomaly type. The results are depicted in Figure 7.11. Again we distinguish between the grid pattern model in Figure 7.11a and the GRU model in Figure 7.11b. The x-axis shows injection number ranges, whereby the correct and incorrect predictions were aggregated over three injection numbers. The plots show results respectively for the compute nodes, the video on demand, and the Clearwater services. Note that five Clearwater services exist. Compared to one video on demand and the compute nodes as injection targets, the Clearwater services have a higher impact on the overall result. Assuming that a correct anomaly type prediction allows a selection of operations automatically, we refer to the ratio of correctly predicted anomaly types to the overall occurred anomaly types as the automation ratio, which is presented in percent.

For the grid pattern model, it can be seen that the homogeneous workload of the Clearwater services enables a correct recognition of anomaly types if few examples are available. This yields a early overall automation ratio of 26 %, which further increases to 66 %, 76 %, 87 %, and 89 %. For the Clearwater services, the grid pattern model can achieve an automation ratio of 94 %, while the ratio for the compute nodes reaches a maximum of 67 %. Since the GRU model is trained after the fourth injection number, its automation ratio remains at 0 % for the

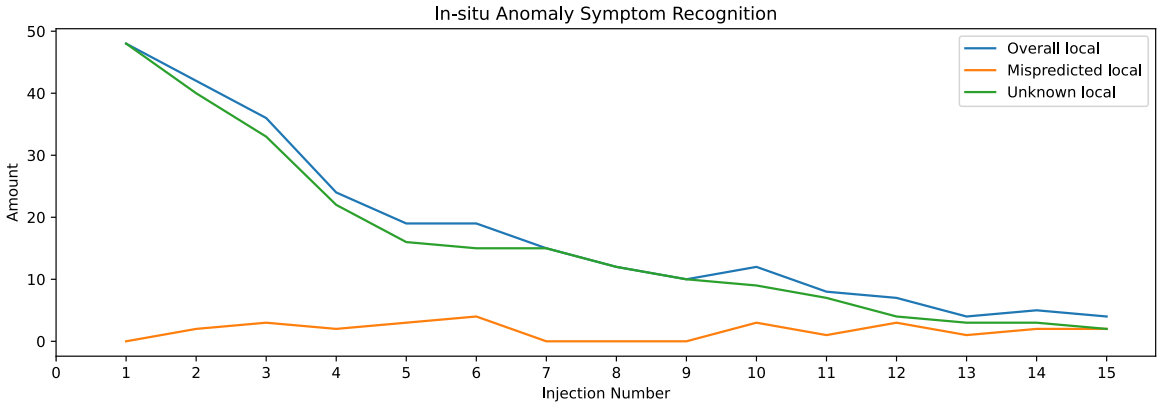
1-3 range. After that, it achieves comparable results to the grid pattern model of 75 % for the injection range 6-9 and outperforms it for the ranges 9-12 (92 %) and 12-15 (99 %). Again it can be seen that the grid pattern model yields better results when having few training examples available. The GRU model requires more examples to meet the early automation ratio results of the grid pattern model. With the increasing availability of training examples, it can achieve a higher automation ratio, eventually reaching 99%.

7.4.4 AIOps in Fog Environments

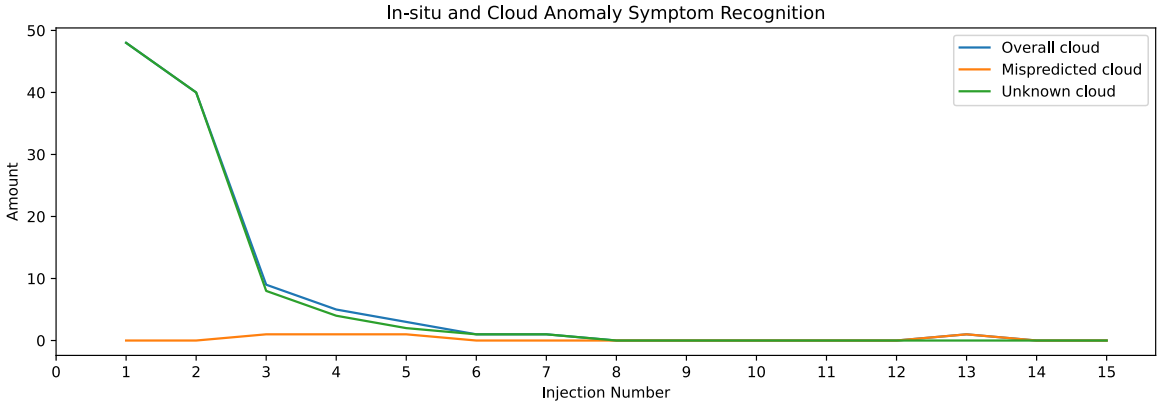
In the following experiments, the services that are deployed on the fog group nodes are evaluated. These are the Clearwater bono service, the video on demand balancer and video server services, and the compute nodes of the two fog groups. Note that each anomaly was injected 15 times into the services and the compute nodes respectively for each fog group. We assume a geographic distance between the compute nodes of the fog groups and between the fog groups and the cloud group. Furthermore, we expect the fog environment to have a limited amount of computation resources available, reflected in the two compute nodes per fog group. In a fog environment, it is infeasible to separate a whole compute node for AIOps. Also, the monitoring data should be processed at their origin to reduce monitoring data transfer via the carrier network. Therefore, we utilize the deployment concept 2, whereby the Bitflow collector, anomaly detection, ReCoMe engine, and operation executor are deployed in-situ. An in-situ deployed AIOps system must realize the data analysis with a limited amount of resources, which is why the grid pattern anomaly symptom recognition is used. Due to the limitations of grid pattern anomaly symptom recognition to recognize complex patterns in the monitoring data series, we also deploy a ReCoMe engine and an operation executor on one AIOps node in the cloud. The GRU anomaly symptom recognition is used there.

Matching anomaly types is realized as a two-step approach. First, the Bitflow collector is configured to read the metric data files at a frequency of 1Hz. Whenever anomalies are detected, the in-situ ingest control requests the buffered samples from the Bitflow collector and forwards them to the in-situ anomaly symptom recognition. The recognition results are forwarded to the in-situ operation selection module, which assumes a single operation for each anomaly type to resolve it. If the correct anomaly type is predicted, the operation selection chooses the respective operation and sends this information to the in-situ operation executor. The executor holds the ground truth, compares it with the selected operation, and sends respective feedback to the operation selection. If the anomaly is reported to be resolved, i.e., the correct anomaly type was predicted, the anomaly is assumed to be resolved locally. If the feedback states that an incorrect operation was chosen or if the anomaly type is predicted to be "unknown", a message to the in-situ ingest control is sent, transmitting all metric samples that it received until this point to the ingest control deployed on the remote AIOps node. The remote ingest control initializes the anomaly symptom recognition on the AIOps nodes. The GRU model is used, which we respectively retrain after each anomaly was observed 4, 8, and 12 times. The in-situ ingest control is configured to preprocess the metrics as a partly aggregated subset of the collected metric data, which are: CPU utilization in percent, a fraction of allocated memory in percent, the sum of disk read and write data volume in bytes, and the sum of rx and tx network traffic volume in bytes. However, it maintains the original metrics to eventually forward them to the remote ingest control, utilizing all recorded metrics. An empty knowledge base is assumed initially, and thus, the cold start problem is considered. The metric data of observed anomaly type examples

are expected to be available for training both the in-situ and remote anomaly symptom recognition models. The in-situ and remote operation selection modules log the amount of "unknown" anomaly type predictions, correct and incorrect operation selections.



(a) In-situ anomaly symptom recognition.



(b) In-situ and remote cloud anomaly symptom recognition.

Figure 7.12: Mispredictions results for the ReCoMe engine using either only the in-situ deployment or the in-situ on combination with a remote deployment.

First, we examine the anomaly symptom recognition performance of the AIOps system. Thereby, it is investigated whether the anomaly type can be predicted locally or in combination with a ReCoMe engine deployed on a dedicated AIOps node in the cloud. The results are depicted in Figure 7.12. Figure 7.12a shows the results for the in-situ executed anomaly symptom recognition while Figure 7.9b represents the combined anomaly symptom recognition. The x-axis represents the injection number, and the y-axis depicts the amount of mispredicted examples (orange line), "unknown" predictions (green line), and the sum of both (blue line). The two fog groups, each containing the bono, load balancer, and video server services together with the compute nodes, result in eight injection targets. The injection of all six anomaly types into each target results in 48 injections, which is the maximum number of mispredictions for each injection number. The plot shows for how many of these 48 injections the correct anomaly type is predicted. The upper plot shows the result of the in-situ anomaly symptom recognition only, while the lower plot shows the combined results for in-situ and remote anomaly symptom

recognition.

The GRU models are retrained after 4, 8, and 12 examples of each anomaly type are available. Since two fog groups exist, the examples are respectively available at injection numbers 2, 4, and 6. For the in-situ density grid model, the knowledge base is extended after each injection number. Therefore, the first injection number yields 48 in-situ and remote cloud "unknown" predictions. After that, the available examples enable the in-situ anomaly symptom recognition to reduce the mispredictions to 42. Relaying these 42 examples to the GRU anomaly symptom recognition result is 42 "unknown" predictions since the model is not trained yet. For injection numbers 3 and 4, the in-situ anomaly symptom recognition mispredicts 36 and 24 examples. Since the GRU model was trained on four anomaly type examples from both fog groups, it can reduce the misprediction numbers to 9, and 5 whereof 8 and 4 were predicted as "unknown". The subsequent retraining of the GRU model is executed after injection number 4. For injection numbers 5 and 6, the in-situ anomaly symptom recognition mispredicts 19 examples, which are reduced to 3 and 1 after the remote anomaly symptom recognition is employed. The in-situ density grid model constantly reduces its mispredictions with additionally available training examples. It reaches overall misprediction numbers of 10, 7, and 4 for injection numbers 9, 12, and 15. The final GRU retraining is done after injection number 6. After that, it mispredicts one example at injection numbers 7 and 13.

