

Autonomic Self-Healing in Cloud Computing Platforms

vorgelegt von
Anton Gulenko, M.Sc.

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

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. Florian Tschorsch

Gutachter: Prof. Dr. Odej Kao

Prof. Dr. Andreas Polze

Prof. Dr. Carsten Griwodz

Tag der wissenschaftlichen Aussprache: 17.06.2020

Berlin 2020

Acknowledgements

First and foremost, I have to thank my advisor, Odej Kao, for his continuous guidance over the last years. Tirelessly, he pushed me back on track whenever I lost focus. Thanks a lot for the opportunity to write this thesis and to present my work on international conferences! I would also like to thank Andreas Polze and Carsten Griwodz for agreeing to review this thesis, and for their valuable feedback.

Many people helped me with precious feedback, reviews and discussions. Thank you, dear Laila, for reading through the whole thing (more than once!), for your continuous curiosity and caring support all the way! And thank you, Tobi, for lightening up the feedback process with humorous comments on every page and for your professional insights from the IT industry. Many thanks also to you, Niko, for your valuable feedback. Matthew, I am very grateful for your input, you really helped me to improve my writing style.

I would like to thank my many colleagues at the CIT chair. The project team, Flo, Alex, Marcel, Sören – it was always interesting and fun to work with you! And thanks to Kevin, Lauritz, Morgan, Kordi, Ilya, Felix, and the rest of the CIT team for a great atmosphere at the office.

Of course I have to thank my dear family, mama, Mary, and Dieter, for your support over all these years!

Abstract

The demand for increasingly rich services with high-level abstractions drives the field of cloud computing. In both research and industry, additional layers and components continuously increase the complexity of these modern platforms. In truth, the size of cloud systems has long surpassed what human administrators are able to manage. Nonetheless, users and customers expect high availability and reliability from both the applications and the underlying platform, which is only possible through automation.

Today, most automated dependability techniques focus on increasing the availability of distributed systems by preventing or masking component outages. However, both software and hardware components often exhibit a behavior where the delivered service degrades without becoming entirely unavailable. Such *anomalies*, also called gray failures or degraded states, originate from software bugs or other unforeseen issues with the system. Some application-specific systems attempt to handle certain types of anomalies by applying a pre-defined set of rules. In general, however, administrators have to resolve anomaly situations manually. In practice, there is no technique or system for detecting and resolving anomaly situations in a generic way.

Accordingly, this thesis suggests an extension to traditional cloud infrastructures by providing self-healing functionalities. Our approach monitors live data streams collected from all critical system components and analyzes the collected data for anomalous behavior. Once an anomaly is detected, the system further investigates the situation, determines the root cause, and automatically implements a remediation plan to resolve the problem. We analyze the requirements to build such a self-healing system and present an abstract system architecture that fulfills the given requirements. The proposed self-healing cloud provides administrators with a coherent set of configuration values that determine the level to which remediation workflows are executed automatically. Further, we design a data analysis engine that executes the necessary processing tasks while co-existing with the cloud workload, and that, without disrupting it. Finally, we apply the abstract architecture to the scenario of a public cloud platform and present a prototypical implementation of the named concepts. We evaluate various properties of our implementation in a practical experimental testbed and a qualitative analysis.

Zusammenfassung

Die steigende Nachfrage nach Diensten mit immer höheren Abstraktionsebenen bestimmt heutzutage das Gebiet des Cloud Computing. Sowohl in der Forschung, als auch in der Industrie, führen zusätzliche Abstraktionsschichten und Komponenten zu wachsender Komplexität moderner IT-Systeme. Längst hat die schiere Größe von Cloudsystemen die Grenze des von Menschen Beherrschbaren überschritten. Nutzer und Kunden von Cloudplattformen erwarten dennoch ein hohes Maß an Zuverlässigkeit und Ausfallsicherheit, sowohl von der Plattform, als auch von den darin ausgeführten Anwendungen. Dies lässt sich nur mithilfe von Automatisierungslösungen erreichen.

Die meisten automatischen Lösungen für die Zuverlässigkeit von verteilten Systemen basieren darauf, Ausfälle von Teilkomponenten zu verhindern oder zu verschleiern. Dabei wird übersehen, dass sowohl Hardware- als auch Software-Komponenten auch ein degradiertes Verhalten aufweisen können, ohne komplett auszufallen. Solche Fälle, auch Anomalien genannt, entstehen häufig aus Fehlern im Programmcode einer Applikation, oder durch andere unvorhergesehene Umstände im System. Anwendungsspezifische Systeme für Anomalieerkennung und -behebung behandeln bestimmte Typen von Anomalien, basierend auf manuell festgelegten Regeln. Im Normalfall müssen Administratoren solche Anomaliefälle aber manuell behandeln. Momentan gibt es kein System in praktischer Benutzung, welches Anomalien erkennt und behebt, ohne dabei Annahmen über die überwachte Anwendung zu treffen.

Daher schlägt diese Dissertation eine Erweiterung von traditionellen Cloudplattformen vor, die solche Plattformen um selbstheilende Fähigkeiten erweitern. Unser Ansatz basiert auf Echtzeitdatenströmen, die von allen kritischen Komponenten des Systems erfasst werden. Diese Datenströme werden kontinuierlich analysiert, um festzustellen, ob die jeweilige Komponente sich normal verhält, oder eine Anomalie aufweist. Sobald eine Anomalie erkannt wird, wird die Situation automatisch weiter untersucht, die Ursache der Anomalie gefunden, und automatisch eine Gegenmaßnahme eingeleitet, um das Problem zu beheben. In dieser Arbeit analysieren wir die Anforderungen, die ein solches System erfüllen muss, und stellen eine abstrakte Systemarchitektur vor, die diese Anforderungen erfüllt. Diese selbstheilende Cloudarchitektur bietet Administratoren verständliche Konfigurationsparameter, die bestimmen, zu welchem Maß Reparaturaktionen automatisch ausgeführt werden. Desweiteren definieren wir das Konzept von “In Situ Datenanalyse”, die verwendet wird, um die Datenströme in der selbstheilenden Cloud effizient auszuwerten, ohne die Plattform bei ihren eigentlichen Aufgaben zu behindern. In einer prototypischen Umsetzung der abstrakten Konzepte zeigen wir, dass die selbstheilende Cloudarchitektur im praktischen Anwendungsfall eines öffentlichen Clouddienstes anwendbar ist. In einer experimentellen Testumgebung messen wir diverse Eigenschaften unserer Plattform, und erweitern diese Evaluierung durch eine qualitative Analyse.