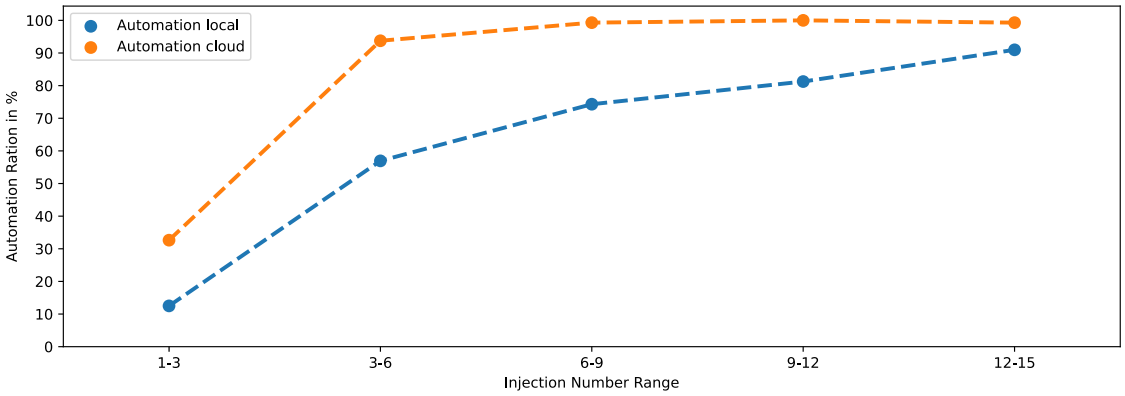


Figure 7.13: Percentage of cases for which an operation can be automatically selected distinguished by pure in-situ and in-situ combined with a AIOps cloud deployment.

Furthermore, we analyze the ability of the two ReCoMe engines to select an operation to resolve anomalies automatically. The automation is possible if the anomaly symptom recognition predicts a correct anomaly type. For the in-situ ReCoMe engine, this means that an anomaly type can be resolved locally. Remote access is not required, and monitoring data must not be sent immediately to a central cloud system. The ReCoMe engine deployed on a dedicated AIOps node in the cloud is a complex model that is considered a fall-back and utilized whenever a local resolving is not possible. Therefore, we interpret the amount of correctly predicted anomaly types as the automation ratio and compare this percentage value for the in-situ AIOps system and the combination of the in-situ and remote AIOps systems. The results are depicted in Figure 7.13. The x-axis shows injection number ranges, whereby the correct and incorrect predictions were aggregated over three injection numbers. The automation ratio is shown on the y-axis and represents the percentage of correctly predicted anomaly types. The blue line shows the results for local automation, which means the ratio of correct predictions done by the in-situ

deployed anomaly symptom recognition. The orange line shows the automation ratio for both the in-situ and remote AIOps anomaly symptom recognition. Whenever the in-situ anomaly symptom recognition yields a misprediction, the GRU anomaly symptom recognition in the cloud is requested to predict the anomaly type. Since the GRU anomaly symptom recognition model has access to training examples from both fog groups, it can be seen that it reaches high automation ratio percentages after injection number 3. For the injection number range 3-6, it yields 93 %, and for the higher rages, it can achieve automation ratios between 99 % and 100 %. The in-situ anomaly symptom recognition has only access to the locally available training examples. This results in a slower automation ratio growth of 15 %, 54 %, 69 %, 78 %, and 90 % for the respective injection number ranges.

The in-situ models and training examples are not shared between the fog groups, e.g., for injection number 2, the density grid model for bono has two available training examples. In contrast to that, the GRU model has access to the training examples of both fog groups. At injection number 2, it can access four examples to train the respective GRU models for the components. Having a central model with access to various remote service instances or compute nodes enables a fast aggregation of training examples for training. In-situ models are responsible for handling local anomalies and aggregate training examples only from the local source. Following this, they improve slower than the central model. However, when enough examples are aggregated, the in-situ AIOps system can handle anomaly symptom recognition and automatic operation selection locally, reducing dependence on remote AIOps systems. We consider the availability of a cloud AIOps system as a reasonable fall-back strategy to overcome the phase where not enough local training data are available.

Chapter 8

Conclusion

This thesis presents an approach for analyzing metric data from components to enable the automatic selection of operations to resolve anomalies. Our anomaly symptom recognition identifies specific patterns and uses them to infer the anomaly type. The inference of previously encountered and resolved anomaly types enables the automatic selection of operations and eventually serves as a support system for human operators. The knowledge base of known anomaly types is constantly extended by enabling the detection of yet unknown anomalies, which are resolved via a human-in-the-loop approach. It represents a gradual transfer of knowledge from human experts into the AIOps system and serves as the concept to resolve the cold start problem. A framework is introduced for the proposed solutions, which we refer to as ReCoMe engine. We implement two methods for anomaly symptom recognition, extend both with an anomaly type inference that enables the recognition of yet unknown anomaly types, and investigate the applicability of those in experimental environments by integrating them into an AIOps platform.

The first anomaly type recognition method aggregates the metric series of anomalous system components while preserving temporal information. The preservation is achieved via density grid transformation and results in grid patterns representing anomaly type-specific symptoms. Grid patterns of historic anomalies are stored in the knowledge base. We define a similarity calculation between grid patterns that tolerates baseline shifts and noise. It is used to compare grid patterns to infer the anomaly type. Evaluation experiments are conducted in a self-hosted cloud computing system with a video on demand service and a virtual implementation of an IMS. The anomaly type recognition is applied to the metric data of the service VMs. We show that density grid pattern comparison achieves an anomaly type recognition accuracy of 0.97 for the IMS and 0.94 for the video on demand service.

The second anomaly type recognition method aims to infer anomaly types by analyzing the sequential dependencies of the anomalous metric series samples. Therefore, an RNN with GRU cells is used as a model. It analyzes a metric series from an anomalous component and predicts the probability that the observed patterns represent one of the known anomaly types. We introduce an augmentation method for metric data, which generates synthetic metric series examples of anomaly types based on a set of historical observations. The augmentation method applies a variation of the residuals and subsamples underrepresented decision regions by applying interpolation between the known examples. These augmentations are used during the training process of the model. The evaluation is done based on metric data from system components that execute heterogeneous workloads. We show that the applied augmentation stabilizes

the training procedure of the model and improves the accuracy of recognizing anomaly types, reaching 0.9.

Both methods are adjusted to recognize yet unknown anomaly types. First, we implement postprocessing to recognize dissimilarities between grid patterns for the density grid pattern model. Given the task of identifying yet unknown anomaly types, it yields an accuracy of 0.97 for the IMS and 0.94 for the video on demand service. For the RNN model with GRU cells, a uniform posterior distribution indicates the inability of the model to assign a known type to the observed symptom patterns. The evaluation to recognize yet unknown anomaly type is done for heterogeneous workload and results in an accuracy of 0.66.

We investigate the applicability of the ReCoMe engine by integrating it into an AIOps platform. Thereby, two deployment concepts are tested, one for a cloud and one for a fog computing environment. The results show that the density grid model can reach a higher anomaly type inference accuracy with less training data. The availability of additional training examples results in a better performance of the RNN model. We show that an AIOps system containing our ReCoMe and utilizing both models can automate the operation selection for up to 99 % of the anomaly cases.

8.1 Limitations

We emphasize the limited availability of examples for each anomaly type used to train the symptom recognition models. Nevertheless, the requirement of multiple recurrences of anomaly types is the main limitation of our work. The evaluations show that the required number of training examples mainly depends on the utilized model and intra-type variations. For a heterogeneous load, the density grid model requires between six and ten examples to reach recognition accuracies above 0.8. The RNN model with GRU cells is prone to overfitting, which we mitigate with metric series augmentation. Thereby, a reasonable sub-sampling of the decision regions by the available anomaly type examples is required. At least ten examples are required for the augmentation to allow a trained model to reach an accuracy of above 0.8.

For the in-situ deployment of the AIOps system containing our ReCoMe engine, we assume the reservation of a fraction of available resources for monitoring data analysis. In our prototype implementation, we utilize Docker for this. It limits our prototype to run on systems where Docker is available. We consider this as a limitation, especially for utilizing computation resources of edge devices.

8.2 Future Work

The methods and systems presented in this thesis open options for future research. We identify three promising directions. First, the utilization of multiple models with the joint objective to ensure an efficient operable system state must be investigated. Our prototype analyses metric data to recognize anomaly symptom patterns. Depending on the system, other data sources such as logs or traces are available. Active probing can be executed on-demand to gather additional information about system components. Furthermore, other methods for auto-scaling or intrusion detection might exist. Therefore, the question of how multiple models can be utilized to jointly ensure an efficient operable state of the system poses a future challenge.

Second, our proposed human-in-the-loop approach hinders the complete autonomy of an AIOps system. The time that is needed to mitigate problems increases whenever humans must be involved. However, especially in critical fields, problems need to be resolved immediately. Different strategies must be investigated to reduce and potentially omit the involvement of human operators. Thereby, the ability to automatically construct remediation and recovery operations for yet unknown anomalies is a central aspect.

Third, the work on AIOps systems should be investigated from a security perspective. AIOps systems aim to introduce increased robustness against failures and anomalies, which is achieved by automation. However, automated execution of operation opens additional attack vectors. An example would be to cause a repeated execution of unnecessary operations by targeting the decision models via adversarial attacks, which eventually results in a denial of service. The identification of such attack vectors in combination with mitigation guidelines and solutions is needed.