Contents

1	Introduction	1
1.1	Problem Statement and Challenges	4
1.2	Main Contributions	5
1.3	Thesis outline	8
2	Background	11
2.1	Cloud Computing	11
2.2	Dependability in Distributed Systems	14
2.2.1	Reactive Fault Handling	15
2.2.2	Proactive Fault Handling	16
2.3	Autonomic Computing	17
3	Related Work	21
3.1	Autonomic Computing Platforms	21
3.1.1	Self-Healing Systems	22
3.1.2	Design-Time Self-Healing	25
3.2	Dependability in Cloud Computing Systems	26
3.3	AIOps Systems	27
3.4	Cloud and Data Center Monitoring	29
3.5	Data Analysis Platforms	30
4	Zero-Touch Operations	33
4.1	The Zero-Touch Operations Loop	35
4.1.1	Zero-Touch Across System Layers	38
4.1.2	Fault Model	40
4.2	Detailed Architecture	43
4.2.1	Modeling the System Topology	47

4.3	Data Analysis Pipeline	51
4.3.1	Overview	51
4.3.2	Anomaly Detection	55
4.3.3	Anomaly Classification	60
4.3.4	Root Cause Analysis	62
4.3.5	Remediation Workflow Assessment	63
4.3.6	Controlled Level of Automation	67
5	In Situ Data Analysis	69
5.1	Interfaces of Data Analysis Processes	71
5.2	Modeling the Data Analytics Pipeline	74
5.3	Enforcing Resource Limitations	78
5.4	Scheduling of Data Processing Steps	81
5.4.1	Static Scheduling	81
5.4.2	Adaptive Scheduling	82
6	ZerOps: A Self-Healing Cloud Platform	87
6.1	Use Cases	89
6.2	Components of the ZerOps Platform	92
6.2.1	High-Availability Deployment of OpenStack	94
6.2.2	Topology Discovery	95
6.3	Bitflow Stream Processing Framework	96
6.3.1	Bitflow Controller	97
6.3.2	Bitflow Stream Processing Runtime	100
6.3.3	Bitflowscript Language	103
6.3.4	Bitflow Data Collector	104
6.4	Experimental Evaluation	107
6.4.1	Tenant Service Scenarios	107
6.4.2	Anomaly Injection	110
6.4.3	Testbed and Experimental Procedure	113
6.4.4	Evaluation Results and Discussion	115
7	Conclusion	119
	Bibliography	121

Abbreviations

AIOps	IT Operations with Artificial Intelligence	MCDM	Multiple Criteria Decision Making
API	Application Programming Interface	NFV	Network Functions Virtualization
CD	Continuous Deployment	PaaS	Platform as a Service
CDN	Content Delivery Network	QoS	Quality of Service
DBaaS	Database as a Service	RCA	Root Cause Analysis
DPI	Deep Packet Inspection	ROC	Recovery Oriented Computing
DSL	Domain-Specific Language	SaaS	Software as a Service
HPC	High Performance Computing	SDN	Software Defined Network
IaaS	Infrastructure as a Service	SIP	Session Initiation Protocol
IDS	Intrusion Detection System	SLA	Service Level Agreement
IFTM	Identity Function Threshold Model	UI	User Interface
IoT	Internet of Things	vIMS	Virtualized IP-Multimedia Subsystem
ISP	Internet Service Provider	VM	Virtual Machine
IMS	IP Multimedia Subsystem	VNF	Virtualized Network Function
LBaaS	Load Balancing as a Service	VR	Virtual Reality

Chapter 1

Introduction

Contents

1.1 Problem Statement and Challenges	4
1.2 Main Contributions	5
1.3 Thesis outline	8

The number of layers and components in modern cloud computing systems keeps growing due to the increasing demand for more elaborate services. Cloud service providers do not limit their offerings to virtualized compute and storage resources in the form of Virtual Machines (VMs), but also provide higher-level abstractions, such as Platform as a Service (PaaS) or serverless computing. In addition to this deepening of the technology stack, future cloud platforms will experience an increase in geographical distribution as well as a diversification of the underlying hardware. Trends like *edge computing* drive this development by promising end-users lower latencies, better response times, and location-agnostic access to services by deploying service endpoints closer to the user's location.

Overlooking the growing complexity of modern cloud platforms, customers rightfully expect the services to be entirely reliable. However, the overwhelming scale of such platforms has long exceeded the maintenance capabilities of human administrators. The sheer number of components and subsystems found in modern data centers leads to an unmanageable fault rate. Furthermore, the number of human administrators maintaining cloud platforms cannot scale at the same pace as the systems' complexity. Therefore, the current industry standard is to rely on automated administration tools for deploying, upgrading, and monitoring cloud infrastructures.

Today, automation tools mainly focus on achieving *high availability* – i.e., keeping the services functioning and reachable. Since an unavailable service is one of the worst-case scenarios, countermeasures for outages have been researched extensively. The main

means of achieving high availability is through a form of redundancy, mostly by executing multiple replicas of the same service. By having a varying number of replicas managed automatically, this approach is also able to handle changes in the system's load. In the context of cloud computing, these concepts are known as *elasticity* and *auto scaling*. In addition to the ability to scale up and down, high availability depends on the fast detection of failed components. Many techniques are readily available for this task, including active and passive probing, heartbeats, and timeouts. However, high availability alone is not sufficient to make a service truly *reliable*: Besides failing entirely, services often exhibit other abnormal behaviors, which can lead to degraded service quality.

An *anomaly* is a deviation from normal system behavior. In the context of cloud computing, anomalies cover a much wider range of scenarios than simple crashes, and are, accordingly, difficult to detect and handle. In truth, an undesired behavior can degrade the service quality or even break Service Level Agreements (SLAs), and that, without making the service, or one of its components, entirely unavailable. Examples of anomalies include software aging effects, such as memory leaks, that lead to crashes after a certain time, and unexpected resource hogging, such as full CPU utilization. While high availability techniques, such as redundancy or failover to spare components, can handle crash faults, other anomalies lack proper handling mechanisms. Anomalies often precede component outages, thus making them a good target for fault prediction. In other words, early remediation of an anomaly can prevent a more severe fault or failure situation. Yet, continuous manual monitoring of all critical components is not feasible.

Automatic mechanisms are necessary when striving to ensure the anomaly-free execution of hundreds of services running on thousands of physical machines in multiple data centers. Traditional monitoring services observe various parts of the system and emit alarms when specific metrics exceed manually defined thresholds. Such alarms can trigger automatic actions, like scaling out, or even help administrators to identify and localize the anomaly in the system. However, state-of-the-art alarm-based monitoring systems have three important drawbacks:

1. Every layer and component is monitored and analyzed individually.
2. Fixed thresholds require expert knowledge and must be continuously updated.
3. Alarms usually mean that service quality is already reduced, which makes this approach reactive instead of proactive.

For large-scale data centers, this can lead to continuously large amounts of false alarms. Therefore, even with such static anomaly detection systems, it is time-consuming and often impossible for human administrators to understand and handle every anomaly manually. Aggregating alarms can reduce the number of alarms that need to be handled by human administrators, but does not solve the underlying problem. The manual handling

of alarms also suffers from the problem that the reaction time in an anomaly situation is much higher than with an automatic system.

This thesis presents an alternative way of managing cloud platforms, which mitigates the above problems by introducing the concept of *zero-touch operations*. The term “zero-touch” in conjunction with “operations” implies that once the system is running, human intervention is not necessary, except when implementing changes such as version upgrades. We envision a self-sustained, self-healing, closed-loop system that includes self-monitoring and data-driven situation analysis to detect and also resolve anomalies. Similarly to a closed-loop control system, the zero-touch operations system acts autonomously to retain the system’s normal operation, while continuous self-monitoring evaluates the success of executed remediation workflows.

Our approach exploits advancements in the field of machine learning and acknowledges the complex, multi-layer nature of modern cloud systems. The resulting zero-touch cloud operations system is a holistic, data-driven platform capable of monitoring, analyzing, and automatically repairing all layers and components of the technology stack. The collection of data describing the behavior of the entire cloud stack – i.e., from the physical nodes up to the running service processes – is the basis of the proposed system. Then, data mining techniques process the collected data and uncover hidden dependencies that would otherwise be missed by humans, but often indicate anomalous behavior. As a result, false alarms are reduced by not relying solely on thresholds of individual metrics, but instead by analyzing the correlations between various metrics on all system layers. Said correlations have the potential to reveal even more anomalies hidden in the data.

A self-healing cloud system has different use case scenarios, which vary based on the cloud service model and the relationship between the cloud provider and the user. In private cloud scenarios, where the maintainer of the service layer also owns the underlying infrastructure, a single instance of the zero-touch architecture manages all system layers. Private clouds have the most significant potential to find hidden anomalies and efficiently resolve them, since the system has complete ownership over the monitored infrastructure. In contrast, public clouds present a slightly different situation, where the zero-touch architecture manages only a part of the cloud and the application. The cloud’s customers own the upper layers and, usually, the service agreements prohibit the cloud provider from monitoring or accessing resources owned by the customers. The zero-touch architecture can still implement self-healing capabilities on the lower layers and offer an optional service with privacy-related restrictions.

An important aspect of a self-healing cloud platform is its *level of automation*. When the platform detects an anomaly: Is it allowed to restart components autonomously, or should it merely notify an administrator and provide helpful insights? Our approach provides a

small, comprehensive set of configuration values that control the automatic execution of remediation workflows. This gives administrators full control over the self-healing system and allows them to gain trust in its autonomous decisions.

1.1 Problem Statement and Challenges

Problem Statement: Design a practical self-healing cloud architecture for commodity hardware that is able to monitor itself and the black-box software running within. Based on the monitoring data, the system automatically detects and analyzes abnormal behavior of any of the monitored components, and selects appropriate countermeasures to remediate anomaly situations. Furthermore, the system gives human administrators comprehensive control over the level of automation.

In the above statement, the term *practical* refers to the applicability of the architecture in commercial or “critical” cloud platforms and applications – i.e., platforms where a malfunction has financial repercussions, or worse. The focus on *commodity hardware* limits the scope of this thesis to cloud platforms built on inherently unreliable, inexpensive hardware components, instead of specialized appliances. We reify the problem statement by defining three main requirements for our self-healing cloud architecture:

1. **Scalability:** All parts of the self-healing system must be scalable to a large number of monitored components. The system must also be capable of dealing with a changing size and topology of the cloud platform.
2. **Extensibility:** A self-healing cloud platform has use cases in many practical and academic fields. The design of such a platform must account for extensions, since new algorithms and monitoring data sources are likely to be added in the future.
3. **Unintrusiveness:** The resources for the involved data analysis tasks are limited in two ways. On the one hand, the results of the analysis must be available fast enough to make timely decisions regarding how to resolve an anomaly situation. On the other hand, the data analysis tasks of the self-healing system must not interfere with the actual workload of the cloud system, and therefore need a well-defined limit for the incurred resource overhead. Furthermore, the self-healing system must adhere to the restrictions imposed by the relationship between the provider and the user of the cloud, especially regarding sensitive user data and data security.

Fulfilling the problem statement and the three requirements holds challenges in different areas:

Algorithmic challenges. The elastic nature of cloud computing platforms leads to frequently changing workloads and a highly dynamic resource utilization. Even bug free software can exhibit unpredictable behaviors that highly resemble anomaly

situations. The algorithms applied for anomaly detection and situation analysis must deal with varying load situations and high similarities between normal and abnormal behaviors. Furthermore, not all possible anomaly situations are known beforehand, and labeled training data is likely not available since injecting artificial anomalies or faults in a productive system is often unacceptable. Lastly, in order to react to faults proactively, all data analysis tasks must work on unbatched data streams and produce results with very low latency. This allows a timely selection and execution of remediation workflows.

Architectural challenges. Since cloud platforms are deployed in large data centers, a self-healing cloud platform must monitor and analyze an accordingly large number of nodes and other components. It is crucial to design the data analysis in a scalable way to allow the extension of the underlying cloud. Both the monitoring and the data analysis must be *unintrusive*, meaning that the resource consumption of both must be limited in order not to disturb the main workload of the cloud. An additional challenge in the public cloud scenario is that customer services must be regarded as black boxes, which limits the options for monitoring and remediation workflows.

1.2 Main Contributions

The first contribution of this thesis lies in the systematic analysis of the problem space of a self-healing cloud platform, as well as an abstract architectural platform design. Our design is abstract in the sense that it can be applied to various cloud infrastructures, such as private or public clouds, and to cloud services on different levels of abstraction. We address the requirements of scalability and unintrusiveness when discussing the monitoring of a cloud platform, the analysis of the collected data, and the selection and automatic execution of remediation workflows in anomaly situations. Our system design allows administrators to define configuration values that determine the automatic execution of remediation workflows.