Appendix A

Evaluation Result Details

A.1 Symptom Recognition Scores

In the following, we list the evaluation results for anomaly type recognition in the form of accuracy, F1 score, precision, and recall individually for each service component. The load balancer (lb) and video server (vs) belong to the video on demand service, while astaire, bono, chronos, homer, homestead, and sprout belong to Clearwater vIMS. The respective tables show the scores for a specific model and split.

Table A.1: Density grid pattern model scores for the 10 % split.

	lb	vs	astaire	bono	chronos	homer	homestead	sprout
Accuracy	0.28	0.29	0.36	0.42	0.35	0.38	0.34	0.36
F1	0.37	0.37	0.45	0.49	0.45	0.48	0.45	0.45
Precision	0.74	0.67	0.78	0.77	0.79	0.80	0.81	0.75
Recall	0.28	0.29	0.36	0.42	0.35	0.38	0.34	0.36

Table A.2: Density grid pattern model scores for the 30 % split.

	lb	vs	astaire	bono	chronos	homer	homestead	sprout
Accuracy	0.74	0.68	0.74	0.78	0.80	0.75	0.73	0.79
F1	0.78	0.71	0.78	0.82	0.84	0.81	0.80	0.84
Precision	0.92	0.85	0.89	0.93	0.95	0.96	0.95	0.95
Recall	0.74	0.68	0.74	0.78	0.80	0.75	0.73	0.79

Table A.3: Density grid pattern model scores for the 50 % split.

	lb	vs	astaire	bono	chronos	homer	homestead	sprout
Accuracy	0.88	0.82	0.83	0.93	0.86	0.90	0.90	0.91
F1	0.89	0.82	0.85	0.93	0.87	0.91	0.91	0.92
Precision	0.93	0.86	0.90	0.95	0.91	0.94	0.94	0.94
Recall	0.88	0.82	0.83	0.93	0.86	0.90	0.90	0.91

Table A.4: Density grid pattern model scores for the 70 % split.

	lb	vs	astaire	bono	chronos	homer	homestead	sprout
Accuracy	0.96	0.94	0.97	0.97	0.98	0.97	0.97	0.97
F1	0.96	0.94	0.97	0.98	0.98	0.97	0.97	0.97
Precision	0.97	0.96	0.98	0.98	0.98	0.98	0.98	0.98
Recall	0.96	0.94	0.97	0.97	0.98	0.97	0.97	0.97

Table A.5: Euclidean model scores for the 10 % split.

	lb	vs	astaire	bono	chronos	homer	homestead	sprout
Accuracy	0.25	0.24	0.18	0.15	0.15	0.31	0.33	0.20
F1	0.27	0.26	0.19	0.17	0.17	0.32	0.34	0.21
Precision	0.31	0.30	0.23	0.20	0.21	0.36	0.38	0.25
Recall	0.25	0.24	0.18	0.15	0.15	0.31	0.33	0.20

Table A.6: Euclidean model scores for the 30 % split.

	lb	vs	astaire	bono	chronos	homer	homestead	sprout
Accuracy	0.41	0.47	0.40	0.43	0.49	0.47	0.44	0.46
F1	0.41	0.46	0.39	0.43	0.48	0.47	0.44	0.47
Precision	0.44	0.50	0.43	0.47	0.52	0.51	0.47	0.50
Recall	0.41	0.47	0.40	0.43	0.49	0.47	0.44	0.46

Table A.7: Euclidean model scores for the 50 % split.

	lb	vs	astaire	bono	chronos	homer	homestead	sprout
Accuracy	0.51	0.58	0.55	0.64	0.61	0.59	0.58	0.53
F1	0.49	0.57	0.55	0.64	0.60	0.57	0.56	0.50
Precision	0.53	0.61	0.58	0.67	0.63	0.59	0.58	0.52
Recall	0.51	0.58	0.55	0.64	0.61	0.59	0.58	0.53

Table A.8: Euclidean model scores for the 70 % split.

	lb	vs	astaire	bono	chronos	homer	homestead	sprout
Accuracy	0.61	0.61	0.57	0.58	0.78	0.71	0.70	0.47
F1	0.58	0.58	0.53	0.54	0.78	0.69	0.68	0.44
Precision	0.59	0.60	0.53	0.55	0.81	0.73	0.71	0.45
Recall	0.61	0.61	0.57	0.58	0.78	0.71	0.70	0.47

Table A.9: DTW model scores for the 10 % split.

	lb	vs	astaire	bono	chronos	homer	homestead	sprout
Accuracy	0.25	0.23	0.32	0.34	0.33	0.32	0.35	0.33
F1	0.25	0.24	0.32	0.34	0.32	0.31	0.34	0.32
Precision	0.27	0.27	0.35	0.36	0.33	0.33	0.35	0.34
Recall	0.25	0.23	0.32	0.34	0.33	0.32	0.35	0.33

Table A.10: DTW model scores for the 30 % split.

	lb	vs	astaire	bono	chronos	homer	homestead	sprout
Accuracy	0.50	0.52	0.46	0.52	0.58	0.53	0.55	0.49
F1	0.50	0.51	0.45	0.51	0.56	0.53	0.55	0.49
Precision	0.54	0.54	0.48	0.54	0.59	0.56	0.58	0.53
Recall	0.50	0.52	0.46	0.52	0.58	0.53	0.55	0.49

Table A.11: DTW model scores for the 50 % split.

	lb	vs	astaire	bono	chronos	homer	homestead	sprout
Accuracy	0.72	0.74	0.78	0.84	0.81	0.83	0.88	0.83
F1	0.72	0.74	0.77	0.84	0.81	0.82	0.88	0.82
Precision	0.75	0.78	0.80	0.87	0.84	0.86	0.90	0.85
Recall	0.72	0.74	0.78	0.84	0.81	0.83	0.88	0.83

Table A.12: DTW model scores for the 70 % split.

	lb	vs	astaire	bono	chronos	homer	homestead	sprout
Accuracy	0.89	0.87	0.96	0.91	0.92	0.97	0.95	0.91
F1	0.89	0.87	0.97	0.91	0.91	0.97	0.95	0.91
Precision	0.91	0.89	0.97	0.93	0.93	0.97	0.96	0.93
Recall	0.89	0.87	0.96	0.91	0.92	0.97	0.95	0.91

A.2 Unknown Anomaly Type Recognition Accuracy

We list the evaluation results for unknown anomaly type recognition in the form of accuracy individually for each service component. The load balancer (lb) and video server (vs) belong to the video on demand service, while astaire, bono, chronos, homer, homestead, and sprout belong to Clearwater vIMS. The respective tables show the scores for a specific model.

Table A.13: Unknown anomaly type recognition accuracy scores for the density grid pattern model.

Traindata used in %	lb	vs	astaire	bono	chronos	homer	homestead	sprout
10	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
30	0.99	0.97	0.98	0.99	1.00	1.00	1.00	0.99
50	0.94	0.93	0.94	0.94	1.00	1.00	1.00	0.93
70	0.92	0.88	0.93	0.89	0.93	0.93	0.93	0.94

Table A.14: Unknown anomaly type recognition accuracy scores for the Euclidean distance model.

Traindata used in %	lb	vs	astaire	bono	chronos	homer	homestead	sprout
10	0.74	0.62	0.76	0.65	0.79	0.73	0.71	0.74
30	0.53	0.55	0.59	0.63	0.70	0.78	0.77	0.53
50	0.46	0.44	0.64	0.67	0.74	0.43	0.44	0.68
70	0.42	0.40	0.51	0.64	0.43	0.53	0.43	0.50

Table A.15: Unknown anomaly type recognition accuracy scores for the DTW model.

Traindata used in %	lb	vs	astaire	bono	chronos	homer	homestead	sprout
10	0.55	0.54	0.52	0.52	0.65	0.58	0.80	0.47
30	0.38	0.38	0.44	0.44	0.43	0.49	0.48	0.43
50	0.37	0.36	0.38	0.41	0.38	0.46	0.41	0.39
70	0.37	0.27	0.34	0.36	0.39	0.35	0.38	0.30

Bibliography

[1] A. Avizienis and J. -. Laprie and B. Randell and C. Landwehr. “Basic concepts and taxonomy of dependable and secure computing”. In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33.

[2] Aceto, Giuseppe and Botta, Alessio and De Donato, Walter and Pescape, Antonio. “Cloud monitoring: A survey”. In: *Computer Networks* 57.9 (2013), pp. 2093–2115.

[3] Acker, Alexander and Schmidt, Florian and Gulenko, Anton and Kao, Odej. “Online Density Grid Pattern Analysis to Classify Anomalies in Cloud and NFV Systems”. In: *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2018, pp. 290–295.

[4] Acker, Alexander and Wittkopp, Thorsten and Nedelkoski, Sasho and Bogatinovski, Jasmin and Kao, Odej. “Superiority of Simplicity: A Lightweight Model for Network Device Workload Prediction”. In: *2020 15th Conference on Computer Science and Information Systems (FedCSIS)*. IEEE. 2020, pp. 7–10.

[5] Ahmed, Mohiuddin and Mahmood, Abdun Naser and Hu, Jiankun. “A survey of network anomaly detection techniques”. In: *Journal of Network and Computer Applications* 60 (2016), pp. 19–31.

[6] Alhosban, Amal and Hashmi, Khayyam and Malik, Zaki and Medjahed, Brahim. “Self-healing framework for cloud-based services”. In: *2013 ACS International Conference on Computer Systems and Applications (AICCSA)*. IEEE. 2013, pp. 1–7.

[7] Amazon.com, Inc. *Press release: Amazon.com Announces Financial Results and CEO Transition*. Tech. rep. `https://press.aboutamazon.com/news-releases/news-release-details/amazoncom-announces-financial-results-and-ceo-transition` (last access 22 April 2021). 2021.

[8] Anderson, James P. *Computer security technology planning study*. Tech. rep. ANDERSON (JAMES P) and CO FORT WASHINGTON PA FORT WASHINGTON, 1972.