The second contribution is the design of a data analysis engine that meets the requirements of our self-healing cloud system. It is not feasible to use a dedicated cluster for data analysis, since streaming the monitoring data over the network consumes bandwidth and increases the data analysis latency. Therefore, we analyze the data directly at its origin – i.e., on the physical machines that also execute the cloud workload. Our *in situ* (Latin for *in place*) data analysis engine implements hard upper bounds for the resources consumed by the algorithms, in order to avoid disturbing the main cloud workload.

As third contribution, this thesis describes an adaptation of our abstract self-healing cloud design for the scenario of a public Infrastructure as a Service (IaaS) cloud. We

present ZerOps, an implementation of our self-healing cloud architecture, that also incorporates the *in situ* data analysis. In an experimental testbed we measure various runtime properties of ZerOps. Finally, we discuss how the platform design fulfills this thesis' problem statement and the defined requirements, while using the experimental results to quantify this evaluation.

Parts of the contributions of this thesis and the algorithms used therein have been published in the following publications:

Architecture

1. **Anton Gulenko, Marcel Wallschläger, Florian Schmidt, Odej Kao, and Feng Liu**
“A System Architecture for Real-Time Anomaly Detection in Large-Scale NFV Systems”
in: *Procedia Computer Science*. Vol. 94. Elsevier. 2016, pp. 491–496
2. **Saeed Haddadi Makhsous, Anton Gulenko, Odej Kao, and Feng Liu**
“High Available Deployment of Cloud-based Virtualized Network Functions”
in: *International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2016, pp. 468–475

Algorithms

1. **Anton Gulenko, Marcel Wallschläger, Florian Schmidt, Odej Kao, and Feng Liu**
“Evaluating Machine Learning Algorithms for Anomaly Detection in Clouds”
in: *International Conference on Big Data*. Elsevier. 2016, pp. 2716–2721
2. **Anton Gulenko, Florian Schmidt, Alexander Acker, Marcel Wallschläger, Odej Kao, and Feng Liu**
“Detecting Anomalous Behavior of Black-Box Services Modeled with Distance-Based Online Clustering”
in: *International Conference on Cloud Computing (CLOUD)*. IEEE. 2018, pp. 912–915

3. **Florian Schmidt, Anton Gulenko, Marcel Wallschläger, Alexander Acker, Vincent Hennig, Feng Liu, and Odej Kao**
“IFTM-Unsupervised Anomaly Detection for Virtualized Network Function Services”
in: *International Conference on Web Services (ICWS)*. IEEE. 2018, pp. 187–194
4. **Alexander Acker, Florian Schmidt, Anton Gulenko, and Odej Kao**
“Online Density Grid Pattern Analysis to Classify Anomalies in Cloud and NFV Systems”
in: *International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2018, pp. 290–295
5. **Florian Schmidt, Florian Suri-Payer, Anton Gulenko, Marcel Wallschläger, Alexander Acker, and Odej Kao**
“Unsupervised Anomaly Event Detection for Cloud Monitoring Using Online Arima”
in: *International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2018, pp. 278–283
6. **Florian Schmidt, Florian Suri-Payer, Anton Gulenko, Marcel Wallschläger, Alexander Acker, and Odej Kao**
“Unsupervised Anomaly Event Detection for VNF Service Monitoring Using Multivariate Online Arima”
in: *International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE/ACM. 2018, pp. 71–76

Monitoring

1. **Anton Gulenko, Marcel Wallschläger, and Odej Kao**
“A Practical Implementation of In-Band Network Telemetry in Open vSwitch”
in: *International Conference on Cloud Networking (CloudNet)*. IEEE. 2018
2. **Marcel Wallschläger, Anton Gulenko, Florian Schmidt, Odej Kao, and Feng Liu**
“Automated Anomaly Detection in Virtualized Services Using Deep Packet Inspection”
in: *Procedia Computer Science*. Vol. 110. Elsevier. 2017, pp. 510–515

3. **Marcel Wallschläger, Anton Gulenko, Florian Schmidt, Alexander Acker, and Odej Kao**

“Anomaly Detection for Black Box Services in Edge Clouds Using Packet Size Distribution”

in: *International Conference on Cloud Networking (CloudNet)*. IEEE. 2018

1.3 Thesis outline

The remainder of this thesis is organized as follows:

Chapter 2 “Background” provides the necessary background to discuss the idea of a self-healing cloud platform. First, we provide a historical and technical overview of cloud computing and its service models. Then we discuss the aspect of dependability in distributed systems, including definitions and techniques for fault handling that are important for defining a proper fault model. Finally, the field of Autonomic Computing provides a framework of terminology that puts the self-healing cloud system in perspective to related and extended visions.

Chapter 3 “Related Work” discusses a body of work related to self-healing cloud and computer systems and their components. The literature contains several concepts and specific platforms for autonomic computing and self-healing systems. The recently established term *IT Operations with Artificial Intelligence (AIOps)* is increasingly gaining attention in the industry but is not yet extensively covered by academic publications. Further topics of interest are monitoring techniques for data centers and cloud platforms, and distributed analysis of data streams.

Chapter 4 “Zero-Touch Operations” presents the first contribution of this thesis: the systematically designed, abstract architecture of a self-healing cloud system. We begin by discussing different areas of application for such a system and derive a fault model that clarifies how the self-healing functionality co-exists and interacts with other typical components of modern data centers. We then discuss the components of the self-healing cloud on different layers of abstraction, starting from a high-level overview and continuing down to a detailed analysis.

Chapter 5 “In Situ Data Analysis” describes our second contribution: the design of a data analysis engine that fulfills the requirements of the self-healing cloud. Instead of relying on a dedicated data analysis infrastructure,

our system processes data *in situ* – i.e., directly where the data is collected.

Chapter 6 “ZerOps: A Self-Healing Cloud Platform” applies the abstract design from Chapter 4 to the scenario of an IaaS cloud. As part of the third contribution of this thesis, we explain all relevant design and implementation decisions. Further, we evaluate multiple aspects of our design in an experimental testbed and discuss the results and the overall platform design.

Chapter 7 “Conclusion” concludes this thesis by summarizing our approach and our results, and by pointing out directions for future research and further improvements.