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

[10] Athreya, Arjun P and DeBruhl, Bruce and Tague, Patrick. “Designing for self-configuration and self-adaptation in the Internet of Things”. In: *9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*. IEEE. 2013, pp. 585–592.

[11] Aupy, Guillaume and Robert, Yves and Vivien, Frederic and Zaidouni, Dounia. “Checkpointing strategies with prediction windows”. In: *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*. IEEE. 2013, pp. 1–10.

[12] Avizienis, Algirdas. “The methodology of n-version programming”. In: *Software fault tolerance* 3 (1995), pp. 23–46.

[13] Avizienis, Algirdas and Kelly, John PJ. “Fault tolerance by design diversity: Concepts and experiments”. In: *Computer* 17.08 (1984), pp. 67–80.

[14] Avizienis, Algirdas and Laprie, J-C. “Dependable computing: From concepts to design diversity”. In: *Proceedings of the IEEE* 74.5 (1986), pp. 629–638.

[15] Avizienis, Algirdas and Laprie, Jean-Claude and Randell, Brian and Landwehr, Carl. “Basic concepts and taxonomy of dependable and secure computing”. In: *IEEE transactions on dependable and secure computing* 1.1 (2004), pp. 11–33.

[16] Aviziens, Algirdas. “Fault-tolerant systems”. In: *IEEE Transactions on Computers* 100.12 (1976), pp. 1304–1312.

[17] Azaiez, Meriem and Chainbi, Walid. “A multi-agent system architecture for self-healing cloud infrastructure”. In: *Proceedings of the International Conference on Internet of things and Cloud Computing*. ACM. 2016, p. 7.

[18] Bakhshi, Zeinab and Rodriguez-Navas, Guillermo and Hansson, Hans. “Dependable fog computing: A systematic literature review”. In: *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2019, pp. 395–403.

[19] Ball, Michael O and Colbourn, Charles J and Provan, J Scott. “Network reliability”. In: *Handbooks in operations research and management science* 7 (1995), pp. 673–762.

[20] Bari, Md Faizul and Boutaba, Raouf and Esteves, Rafael and Granville, Lisandro Zambenedetti and Podlesny, Maxim and Rabbani, Md Golam and Zhang, Qi and Zhani, Mohamed Faten. “Data center network virtualization: A survey”. In: *IEEE communications surveys & tutorials* 15.2 (2012), pp. 909–928.

[21] Barlow, Richard E and HUNTER, LARRY C. *Mathematical models for system reliability*. Tech. rep. SYLVANIA ELECTRONIC SYSTEMS-WEST MOUNTAIN VIEW CALIF ELECTRONIC DEFENSE LABS, 1959.

[22] Barroso, Luiz Andre and Hoelzle, Urs. “The datacenter as a computer: An introduction to the design of warehouse-scale machines”. In: *Synthesis lectures on computer architecture* 4.1 (2009), pp. 1–108.

[23] Becker, Soeren and Schmidt, Florian and Gulenko, Anton and Acker, Alexander and Kao, Odej. “Towards AIOps in Edge Computing Environments”. In: *2020 IEEE International Conference on Big Data (Big Data)*. IEEE. 2020, pp. 3470–3475.

[24] Bennett, Keith and Layzell, Paul and Budgen, David and Brereton, Pearl and Macaulay, Linda and Munro, Malcolm. “Service-based software: the future for flexible software”. In: *Proceedings Seventh Asia-Pacific Software Engeering Conference. APSEC 2000*. IEEE. 2000, pp. 214–221.

[25] Bermbach, David and Pallas, Frank and Pérez, David García and Plebani, Pierluigi and Anderson, Maya and Kat, Ronen and Tai, Stefan. "A research perspective on fog computing". In: *International Conference on Service-Oriented Computing*. Springer. 2017, pp. 198–210.

[26] Berndt, Donald J and Clifford, James. "Using dynamic time warping to find patterns in time series." In: *KDD workshop*. Vol. 10. 16. Seattle, WA, USA: 1994, pp. 359–370.

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

[28] Bishop, Christopher M. *Pattern recognition and machine learning*. springer, 2006.

[29] Blank-Edelman, David N. *Seeking SRE: Conversations About Running Production Systems at Scale*. " O'Reilly Media, Inc.", 2018.

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

[31] Bogatinovski, Jasmin and Nedelkoski, Sasho. "Multi-source anomaly detection in distributed it systems". In: *International Conference on Service-Oriented Computing*. Springer. 2020, pp. 201–213.

[32] Borg, Anita and Baumbach, Jim and Glazer, Sam. "A message system supporting fault tolerance". In: *ACM SIGOPS Operating Systems Review* 17.5 (1983), pp. 90–99.

[33] Borg, Anita and Blau, Wolfgang and Graetsch, Wolfgang and Herrmann, Ferdinand and Oberle, Wolfgang. "Fault tolerance under UNIX". In: *ACM Transactions on Computer Systems (TOCS)* 7.1 (1989), pp. 1–24.

[34] Bosilca, George and Delmas, Remi and Dongarra, Jack and Langou, Julien. "Algorithm-based fault tolerance applied to high performance computing". In: *Journal of Parallel and Distributed Computing* 69.4 (2009), pp. 410–416.

[35] Bouguerra, Mohamed Slim and Gainaru, Ana and Gomez, Leonardo Bautista and Cappello, Franck and Matsuoka, Satoshi and Maruyam, Naoya. "Improving the computing efficiency of hpc systems using a combination of proactive and preventive checkpointing". In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE. 2013, pp. 501–512.

[36] Box, George EP and Cox, David R. "An analysis of transformations". In: *Journal of the Royal Statistical Society: Series B (Methodological)* 26.2 (1964), pp. 211–243.

[37] Bridle, John S. "Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition". In: *Neurocomputing*. Springer, 1990, pp. 227–236.

[38] Briscoe, Bob and Brunstrom, Anna and Petlund, Andreas and Hayes, David and Ros, David and Tsang, Jyh and Gjessing, Stein and Fairhurst, Gorry and Griwodz, Carsten and Welzl, Michael. "Reducing internet latency: A survey of techniques and their merits". In: *IEEE Communications Surveys & Tutorials* 18.3 (2014), pp. 2149–2196.

[39] Bruneo, Dario and Distefano, Salvatore and Longo, Francesco and Puliafito, Antonio and Scarpa, Marco. "Workload-based software rejuvenation in cloud systems". In: *IEEE Transactions on Computers* 62.6 (2013), pp. 1072–1085.

[40] Cai, Xingyu and Xu, Tingyang and Yi, Jinfeng and Huang, Junzhou and Rajasekaran, Sanguthevar. "DTWNet: a dynamic time warping network". In: *Advances in neural information processing systems* 32 (2019).

[41] Calheiros, Rodrigo N and Masoumi, Enayat and Ranjan, Rajiv and Buyya, Rajkumar. "Workload prediction using ARIMA model and its impact on cloud applications' QoS". In: *IEEE transactions on cloud computing* 3.4 (2014), pp. 449–458.

[42] Calheiros, Rodrigo N and Ramamohanarao, Kotagiri and Buyya, Rajkumar and Leckie, Christopher and Versteeg, Steve. "On the effectiveness of isolation-based anomaly detection in cloud data centers". In: *Concurrency and Computation: Practice and Experience* 29.18 (2017), e4169.

[43] Caliva, Francesco and De Ribeiro, Fabio Sousa and Mylonakis, Antonios and Demazi'ere, Christophe and Vinai, Paolo and Leontidis, Georgios and Kollias, Stefanos. "A deep learning approach to anomaly detection in nuclear reactors". In: *2018 International joint conference on neural networks (IJCNN)*. IEEE. 2018, pp. 1–8.

[44] Camarillo, Gonzalo and Garcia-Martin, Miguel-Angel. *The 3G IP multimedia subsystem (IMS): merging the Internet and the cellular worlds*. John Wiley & Sons, 2007.

[45] Candea, George and Kawamoto, Shinichi and Fujiki, Yuichi and Friedman, Greg and Fox, Armando. "Microreboot - A Technique for Cheap Recovery". In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*. OSDI 04. USENIX Association, 2004, p. 3.

[46] Castelli, Vittorio and Harper, Richard E and Heidelberger, Philip and Hunter, Steven W and Trivedi, Kishor S and Vaidyanathan, Kalyanaraman and Zeggert, William P. "Proactive management of software aging". In: *IBM Journal of Research and Development* 45.2 (2001), pp. 311–332.

[47] Catal, Cagatay and Diri, Banu. "A systematic review of software fault prediction studies". In: *Expert systems with applications* 36.4 (2009), pp. 7346–7354.

[48] Catchpoint. *2020 SRE Report - The Distributed SRE*. `https://www.catchpoint.com/sre-report`. (last access 26 April 2021). 2020.

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

[50] Chang, Victor. "Towards a big data system disaster recovery in a private cloud". In: *Ad Hoc Networks* 35 (2015), pp. 65–82.

[51] Chatzigiannakis, Ioannis and Hasemann, Henning and Karnstedt, Marcel and Kleine, Oliver and Kroeller, Alexander and Leggieri, Myriam and Pfisterer, Dennis and Roemer, Kay and Truong, Cuong. "True self-configuration for the IoT". In: *2012 3rd IEEE International Conference on the Internet of Things*. IEEE. 2012, pp. 9–15.

[52] Chen, Liming and Avizienis, Algirdas. "N-version programming: A fault-tolerance approach to reliability of software operation". In: *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*. Vol. 1. 1978, pp. 3–9.

[53] Chen, Yixin and Tu, Li. "Density-based clustering for real-time stream data". In: *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2007, pp. 133–142.

[54] Cheng, Min and Li, Qing and Lv, Jianming and Liu, Wenyin and Wang, Jianping. "Multi-Scale LSTM Model for BGP Anomaly Classification". In: *IEEE Transactions on Services Computing* (2018).

[55] Cheng, Shang-Wen and Huang, An-Cheng and Garlan, David and Schmerl, Bradley and Steenkiste, Peter. "Rainbow: Architecture-based self-adaptation with reusable infrastructure". In: *International Conference on Autonomic Computing, 2004. Proceedings.* IEEE. 2004, pp. 276–277.

[56] Chiosi, Margaret and Clarke, Don and Willis, Peter and Reid, Andy and Feger, James and Bugenhagen, Michael and Khan, Waqar and Fargano, Michael and Cui, Chunfeng and Deng, Hui and others. "Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action". In: *SDN and OpenFlow world congress*. Vol. 48. sn. 2012, pp. 1–16.

[57] Cho, Kyunghyun and Van Merriënboer, Bart and Gulcehre, Caglar and Bahdanau, Dzmitry and Bougares, Fethi and Schwenk, Holger and Bengio, Yoshua. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: (2014), pp. 1724–1734.

[58] Chung, Junyoung and Gulcehre, Caglar and Cho, Kyunghyun and Bengio, Yoshua. "Empirical evaluation of gated recurrent neural networks on sequence modeling". In: *NIPS 2014 Workshop on Deep Learning, December 2014*. 2014.

[59] Cohen, Ira and Zhang, Steve and Goldszmidt, Moises and Symons, Julie and Kelly, Terence and Fox, Armando. "Capturing, indexing, clustering, and retrieving system history". In: *ACM SIGOPS Operating Systems Review* 39.5 (2005), pp. 105–118.

[60] Coit, David W. "Cold-standby redundancy optimization for nonrepairable systems". In: *Iie Transactions* 33.6 (2001), pp. 471–478.

[61] Cotroneo, Domenico and De Simone, Luigi and Iannillo, Antonio Ken and Lanzaro, Anna and Natella, Roberto and Fan, Jiang and Ping, Wang. "Network function virtualization: Challenges and directions for reliability assurance". In: *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE. 2014, pp. 37–42.

[62] Cotroneo, Domenico and De Simone, Luigi and Natella, Roberto. "Dependability certification guidelines for NFVIS through fault injection". In: *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE. 2018, pp. 321–328.

[63] Cotroneo, Domenico and De Simone, Luigi and Natella, Roberto. "NFV-bench: A dependability benchmark for network function virtualization systems". In: *IEEE Transactions on Network and Service Management* 14.4 (2017), pp. 934–948.

[64] Cotroneo, Domenico and Natella, Roberto and Rosiello, Stefano. "A fault correlation approach to detect performance anomalies in virtual network function chains". In: *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2017, pp. 90–100.

[65] Cotroneo, Domenico and Pietrantuono, Roberto and Russo, Stefano and Trivedi, Kishor. “How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation”. In: *Journal of Systems and Software* 113 (2016), pp. 27–43.

[66] Cristian, Flavin. “Understanding fault-tolerant distributed systems”. In: *Communications of the ACM* 34.2 (1991), pp. 56–78.

[67] Cristian, Flaviu and Dancey, Bob and Dehn, Jon. “Fault-tolerance in the advanced automation system”. In: *Proceedings of the 4th Workshop on ACM SIGOPS European Workshop*. 1990, pp. 6–17.

[68] Dai, Yuan-Shun and Yang, Bo and Dongarra, Jack and Zhang, Gewei. “Cloud service reliability: Modeling and analysis”. In: *15th IEEE Pacific Rim International Symposium on Dependable Computing*. Citeseer. 2009, pp. 1–17.

[69] Dang, Yingnong and Lin, Qingwei and Huang, Peng. “AIOps: real-world challenges and research innovations”. In: *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*. IEEE Press. 2019, pp. 4–5.

[70] Dashofy, Eric M and Van der Hoek, André and Taylor, Richard N. “Towards architecture-based self-healing systems”. In: *Proceedings of the first workshop on Self-healing systems*. 2002, pp. 21–26.

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

[72] Ding, Hui and Trajcevski, Goce and Scheuermann, Peter and Wang, Xiaoyue and Keogh, Eamonn. “Querying and mining of time series data: experimental comparison of representations and distance measures”. In: *Proceedings of the VLDB Endowment* 1.2 (2008), pp. 1542–1552.

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

[74] Du, Qingfeng and He, Yu and Xie, Tiandi and Yin, Kanglin and Qiu, Juan. “An approach of collecting performance anomaly dataset for NFV Infrastructure”. In: *International Conference on Algorithms and Architectures for Parallel Processing*. Springer. 2018, pp. 59–71.

[75] Eckart, Ben and Chen, Xin and He, Xubin and Scott, Stephen L. “Failure prediction models for proactive fault tolerance within storage systems”. In: *2008 IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems*. IEEE. 2008, pp. 1–8.

[76] Egwutuoha, Ifeanyi P and Chen, Shiping and Levy, David and Selic, Bran and Calvo, Rafael. “A proactive fault tolerance approach to High Performance Computing (HPC) in the cloud”. In: *2012 Second International Conference on Cloud and Green Computing*. IEEE. 2012, pp. 268–273.

[77] Epstein, Benjamin and Sobel, Milton. “Life testing”. In: *Journal of the American Statistical Association* 48.263 (1953), pp. 486–502.

[78] Etro, Federico. “The economic impact of cloud computing on business creation, employment and output in Europe”. In: *Review of Business and Economics* 54.2 (2009), pp. 179–208.

[79] Evensen, Kristian and Kupka, Tomas and Kaspar, Dominik and Halvorsen, Paal and Griwodz, Carsten. “Quality-adaptive scheduling for live streaming over multiple access networks”. In: *Proceedings of the 20th international workshop on Network and operating systems support for digital audio and video*. 2010, pp. 21–26.

[80] Fawaz, Hassan Ismail and Lucas, Benjamin and Forestier, Germain and Pelletier, Charlotte and Schmidt, Daniel F and Weber, Jonathan and Webb, Geoffrey I and Idoumghar, Lhassane and Muller, Pierre-Alain and Petitjean, Franccois. “Inceptiontime: Finding alexnet for time series classification”. In: *Data Mining and Knowledge Discovery* 34.6 (2020), pp. 1936–1962.

[81] Fayyaz, Muhammad and Vladimirova, Tanya. “Survey and future directions of fault-tolerant distributed computing on board spacecraft”. In: *Advances in Space Research* 58.11 (2016), pp. 2352–2375.

[82] Feldmann, Anja and Gasser, Oliver and Lichtblau, Franziska and Pujol, Eric and Poese, Igmar and Dietzel, Christoph and Wagner, Daniel and Wichtlhuber, Matthias and Tapiador, Juan and Vallina-Rodriguez, Narseo and others. “Implications of the COVID-19 Pandemic on the Internet Traffic”. In: *Broadband Coverage in Germany; 15th ITG-Symposium*. VDE. 2021, pp. 1–5.

[83] Fiondella, Lance and Gokhale, Swapna S and Mendiratta, Veena B. “Cloud incident data: An empirical analysis”. In: *2013 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2013, pp. 241–249.

[84] Forecast, GMDT. “Cisco visual networking index: global mobile data traffic forecast update, 2017–2022”. In: *Update* 2017 (2019), p. 2022.

[85] Fox, Armando and Griffith, Rean and Joseph, Anthony and Katz, Randy and Konwinski, Andrew and Lee, Gunho and Patterson, David and Rabkin, Ariel and Stoica, Ion and others. “Above the clouds: A berkeley view of cloud computing”. In: *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS* 28.13 (2009), p. 2009.

[86] Fujimaki, Ryohei and Yairi, Takehisa and Machida, Kazuo. “An approach to spacecraft anomaly detection problem using kernel feature space”. In: *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. 2005, pp. 401–410.

[87] Ganesh, Amal and Sandhya, M and Shankar, Sharmila. “A study on fault tolerance methods in cloud computing”. In: *2014 IEEE International Advance Computing Conference (IACC)*. IEEE. 2014, pp. 844–849.

[88] Gao, Jing and Li, Jianzhong and Zhang, Zhaogong and Tan, Pang-Ning. “An incremental data stream clustering algorithm based on dense units detection”. In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2005, pp. 420–425.

[89] Garg, Sahil and Kaur, Kuljeet and Batra, Shalini and Aujla, Gagangeet Singh and Morgan, Graham and Kumar, Neeraj and Zomaya, Albert Y and Ranjan, Rajiv. "En-ABC: An ensemble artificial bee colony based anomaly detection scheme for cloud environment". In: *Journal of Parallel and Distributed Computing* 135 (2020), pp. 219–233.

[90] Garg, Sahil and Kaur, Kuljeet and Kumar, Neeraj and Kaddoum, Georges and Zomaya, Albert Y and Ranjan, Rajiv. "A hybrid deep learning-based model for anomaly detection in cloud datacenter networks". In: *IEEE Transactions on Network and Service Management* 16.3 (2019), pp. 924–935.

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

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

[93] Garraghan, Peter and Townend, Paul and Xu, Jie. "An empirical failure-analysis of a large-scale cloud computing environment". In: *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*. IEEE. 2014, pp. 113–120.

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

[95] Gill, Sukhpal Singh and Chana, Inderveer and Singh, Maninder and Buyya, Rajkumar. "RADAR: Self-Configuring and Self-Healing in Resource Management for Enhancing Quality of Cloud Services". In: *Concurrency and Computation: Practice and Experience (CCPE)* (2018).

[96] Gill, Sukhpal Singh and Chana, Inderveer and Singh, Maninder and Buyya, Rajkumar. "RADAR: Self-configuring and self-healing in resource management for enhancing quality of cloud services". In: *Concurrency and Computation: Practice and Experience* 31.1 (2019), e4834.

[97] Google Cloud Architecture Center. *MLOps: Continuous delivery and automation pipelines in machine learning*. `https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning`. (last access 21 May 2021). 2021.