Chapter 2

Background

Contents

2.1	Cloud Computing	11
2.2	Dependability in Distributed Systems	14
2.2.1	Reactive Fault Handling	15
2.2.2	Proactive Fault Handling	16
2.3	Autonomic Computing	17

The motivation behind a self-healing healing cloud system is to increase the dependability through automation. This chapter provides the reader with the necessary background to understand our approach and the underlying concepts. We cover the field and technology of cloud computing, which is the domain of our approach. The taxonomies of dependability lay a common ground for discussions of faults, errors, and other related terminology. Finally, the field of Autonomic Computing defines the concept of *self-healing*.

2.1 Cloud Computing

The National Institute of Standards and Technology (NIST) defines cloud computing as:

“A model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.” [118]

Although the term *cloud* has been first mentioned and properly defined in the late 2000's, the vision of *utility computing* originates in the sixties [12, 138]. The computer utility vision was to make computing power available to the public in an affordable way, similarly to electrical power or the telephone service [12]. However, utility computing remained a more or less theoretical concept for a long period. Academical approaches attempted to realize this vision in the eras of cluster computing and grid computing, but ultimately only the advance of public cloud platforms managed to make computing resources available to customers as a utility. It is interesting to note which circumstances lead to the success of the cloud computing paradigm around 2007. According to an analysis performed by Fox et al. [56], it was a combination of easier payments for internet-based services, a decrease in prices for electricity and data center housing, and the need for new types of applications such as mobile interactive applications and big data analytics.

Around the advent of cloud computing, many researchers and companies published taxonomies and surveys as an effort to define and standardize this new technology [56, 80, 118, 147, 184]. At the same time, both open source cloud management software, and public cloud providers working with closed source software, started to evolve rapidly. The most commonly used taxonomy is the differentiation of cloud service categories, based on how the abstraction level of the service compares to the underlying hardware. Figure 2.1 shows the main stack of cloud computing abstractions and service models. IaaS denotes a service model where cloud infrastructure is made available to customers in a self-served and elastic manner. Cloud infrastructure includes VMs as compute resources, cloud data storage, and virtual networks with common network functions to control the access to provisioned VMs. The network functions cover access control through concepts similar to firewalls, and often some commonly used functions such as load balancers or IP address management. The PaaS model allows customers to execute software on an agreed-upon platform, while hiding underlying details such as where and how the platform is deployed, how many machines it runs on, where the data is stored, and other aspects such as the reliability mechanisms of the platform. This model offers customers a richer set of functionalities and an easier application deployment process, but in return provides less flexibility for configuring the infrastructure and platform layers. Finally, Software as a Service (SaaS) forms the upper layer of the cloud stack and describes a service model where customers do not deploy their own application software, but instead use the application that is entirely managed by the cloud provider. These applications must be accessible remotely and usually have a web-based user interface. Since a large number of applications fulfill these requirements, SaaS offerings are far more numerous than services on the lower

cloud stack layers.




Service Class	Main Access Tool	Service Content
 SaaS	Web Browser	Cloud Applications Social networks, Office suites, CRM, Video processing
 PaaS	APIs, IDEs, Versioning Systems	Cloud Platform Programming languages, Frameworks
 IaaS	APIs	Cloud Infrastructure Compute Servers, Data Storage, Firewall, Load Balancer

Figure 2.1: The cloud computing stack and service models [174] (p.14)

In more recent years, services associated with cloud computing have evolved rapidly. Public cloud providers keep incorporating new trends of the IT industry into their service portfolio, which go far beyond the original IaaS model. Depending on the point of view, higher-level services can be associated with either the PaaS or SaaS layers. As-a-service offerings that host specific software solutions include Database as a Service (DBaaS) [41, 73, 75] for managed and elastic database storage, or more generally Backend as a Service [39, 94], which covers both storage and backend processing for mobile applications. Other cloud services are more domain specific, like Sensing as a Service for Internet of Things (IoT) applications and smart cities [140, 160, 183], or Machine Learning as a Service [16, 34, 146] for specialized or general purpose data analysis and knowledge extraction tasks.

These services are subject to clearly defined SLAs between the service providers and their customers. Failing to uphold these SLAs has direct financial repercussions. For example, as of January 2020, most Google Cloud Platform services entitle the customer to a refund of 10% when the service availability drops below

99.9% during one month¹. An availability below 95% results in a 50% refund. The Amazon Web Services (AWS) Elastic Compute Cloud (EC2) service provides slightly “stronger” SLAs²: A 10% refund is promised for an availability below 99.99%, a 30% refund below 99%, and a full refund of 100% below 95% availability. The exact definition of “availability” differs amongst cloud providers, but is usually reduced to a binary decision of whether the service is reachable or not. In reality, cloud services can exhibit reduced service quality such as higher response latency or resource starvation. Such degraded service states are usually not covered by modern SLAs.

2.2 Dependability in Distributed Systems

Like any distributed system, or any computer system in general, cloud systems suffer from reduced dependability due to faults, errors, and failures. Laprie defines the dependability of a computer system as “*the quality of the delivered service such that reliance can justifiably be placed on this service*” [95]. A more quantifiable definition states that dependability denotes a system’s ability to avoid service failures that are more frequent and more severe than acceptable [14]. Following the common taxonomy of Laprie on dependable systems [95], a *fault* is a system-internal event that puts the system, or an individual component, into a latent *error* state. Once the service provided by the system deviates from the specified characteristics in any way, the error manifests itself in a system *failure*.

The dependability of a system can be measured as a number of *attributes*. The **availability** denotes the percentage of the system’s execution time, during which the service was delivered correctly. The **reliability** is the likelihood that the system performs failure-free for a given time, or in other words the average duration until the system fails. **Safety** is the absence of catastrophic consequences for the users of the system and its environment, **integrity** is the absence of improper transitions of internal system state, and **maintainability** is the ability to undergo changes such as modifications and repairs.

A systems dependability can be increased through techniques that are generally classified into four categories. Through **fault prevention**, a system is constructed in a way that faults are less likely to occur. Fault prevention techniques can reduce the number of faults, but never entirely eliminate their occurrence. Therefore, **fault tolerance** techniques are applied to prevent client-visible system failures even in the presence of faults. In practice, this is the predominant strategy to

¹<https://cloud.google.com/terms/sla/>

²<https://aws.amazon.com/compute/sla/>

increase the availability of a system. **Fault removal** techniques deal with localization and reduction of faults and error states, and finally **fault forecasting** tries to estimate the current and upcoming number and type of faults.