[98] Grambow, Martin and Hasenburg, Jonathan and Bermbach, David. "Public video surveillance: Using the fog to increase privacy". In: *Proceedings of the 5th Workshop on Middleware and Applications for the Internet of Things*. 2018, pp. 11–14.

[99] Gray, Jim. "A census of Tandem system availability between 1985 and 1990". In: *IEEE Transactions on reliability* 39.4 (1990), pp. 409–418.

[100] Gray, Jim and Siewiorek, Daniel P. "High-availability computer systems". In: *Computer* 24.9 (1991), pp. 39–48.

[101] Gregg, Brendan. *Systems performance: enterprise and the cloud*. Pearson Education, 2014.

[102] Griwodz, Carsten and Bär, Michael and Wolf, Lars C. "Long-term movie popularity models in video-on-demand systems: or the life of an on-demand movie". In: *Proceedings of the fifth ACM international conference on Multimedia*. 1997, pp. 349–357.

[103] Grubbs, Frank E. "Procedures for detecting outlying observations in samples". In: *Technometrics* 11.1 (1969), pp. 1–21.

[104] Guerraoui, Rachid and Schiper, André. "Software-based replication for fault tolerance". In: *Computer* 30.4 (1997), pp. 68–74.

[105] Gulenko, Anton and Acker, Alexander and Kao, Odej and Liu, Feng. "AI-Governance and Levels of Automation for AIOps-supported System Administration". In: *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. IEEE. 2020, pp. 1–6.

[106] Gulenko, Anton and Acker, Alexander and Schmidt, Florian and Becker, Soeren and Kao, Odej. "Bitflow: An In Situ Stream Processing Framework". In: *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. IEEE. 2020, pp. 182–187.

[107] Gulenko, Anton and Kao, Odej and Schmidt, Florian. "Anomaly Detection and Levels of Automation for AI-Supported System Administration". In: *Annual International Symposium on Information Management and Big Data*. Springer. 2019, pp. 1–7.

[108] Gulenko, Anton and Schmidt, Florian and Acker, Alexander and Wallschlaeger, Marcel and Kao, Odej and Liu, Feng. "Detecting anomalous behavior of black-box services modeled with distance-based online clustering". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE. 2018, pp. 912–915.

[109] Gulenko, Anton and Wallschlaeger, Marcel and Schmidt, Florian and Kao, Odej and Liu, Feng. "A system architecture for real-time anomaly detection in large-scale nfv systems". In: *Procedia Computer Science* 94 (2016), pp. 491–496.

[110] Gulenko, Anton and Wallschlaeger, Marcel and Schmidt, Florian and Kao, Odej and Liu, Feng. "Evaluating machine learning algorithms for anomaly detection in clouds". In: *2016 IEEE International Conference on Big Data (Big Data)*. IEEE. 2016, pp. 2716–2721.

[111] Gunawi, Haryadi S and Hao, Mingzhe and Leesatapornwongsa, Tanakorn and Patana-anake, Tiratat and Do, Thanh and Adityatama, Jeffry and Eliazar, Kurnia J and Laksono, Agung and Lukman, Jeffrey F and Martin, Vincentius and others. "What bugs live in the cloud? a study of 3000+ issues in cloud systems". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2014, pp. 1–14.

[112] Hamilton, James Douglas. *Time series analysis*. Princeton university press, 1994.

[113] Han, Bo and Gopalakrishnan, Vijay and Ji, Lusheng and Lee, Seungjoon. "Network function virtualization: Challenges and opportunities for innovations". In: *IEEE Communications Magazine* 53.2 (2015), pp. 90–97.

[114] Haselboeck, Stefan and Weinreich, Rainer. "Decision guidance models for microservice monitoring". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE. 2017, pp. 54–61.

[115] Hassan, Ahmed E and Holt, Richard C. "The top ten list: Dynamic fault prediction". In: *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE. 2005, pp. 263–272.

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

[117] Hochreiter, Sepp. "The vanishing gradient problem during learning recurrent neural nets and problem solutions". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.02 (1998), pp. 107–116.

[118] Hochreiter, Sepp and Schmidhuber, Jürgen. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[119] Horn, Paul. "Autonomic computing: IBM's perspective on the state of information technology". In: (2001).

[120] Hosford, John E. "Measures of dependability". In: *Operations Research* 8.1 (1960), pp. 53–64.

[121] Hu, Honglin and Zhang, Jian and Zheng, Xiaoying and Yang, Yang and Wu, Ping. "Self-configuration and self-optimization for LTE networks". In: *IEEE Communications Magazine* 48.2 (2010), pp. 94–100.

[122] Huang, Kuang-Hua and Abraham, Jacob A. "Algorithm-based fault tolerance for matrix operations". In: *IEEE transactions on computers* 100.6 (1984), pp. 518–528.

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

[124] Hung, Mark. "Leading the iot, gartner insights on how to lead in a connected world". In: *Gartner Research* 1 (2017), pp. 1–5.

[125] Hur, Junbeom and Kang, Kyungtae. "Dependable and secure computing in medical information systems". In: *Computer Communications* 36.1 (2012), pp. 20–28.

[126] IBM. "An architectural blueprint for autonomic computing". In: *IBM White Paper* 31.2006 (2006), pp. 1–6.

[127] Iglesias, Félix and Kastner, Wolfgang. "Analysis of similarity measures in times series clustering for the discovery of building energy patterns". In: *Energies* 6.2 (2013), pp. 579–597.

[128] Ikeuchi, Hiroki and Watanabe, Akio and Hirao, Tsutomu and Morishita, Makoto and Nishino, Masaaki and Matsuo, Yoichi and Watanabe, Keishiro. "Recovery command generation towards automatic recovery in ict systems by seq2seq learning". In: *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE. 2020, pp. 1–6.

[129] Ioffe, Sergey and Szegedy, Christian. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.

[130] Iorga, Michaela and Goren, Nedim and Feldman, Larry and Barton, Robert and Martin, Michael and Mahmoudi, Charif. *Fog Computing Conceptual Model*. Tech. rep. `https://doi.org/10.6028/NIST.SP.500-325` (last access 21 April 2021). National Institute of Standards and Technology, 2018.

[131] Islam, Tariqul and Manivannan, Dakshnamoorthy. “Predicting application failure in cloud: A machine learning approach”. In: *2017 IEEE International Conference on Cognitive Computing (ICCC)*. IEEE. 2017, pp. 24–31.

[132] Javadi, Bahman and Abawajy, Jemal and Buyya, Rajkumar. “Failure-aware resource provisioning for hybrid cloud infrastructure”. In: *Journal of parallel and distributed computing* 72.10 (2012), pp. 1318–1331.

[133] Javed, Asad and Heljanko, Keijo and Buda, Andrea and Fraemling, Kary. “Cefiot: A fault-tolerant iot architecture for edge and cloud”. In: *2018 IEEE 4th world forum on internet of things (WF-IoT)*. IEEE. 2018, pp. 813–818.

[134] Jia, Chen and Tan, ChengYu and Yong, Ai. “A grid and density-based clustering algorithm for processing data stream”. In: *2008 Second International Conference on Genetic and Evolutionary Computing*. IEEE. 2008, pp. 517–521.

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

[136] Kaitovic, Igor and Malek, Miroslaw. “Impact of Failure Prediction on Availability: Modeling and Comparative Analysis of Predictive and Reactive Methods”. In: *IEEE Transactions on Dependable and Secure Computing* 17.3 (2018), pp. 493–505.

[137] Kajó, Márton and Nováczki, Szabolcs. “A genetic feature selection algorithm for anomaly classification in mobile networks”. In: *19th International ICIN conference-Innovations in Clouds, Internet and Networks*. 2016.

[138] Kephart, Jeffrey O and Chess, David M. “The vision of autonomic computing”. In: *Computer* 36.1 (2003), pp. 41–50.

[139] Kotliar, A and Kotliar, V. “Event-driven automation and chatops on IHEP computing cluster”. In: *CEUR Workshop Proceedings*. 2018, pp. 558–562.

[140] Lan, Zhiling and Li, Yawei. “Adaptive fault management of parallel applications for high-performance computing”. In: *IEEE Transactions on Computers* 57.12 (2008), pp. 1647–1660.

[141] Laprie, Jean-Claude. “Dependable computing and fault-tolerance”. In: *Digest of Papers FTCS-15* (1985), pp. 2–11.

[142] Laranjeira, Luiz A and Malek, Miroslaw and Jenevein, Roy. “On tolerating faults in naturally redundant algorithms”. In: *Proceedings Tenth Symposium on Reliable Distributed Systems*. IEEE Computer Society. 1991, pp. 118–119.

[143] Lavriv, Orest and Klymash, Mykhailo and Grynkevych, Ganna and Tkachenko, Olga and Vasylenko, Volodymyr. “Method of cloud system disaster recovery based on" Infrastructure as a code" concept”. In: *2018 14th International Conference on Advanced Trends in Radioelecrtronics, Telecommunications and Computer Engineering (TCSET)*. IEEE. 2018, pp. 1139–1142.

[144] Lee, Peter Alan and Anderson, Thomas. “Fault tolerance”. In: *Fault Tolerance*. Springer, 1990, pp. 51–77.

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

[146] Ligus, Slawek. *Effective monitoring and alerting*. " O’Reilly Media, Inc.", 2012.

[147] Linnainmaa, Seppo. “The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors”. In: *Master’s Thesis (in Finnish), Univ. Helsinki* (1970), pp. 6–7.

[148] Liu, Ping and Xu, Haowen and Ouyang, Qianyu and Jiao, Rui and Chen, Zhekang and Zhang, Shenglin and Yang, Jiahai and Mo, Linlin and Zeng, Jice and Xue, Wenman and others. “Unsupervised Detection of Microservice Trace Anomalies through Service-Level Deep Bayesian Networks”. In: *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2020, pp. 48–58.

[149] Luo, Chuan and Qiao, Bo and Chen, Xin and Zhao, Pu and Yao, Randolph and Zhang, Hongyu and Wu, Wei and Zhou, Andrew and Lin, Qingwei. “Intelligent Virtual Machine Provisioning in Cloud Computing”. In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*. 2020, pp. 1495–1502.

[150] Mach, Pavel and Becvar, Zdenek. “Mobile edge computing: A survey on architecture and computation offloading”. In: *IEEE Communications Surveys & Tutorials* 19.3 (2017), pp. 1628–1656.

[151] Madsen, Henrik and Burtschy, Bernard and Albeanu, G and Popentiu-Vladicescu, FL. “Reliability in the utility computing era: Towards reliable fog computing”. In: *2013 20th International Conference on Systems, Signals and Image Processing (IWSSIP)*. IEEE. 2013, pp. 43–46.

[152] Magalhaes, Joao Paulo and Silva, Luis Moura. “A framework for self-healing and self-adaptation of cloud-hosted web-based applications”. In: *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. Vol. 1. IEEE. 2013, pp. 555–564.

[153] Malek, Miroslaw. “Predictive analytics: a shortcut to dependable computing”. In: *International Workshop on Software Engineering for Resilient Systems*. Springer. 2017, pp. 3–17.

[154] Mariani, Leonardo and Pezzè, Mauro and Riganelli, Oliviero and Xin, Rui. “Predicting failures in multi-tier distributed systems”. In: *Journal of Systems and Software* 161 (2020), p. 110464.

[155] Mark Russinovich. *Advancing Azure service quality with artificial intelligence: AIOps*. `https://azure.microsoft.com/en-us/blog/advancing-azure-service-quality-with-artificial-intelligence-aiops/`. (last access 21 May 2021). 2020.

[156] Masood, Adnan and Hashmi, Adnan. “AIOps: Predictive Analytics & Machine Learning in Operations”. In: *Cognitive Computing Recipes*. Springer, 2019, pp. 359–382.

[157] Matos, Rubens de S and Maciel, Paulo RM and Machida, Fumio and Kim, Dong Seong and Trivedi, Kishor S. "Sensitivity analysis of server virtualized system availability". In: *IEEE Transactions on Reliability* 61.4 (2012), pp. 994–1006.

[158] Maurer, Michael and Breskovic, Ivan and Emeakaroha, Vincent C and Brandic, Ivona. "Revealing the MAPE loop for the autonomic management of cloud infrastructures". In: *2011 IEEE symposium on computers and communications (ISCC)*. IEEE. 2011, pp. 147–152.

[159] Mell, Peter and Grance, Tim and others. "The NIST definition of cloud computing". In: (2011).

[160] Menage, Paul and Jackson, Paul and Lameter, Christoph. *cgroups - Linux Kernel Documentation*. `https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`. (last access 21 May 2021). 2004.

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

[162] Microsoft Corporation. *Press Release & Webcast: Earnings Release FY21 Q1 - Microsoft Cloud Strength Fuels First Quarter Results*. Tech. rep. `https://www.microsoft.com/en-us/Investor/earnings/FY-2021-Q1/press-release-webcast` (last access 22 April 2021). 2021.

[163] Miorandi, Daniele and Carreras, Iacopo and Altman, Eitan and Yamamoto, Lidia and Chlamtac, Imrich. "Bio-inspired approaches for autonomic pervasive computing systems". In: *Workshop on Bio-Inspired Design of Networks*. Springer. 2007, pp. 217–228.

[164] Miorandi, Daniele and Lowe, David and Yamamoto, Lidia. "Embryonic models for self–healing distributed services". In: *International Conference on Bio-Inspired Models of Network, Information, and Computing Systems*. Springer. 2009, pp. 152–166.

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

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

[167] Moulding, MR. *Fault Tolerant Systems in Military Applications*. Tech. rep. ROYAL MILITARY COLL OF SCIENCE SHRIVENHAM (ENGLAND), 1985.

[168] Mullany, Francis J and Ho, Lester TW and Samuel, Louis G and Claussen, Holger. "Self-deployment, self-configuration: Critical future paradigms for wireless access networks". In: *Workshop on Autonomic Communication*. Springer. 2004, pp. 58–68.

[169] Najjar, Walid and Gaudiot, J-L. "Network resilience: A measure of network fault tolerance". In: *IEEE Transactions on Computers* 39.2 (1990), pp. 174–181.

[170] Nallur, Vivek and Bahsoon, Rami. “A decentralized self-adaptation mechanism for service-based applications in the cloud”. In: *IEEE Transactions on Software Engineering* 39.5 (2012), pp. 591–612.

[171] Nedelkoski, Sasho and Bogatinovski, Jasmin and Acker, Alexander and Cardoso, Jorge and Kao, Odej. “Self-Attentive Classification-Based Anomaly Detection in Unstructured Logs”. In: *2020 IEEE International Conference on Data Mining (ICDM)*. IEEE. 2020, pp. 1196–1201.

[172] Nedelkoski, Sasho and Cardoso, Jorge and Kao, Odej. “Anomaly detection and classification using distributed tracing and deep learning”. In: *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE. 2019, pp. 241–250.

[173] Nedelkoski, Sasho and Cardoso, Jorge and Kao, Odej. “Anomaly detection from system tracing data using multimodal deep learning”. In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE. 2019, pp. 179–186.

[174] Netti, Alessio and Kiziltan, Zeynep and Babaoglu, Ozalp and Sîrbu, Alina and Bartolini, Andrea and Borghesi, Andrea. “Online fault classification in hpc systems through machine learning”. In: *European Conference on Parallel Processing*. Springer. 2019, pp. 3–16.

[175] Neumann, Peter G. *Computer-related risks*. Addison-Wesley Professional, 1994.

[176] Newman, Sam. *Building microservices: designing fine-grained systems*. " O’Reilly Media, Inc.", 2015.

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

[178] Northrop, Linda and Feiler, Peter and Gabriel, Richard P and Goodenough, John and Linger, Rick and Longstaff, Tom and Kazman, Rick and Klein, Mark and Schmidt, Douglas and Sullivan, Kevin and others. *Ultra-large-scale systems: The software challenge of the future*. Tech. rep. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 2006.

[179] Pallas, Frank and Raschke, Philip and Bermbach, David. “Fog computing as privacy enabler”. In: *IEEE Internet Computing* 24.4 (2020), pp. 15–21.

[180] Pandeeswari, N and Kumar, Ganesh. “Anomaly detection system in cloud environment using fuzzy clustering based ANN”. In: *Mobile Networks and Applications* 21.3 (2016), pp. 494–505.

[181] Paparoditis, Efstathios and Politis, Dimitris N. “Tapered block bootstrap”. In: *Biometrika* 88.4 (2001), pp. 1105–1119.

[182] Parmar, H. and Thornburgh, M. *Adobe’s Real Time Messaging Protocol*. `https://wwwimages2.adobe.com/content/dam/acom/en/devnet/rtmp/pdf/rtmp_specification_1.0.pdf`. (last access 20 April 2021). 2012.

[183] Paszke, Adam and Gross, Sam and Massa, Francisco and Lerer, Adam and Bradbury, James and Chanan, Gregory and Killeen, Trevor and Lin, Zeming and Gimelshein, Natalia and Antiga, Luca and others. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems* 32 (2019), pp. 8026–8037.

[184] Peng, Mugen and Liang, Dong and Wei, Yao and Li, Jian and Chen, Hsiao-Hwa. "Self-configuration and self-optimization in LTE-advanced heterogeneous networks". In: *IEEE Communications Magazine* 51.5 (2013), pp. 36–45.

[185] Petitjean, Franccois and Ketterlin, Alain and Ganccarski, Pierre. "A global averaging method for dynamic time warping, with applications to clustering". In: *Pattern recognition* 44.3 (2011), pp. 678–693.

[186] Pfandzelter, Tobias and Bermbach, David. "Iot data processing in the fog: Functions, streams, or batch processing?" In: *2019 IEEE International Conference on Fog Computing (ICFC)*. IEEE. 2019, pp. 201–206.

[187] Pfandzelter, Tobias and Bermbach, David. "tinyFaaS: A lightweight faas platform for edge environments". In: *2020 IEEE International Conference on Fog Computing (ICFC)*. IEEE. 2020, pp. 17–24.

[188] Pitakrat, Teerat and Okanovic, Dusan and Van Hoorn, André and Grunske, Lars. "An architecture-aware approach to hierarchical online failure prediction". In: *2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*. IEEE. 2016, pp. 60–69.

[189] Pokharel, Manish and Lee, Seulki and Park, Jong Sou. "Disaster recovery for system architecture using cloud computing". In: *2010 10th IEEE/IPSJ International Symposium on Applications and the Internet*. IEEE. 2010, pp. 304–307.

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

[191] Randell, Brian. "System structure for software fault tolerance". In: *Ieee transactions on software engineering* SE-1.2 (1975), pp. 220–232.

[192] Rausand, Marvin and Hoyland, Arnljot. *System reliability theory: models, statistical methods, and applications*. Vol. 396. John Wiley & Sons, 2003.

[193] Rennels, David A. "Architectures for fault-tolerant spacecraft computers". In: *Proceedings of the IEEE* 66.10 (1978), pp. 1255–1268.

[194] Riiser, Haakon and Endestad, Tore and Vigmostad, Paul and Griwodz, Carsten and Halvorsen, Pâl. "Video streaming using a location-based bandwidth-lookup service for bitrate planning". In: *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 8.3 (2012), pp. 1–19.

[195] Riiser, Haakon and Vigmostad, Paul and Griwodz, Carsten and Halvorsen, Paal. "Commute path bandwidth traces from 3G networks: analysis and applications". In: *Proceedings of the 4th ACM Multimedia Systems Conference*. 2013, pp. 114–118.