Figure 2.2 summarizes the main concepts related to dependability in a taxonomy tree. In the context of cloud computing, the term *system* is rather ambiguous: Depending on the point of view, it can denote the infrastructure managed by the cloud provider, or additionally include the service software deployed on the cloud. In fact, following the taxonomy of dependability, a system is recursively composed of other systems, until an *atomic* system prevents further destructuring. However, in this thesis, when not noted otherwise, the term *system* will refer to what is perceived by a human end-user, namely the entire cloud stack including the application.

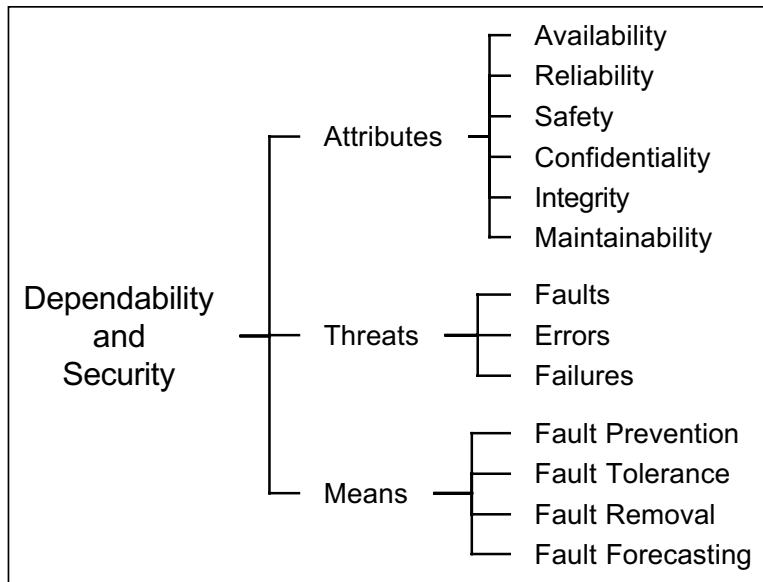


Figure 2.2: The tree of concepts related to dependability [14]

2.2.1 Reactive Fault Handling

Out of the four dependability means, fault forecasting and fault prevention can be classified as proactive, while fault removal and fault tolerance are applied after the fault and are therefore reactive. Fault tolerance is mainly achieved by masking faults through redundancy in space or time. Redundancy in *space* refers to the replication of a resource, so that a service can be fulfilled even when one of the replicas has experienced a failure. Redundancy in *time* means retrying a request,

resubmitting a job, or another form of duplication of the same behavior. While retrying or resubmitting is the simplest form of redundancy in time, there are more elaborate solutions that raise the probability of success. A typical technique in this category is checkpointing: After experiencing a failure, a component or subsystem is reset to an earlier state (or checkpoint) that was known to be error-free. Afterwards, the original service is re-requested, or the previous computation is resumed from the checkpointed state. Another example for “smart” retrying is the *circuit breaker* pattern [126]. When the requested service of another component fails repeatedly, the component is internally marked as failed. The next time the service is required, it is immediately assumed to have failed, and a graceful alternative or error handling mechanism is executed instead. This avoids lingering resources and resource leaks, for example of timed out TCP connections or waiting threads. The circuit breaker pattern decouples system components from each other and prevents cascading failures. In regular intervals the failed service is marked to be requested again, to find out whether it has recovered in the meantime.

On a more generic level that requires no involvement of the application software, the most common fault handling strategy is to reboot a machine or restart a service. A reboot is the most trusted way of reclaiming resources and bringing a system back to a well-known and functioning state. Rebooting is easily understood by administrators and users [29], and often fixes even pseudo-nondeterministic faults that are hard to reproduce and to handle through other means. The drawback is that the rebooted resource is unavailable during the time of shutting down, booting up, and initialization, before the service can resume normal operation. Cloud computing offers possible mitigation strategies for this through rapid elasticity and replication. Due to the stateless nature of cloud-native applications, it is often feasible to simply deallocate a failed resource entirely and rapidly provision a replacement.

2.2.2 Proactive Fault Handling

Proactive fault handling necessitates a criterion that indicates the probability for an active fault in the future. This is opposed to the fault detection in reactive fault handling, because detecting actual non-dormant faults is usually easier and more precise than predicting them.

The most common proactive fault handling mechanism in the cloud world is auto-scaling [103]. When certain predefined thresholds for the load of an application component are exceeded, the number of replicas for the component is increased or decreased, respectively. The chosen thresholds attempt to optimize the trade-

off between costly overprovisioning and reduced service quality due to overload. Therefore, the main fault that auto-scaling tries to prevent is overload, which could otherwise impact the service quality or lead to a crash. A major challenge and research topic around auto-scaling is the definition of the underlying resource thresholds [48, 107].

Grottke et al. have proposed an extension to the classical terminology of faults, errors, and failures [67, 150]. The term *symptom* is introduced as a cause of an error state that has not yet turned into a failure, but is causing “out-of-norm behavior of system parameters as a side effect” [150]. Since symptoms manifest in system parameters and metrics, they enable proactive fault handling by monitoring said parameters. Gabel et al. use the term *latent fault* referring to periods of degraded performance that precedes failures [57]. They report that 20% of machine failures are preceded by such degraded states. Throughout this thesis, we refer to symptoms and latent faults as *anomalies*.

2.3 Autonomic Computing

Due to dropping costs and rising computational capacity, computer systems are evolving to perform more and more tasks with steadily decreasing human interaction. Different aspects of this property of computer systems are referred to by multiple phonetically similar terms: *automatic* (or *automated*), *autonomous*, and *autonomic*. The following disambiguation shows the differences between these terms and defines how we use these terms throughout this thesis.

An **automated** system performs a fixed set of tasks **automatically** – i.e., in a deterministic fashion, following pre-defined rules and conditions. No additional intelligence is applied when selecting and executing an action, beyond the heuristics implemented by the designer of the automated system. The automated system continues functioning in the same way, regardless of changes in the environment or external influences. The automated task can be arbitrarily complex and range from simple time-based alarms to the execution of software integration tests, or more complex robotic actions in factories or power plants.

On the other end of the spectrum, a fully **autonomous** system performs a task entirely without human interaction, while continuously adapting to changes in the environment. This term is often used in the artificial intelligence community, since some form of intelligence is necessary to adapt to previously unknown circumstances. Typically, an autonomous computer system maintains a machine learning representation of its surroundings and of the goal to achieve, and uses this repre-

sentation to derive the optimal action in each situation. For an autonomous actor, the communication with its environment and other autonomous actors within it, play an important role. Research in the field of autonomous systems often draws inspirations from biology and the human brain, since living organisms in nature are prime examples of autonomous systems [177]. Following this description, automated systems are a subset of autonomous systems.

The term **autonomic** originates from the autonomic nervous system, which is the part of the nervous system that acts unconsciously and controls many bodily functions such as the heart rate or the digestive system. In the early 2000s, IBM used this biological term to postulate *autonomic computing*, which laid the common ground for numerous research activities striving towards the common goal of an entirely self-managed computer system [59, 81, 89]. Autonomous and autonomic computing have large overlappings in their general visions but emphasize different aspects. Autonomous systems, on the one hand, mainly interact with their environment and other autonomous systems within it, to reach a common goal. An autonomic system, on the other hand, puts more focus on being self-aware and self-managed, understanding what components and subsystems it consists of, and what states these parts are in. An autonomous system usually strives to achieve some functional, domain-specific mission in an optimal way, and optimizes less towards non-functional requirements such as resource consumption, or upholding SLAs. This, however, is exactly the main goal of autonomic computing: Ensure that the domain-specific part of the system operates within designated boundaries of metrics such as request latency, service availability, or cost. Depending on the advancement of an autonomic system, these tasks require elaborate algorithmic models that are on par with the requirements of autonomous systems.

A natural illustrating example for an autonomous system is a person that has been given a task. That person autonomously interacts with other people and the environment to fulfill that task. In the meantime, the autonomic nervous system of that person monitors its bodily functions and ensures that the person can fulfill the task under the best conditions. To fulfill its own function, the autonomic nervous system does not communicate with anything external and focuses on its internal domain only, while the autonomous person handles the actual “domain-specific” task. This example shows how autonomous and autonomic aspects of one system complement each other.

In the IBM vision for autonomic computing, any regular component of a computer system is enhanced by a so-called *autonomic agent*. While this agent handles the aspects of autonomic computing, the component itself implements the domain-

specific functionality. The autonomic agent performs four core activities: self-configuration, self-healing, self-optimization, and self-protection. As previously noted, all these activities are primarily focused on understanding and managing its own internal structure, state, and the state of its subsystems. In order to implement these activities, the autonomic agent consists of high-level building blocks as depicted in Figure 2.3. Sensors deliver monitoring data that describes aspects of the managed element. The monitoring data can be numerical or of some other format, but must ultimately be machine readable. Next, the collected data is analyzed, a plan of action is created, and finally executed. Since the sensors continuously deliver their data, the consequences of the executed actions can be observed, and saved for future iterations. This closed-loop system allows to continuously learn how the managed element behaves in different situations, which is symbolized by the central *knowledge* extraction part. Together, the steps Monitor, Analyze, Plan, Execute, and Knowledge form the MAPE-K loop [82].

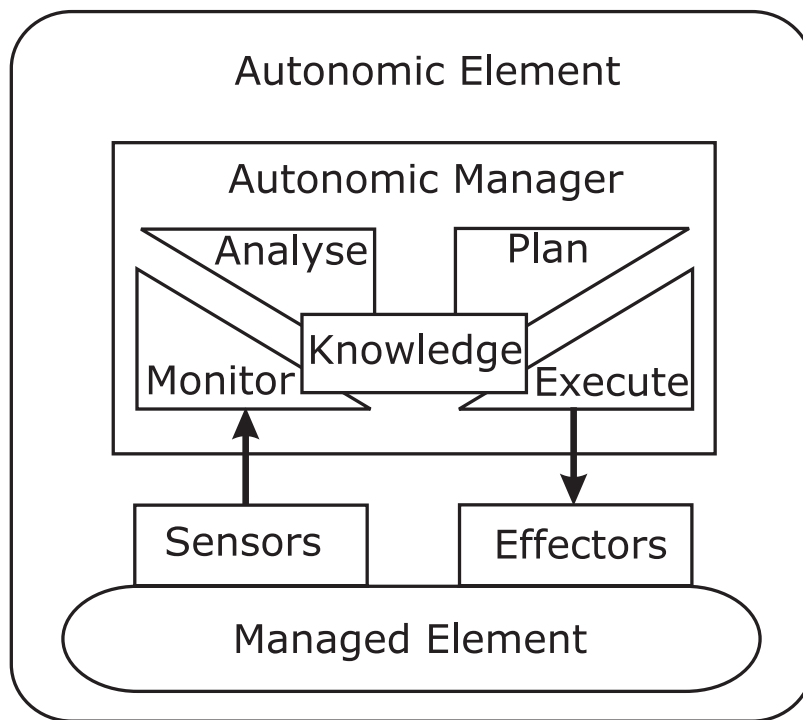


Figure 2.3: The MAPE-K loop of autonomic computing systems: Monitor, Analyze, Plan, Execute & Knowledge [82]

Levels of Automation

When working with automation, it is important to have a clear understanding of the type and degree of the automated tasks. The literature contains a multitude of models and taxonomies to describe automation [137, 169]. The idea of modeling the interaction between humans and automated computer systems dates back to the seventies, when Sheridan et al. discussed the interaction between divers and underwater support systems [161].

Ganek et al. list five evolutionary steps before a system can be called fully autonomic. In *basic* systems, the IT staff has to manually process monitoring data from multiple sources and act accordingly. A *managed* system provides a consolidated view over all components and an overview over all available data. A system with *predictive* capabilities autonomously recommends actions, which the IT staff reviews and approves. Finally, an *adaptive* system automatically acts according to predefined rules, and a fully *autonomic* system acts based on high-level business policies. Another white paper by IBM [83] contains one of the earliest holistic architectural concepts of an autonomic, self-managing computer system. The white paper describes five degrees of automation (ranging from manual management to business-aware closed-loop self-management) and five control scopes on which self-management can be incorporated (ranging from the sub-component level up to an entire business). No specific methodologies or algorithms are mentioned to actually implement these goals.

Parasuraman et al. propose a more general taxonomy of automation [137]. The authors identify four basic functions performed by an automated system: information acquisition, information analysis, decision selection, and action implementation. Each automated system performs these functions to a certain degree on a scale that ranges from low to high. For decision selection and action implementation, Parasuraman et al. suggest ten levels of automation that range from no assistance at all, over the suggestion of action alternatives, up to full automation without any human interaction.

Of all taxonomies suggested to date, none have been generally accepted as a default framework for describing levels of automation. Therefore, we do not strictly follow any suggested taxonomy in this thesis. Instead, our approach is based on the following idea: In the context of a self-healing cloud system, the main distinction is whether the system merely *suggests* repair actions to a human administrator, or automatically implements them.