[196] Robbins, Herbert and Monro, Sutton. "A stochastic approximation method". In: *The annals of mathematical statistics* (1951), pp. 400–407.

[197] Robert, Cleveland and William, C and Irma, Terpenning. “STL: A seasonal-trend decomposition procedure based on loess”. In: *Journal of official statistics* 6.1 (1990), pp. 3–73.

[198] Rohn, WB. “Reliability prediction for complex systems”. In: *Proceedings of the Fifth National Symposium on Reliability and Quality Control*. 1959, pp. 381–388.

[199] Rumelhart, David E and Hinton, Geoffrey E and Williams, Ronald J. “Learning representations by back-propagating errors”. In: *nature* 323.6088 (1986), pp. 533–536.

[200] Rushby, John. “Critical system properties: Survey and taxonomy”. In: *Reliability Engineering & System Safety* 43.2 (1994), pp. 189–219.

[201] Sadashiv, Naidila and Kumar, SM Dilip. “Cluster, grid and cloud computing: A detailed comparison”. In: *2011 6th International Conference on Computer Science & Education (ICCSE)*. IEEE. 2011, pp. 477–482.

[202] Salfner, Felix. *Event-based failure prediction: an extended hidden markov model approach*. dissertation.de, 2008.

[203] Salfner, Felix and Lenk, Maren and Malek, Miroslaw. “A survey of online failure prediction methods”. In: *ACM Computing Surveys (CSUR)* 42.3 (2010), pp. 1–42.

[204] Salfner, Felix and Malek, Miroslaw. “Proactive fault handling for system availability enhancement”. In: *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2005, 7–pp.

[205] Sandvine. *The Global Internet Phenomena Report, October 2019*. `https://www.sandvine.com/press-releases/sandvine-releases-2019-global-internet-phenomena-report`. (last access 20 April 2021). Oct. 2019.

[206] Sari, Arif and others. “A review of anomaly detection systems in cloud networks and survey of cloud security measures in cloud storage applications”. In: *Journal of Information Security* 6.02 (2015), p. 142.

[207] Sarle, Warren S. “Stopped training and other remedies for overfitting”. In: *Computing science and statistics* (1996), pp. 352–360.

[208] Sauvanaud, Carla and Lazri, Kahina and Kaâniche, Mohamed and Kanoun, Karama. “Anomaly detection and root cause localization in virtual network functions”. In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2016, pp. 196–206.

[209] Scheinert, Dominik and Acker, Alexander. “Telesto: A graph neural network model for anomaly classification in cloud services”. In: *Service-Oriented Computing – ICSOC 2020 Workshops*. Springer International Publishing. 2021, pp. 214–227.

[210] Schmidt, Florian and Gulenko, Anton and Wallschlaeger, Marcel and Acker, Alexander and Hennig, Vincent and Liu, Feng and Kao, Odej. “Iftm-unsupervised anomaly detection for virtualized network function services”. In: *2018 IEEE International Conference on Web Services (ICWS)*. IEEE. 2018, pp. 187–194.

[211] Schmidt, Florian and Suri-Payer, Florian and Gulenko, Anton and Wallschlaeger, Marcel and Acker, Alexander and Kao, Odej. "Unsupervised anomaly event detection for cloud monitoring using online arima". In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE. 2018, pp. 71–76.

[212] Schmidt, Florian and Suri-Payer, Florian and Gulenko, Anton and Wallschlaeger, Marcel and Acker, Alexander and Kao, Odej. "Unsupervised anomaly event detection for vnf service monitoring using multivariate online arima". In: *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2018, pp. 278–283.

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

[214] Sefraoui, Omar and Aissaoui, Mohammed and Eleuldj, Mohsine. "OpenStack: toward an open-source solution for cloud computing". In: *International Journal of Computer Applications* 55.3 (2012), pp. 38–42.

[215] Shangguan, Chong and Tamo, Itzhak. "Error detection and correction in communication networks". In: *2020 IEEE International Symposium on Information Theory (ISIT)*. IEEE. 2020, pp. 96–101.

[216] Sharma, GN. "Hot redundant versus cold redundant systems". In: *Reliability Engineering* 2.3 (1981), pp. 193–197.

[217] Shkuro, Yuri. *Mastering Distributed Tracing*. Packt Publishing, 2019.

[218] Sidiroglou-Douskos, Stelios and Misailovic, Sasa and Hoffmann, Henry and Rinard, Martin. "Managing performance vs. accuracy trade-offs with loop perforation". In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011, pp. 124–134.

[219] Sillito, Jonathan and Kutomi, Esdras. "Failures and Fixes: A Study of Software System Incident Response". In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2020, pp. 185–195.

[220] Spiteri, Kevin and Urgaonkar, Rahul and Sitaraman, Ramesh K. "BOLA: Near-optimal bitrate adaptation for online videos". In: *IEEE/ACM Transactions on Networking* 28.4 (2020), pp. 1698–1711.

[221] Srivastava, Nitish and Hinton, Geoffrey and Krizhevsky, Alex and Sutskever, Ilya and Salakhutdinov, Ruslan. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[222] Su, Ya and Zhao, Youjian and Niu, Chenhao and Liu, Rong and Sun, Wei and Pei, Dan. "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*. 2019, pp. 2828–2837.

[223] Tanenbaum, Andrew S and Van Steen, Maarten. *Distributed systems: principles and paradigms*. Prentice-hall, 2007.

[224] Thanigaivelan, Nanda Kumar and Nigussie, Ethiopia and Kanth, Rajeev Kumar and Virtanen, Seppo and Isoaho, Jouni. "Distributed internal anomaly detection system for Internet-of-Things". In: *2016 13th IEEE annual consumer communications & networking conference (CCNC)*. IEEE. 2016, pp. 319–320.

[225] Tokoro, Mario. *Open systems dependability: dependability engineering for ever-changing systems*. CRC press, 2015.

[226] Tran, Lam and Choi, Deokjai. "Data augmentation for inertial sensor-based gait deep neural network". In: *IEEE Access* 8 (2020), pp. 12364–12378.

[227] Troeger, Peter. *Unsicherheit und Uneindeutigkeit in Verlaesslichkeitsmodellen*. Springer, 2018.

[228] Tuncer, Ozan and Ates, Emre and Zhang, Yijia and Turk, Ata and Brandt, Jim and Leung, Vitus J and Egele, Manuel and Coskun, Ayse K. "Diagnosing performance variations in HPC applications using machine learning". In: *International Supercomputing Conference*. Springer. 2017, pp. 355–373.

[229] Um, Terry T and Pfister, Franz MJ and Pichler, Daniel and Endo, Satoshi and Lang, Muriel and Hirche, Sandra and Fietzek, Urban and Kulić, Dana. "Data augmentation of wearable sensor data for parkinson's disease monitoring using convolutional neural networks". In: *Proceedings of the 19th ACM International Conference on Multimodal Interaction*. 2017, pp. 216–220.

[230] Upton, Graham and Cook, Ian. *A dictionary of statistics 3e*. Oxford university press, 2014.

[231] Variabilità, Gini C. "mutuabilità: contributo allo studio delle distribuzioni e delle relazioni statistiche". In: *Bologna (ITA): Tipogr. di P. Cuppini* (1912).

[232] Wallschlaeger, Marcel and Acker, Alexander and Kao, Odej. "Silent Consensus: Probabilistic Packet Sampling for Lightweight Network Monitoring". In: *International Conference on Computational Science and Its Applications*. Springer. 2019, pp. 241–256.

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

[234] Wang, Tao and Xu, Jiwei and Zhang, Wenbo and Gu, Zeyu and Zhong, Hua. "Self-adaptive cloud monitoring with online anomaly detection". In: *Future Generation Computer Systems* 80 (2018), pp. 89–101.

[235] Wang, Yu and Ma, Eden WM and Chow, Tommy WS and Tsui, Kwok-Leung. "A two-step parametric method for failure prediction in hard disk drives". In: *IEEE Transactions on industrial informatics* 10.1 (2013), pp. 419–430.

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

[237] Wensley, John H and Lamport, Leslie and Goldberg, Jack and Green, Milton W and Levitt, Karl N and Melliar-Smith, Po Mo and Shostak, Robert E and Weinstock, Charles B. "SIFT: Design and analysis of a fault-tolerant computer for aircraft control". In: *Proceedings of the IEEE* 66.10 (1978), pp. 1240–1255.

[238] Wetzig, René and Gulenko, Anton and Schmidt, Florian. "Unsupervised Anomaly Alerting for IoT-Gateway Monitoring using Adaptive Thresholds and Half-Space Trees". In: *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE. 2019, pp. 161–168.

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

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

[241] Xu, Haowen and Chen, Wenxiao and Zhao, Nengwen and Li, Zeyan and Bu, Jiahao and Li, Zhihan and Liu, Ying and Zhao, Youjian and Pei, Dan and Feng, Yang and others. "Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications". In: *Proceedings of the 2018 World Wide Web Conference*. 2018, pp. 187–196.

[242] Yang, Chi and Liu, Chang and Zhang, Xuyun and Nepal, Surya and Chen, Jinjun. "A time efficient approach for detecting errors in big sensor data on cloud". In: *IEEE Transactions on Parallel and Distributed Systems* 26.2 (2014), pp. 329–339.

[243] Yao, Yuan and Sharma, Abhishek and Golubchik, Leana and Govindan, Ramesh. "Online anomaly detection for sensor systems: A simple and efficient approach". In: *Performance Evaluation* 67.11 (2010), pp. 1059–1075.

[244] Zamojski, Wojciech and Mazurkiewicz, Jacek and Sugier, Jaroslaw and Walkowiak, Tomasz and Kacprzyk, Janusz. "Theory and Engineering of Complex Systems and Dependability". In: *Conference proceedings DepCoS-RELCOMEX*. 2015, p. 125.

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