Chapter 3

Related Work

Contents

3.1	Autonomic Computing Platforms	21
3.1.1	Self-Healing Systems	22
3.1.2	Design-Time Self-Healing	25
3.2	Dependability in Cloud Computing Systems	26
3.3	AIOps Systems	27
3.4	Cloud and Data Center Monitoring	29
3.5	Data Analysis Platforms	30

In this thesis we develop the architecture of a self-healing cloud platform. Similar to the vision of *autonomic computing*, our approach depends on the following basic building blocks:

- collection of monitoring data,
- analysis of the collected data and extraction of knowledge, and
- acting upon the analysis results.

This chapter presents a state-of-the-art review of these individual building blocks, as well as of the surrounding fields of research and of existing systems with related capabilities.

3.1 Autonomic Computing Platforms

The methodology introduced in this thesis describes a cloud computing platform that implements a large part of the autonomic computing activities for a cloud platform. The focus lies on self-healing, but self-configuration and self-optimization

are tackled as well. Self-protection is out of scope, as the protection against attacks on a cloud platform falls in the domain of internet security, and is handled by dedicated security systems. We consider a self-healing cloud platform as complementary to security systems such as Intrusion Detection Systems (IDSs).

Different approaches attempt to practically implement parts of the autonomic computing vision for distributed applications. Before the elasticity of cloud computing, Ardagna et al. proposed a Quality of Service (QoS)-driven scheme for selecting and chaining web services [11]. This approach relies on dynamic redistribution of resources between web services, which can be seen as a limited form of the rapid provisioning of cloud platforms, and works solely on the application layer. Similarly, Kephart et al. propose the use of abstract utility functions to express the global goal of a system, and leave it to the system to achieve that goal with the appropriate means [90]. Utility functions provide great flexibility and expressiveness and are generally preferred over simple “action policies”, where specific conditions are explicitly mapped to actions that must be executed. However, utility functions depend on domain-specific knowledge and are therefore not applicable to cloud computing platforms with multiple independent tenants and black-box workloads. Maurer et al. discuss extending the MAPE-K autonomic computing loop to cloud architectures [113]. Their approach models SLAs in form of resources requested from the cloud provider, and uses monitoring to detect SLA violations. Although Maurer et al. explicitly model the virtualization layer of a cloud infrastructure, they do not consider monitoring on lower physical layers [113]. All noted approaches have in common that the execution phase of the MAPE-K cycle is either missing or not practically applicable. Therefore, we do not consider this class of related approaches as *closed-loop* systems that really implement self-healing capabilities.

3.1.1 Self-Healing Systems

Self-healing is the activity of an autonomic system that is closest related to the approach presented in this thesis. According to Rodosek et al., a *self-healing* system “can make by itself all necessary recovery steps to restore its disturbed behavior to a specified mode of operation” [148].

Many systems and domains exhibit self-healing capabilities [143]. Networks of embedded systems use reconfiguration and rerouting to ensure continuous connectivity between critical components [64]. Experimental support for low-level self-healing has been introduced on the operating system level, e.g., the reincarnation server for automatic restarts of applications in Minix 3 [78], or predictive

self-healing in Solaris 10 [158].

Reflective middleware [91, 117] allows a form of meta programming through self-representation: An abstract model of the entire distributed system is available to all interacting components. Self-representation is the basis to enable self-healing capabilities in middleware systems, such as OpenCORBA [96] and OpenORB [20]. The distributed service middleware OSIRIS-SR [155, 165] implements a peer-to-peer scheme for mutual monitoring and fault handling. The peer-to-peer approach makes this system exceptionally scalable and resilient due to the lack of a central management entity. OSIRIS-SR mainly considers crash faults and migrations as recovery strategy, but other faults and recoveries could likely be integrated in the general scheme. Numerous other middleware approaches follow a similar approach of online adaptation and dynamic composition of services [30, 120, 167]. All named systems and approaches are either domain-specific, or require explicit adoption by the application, and hence are not generically applicable for arbitrary cloud applications.

In the area of High Performance Computing (HPC) and cluster computing, the project HA-OSCAR [102, 131] provides high-availability capabilities that can recover from crash faults. The different parts of a cluster, namely network, compute servers and clients, are monitored through a simple polling mechanism, and redundant standby resources automatically take over for failed components until they are repaired. Such a classical high-availability setup works solely with crash faults and imposes additional cost through standby replicas. Due to these limitations, we do not consider such a system as fully “self-healing”.

The research field of *computer immunology* shows similarities to self-healing systems [26, 54, 55, 163]. Computer immunology borrows ideas from natural biological immune systems and follows the goal to autonomously protect a system from external harmful influences, such as infections and viruses. Translated to distributed computer architectures, malicious external influences are cyber attacks. The main similarity between the concept of a computer immune system and a self-healing system are design principles such as *distributability*, *multi-layered*, *autonomy*, and *adaptability* [163]. Since a self-healing system attempts to deal with anomaly situations of mostly non-malicious origin, we consider computer immunology as an aspect of computer security and therefore complementary to the approach of this thesis.

Software rejuvenation [112, 170] is the process of periodically reinitializing running applications to counter the effects of software aging [60, 100]. Software aging has

been reported as the effect of potential fault conditions that accumulate over time and lead to performance degradation, transient faults, or both [170]. The most prominent example of software aging is leakage of resources, such as memory. The reinitialization process runs either in an *open-loop* architecture – i.e., in regular time intervals and without regard to the actual system state, or in a *closed-loop*, which implies application monitoring and a policy that defines the condition for triggering a rejuvenation process. We regard software rejuvenation as the simplest form of self-healing. The self-healing cloud approach presented in this thesis supports a closed-loop form of software rejuvenation without manual definition of resource thresholds as trigger conditions.

Breitgand et al. propose a system that performs both design-time and run-time self-healing for functional problems, concurrency bugs, and performance SLA violations [24]. The authors do not provide details on the problem resolution strategy, and the run-time remediation is limited to performance-related problems.

The Rainbow framework [61] defines a reusable high-level architectural scheme to add self-adaptation capabilities to a system. Rainbow continuously observes the managed system and maintains a dynamic graph-based model of relevant system components. Designers of the self-adaptation capabilities define *invariants* based on the information available in the abstract system model, e.g., that the response latency of a web server must be below a certain threshold. Violated invariants trigger remediation strategies, that are expressed in an imperative-style programming language. Rainbow consists of many components that also make up the self-healing cloud architecture suggested in this thesis. The main difference is that the designer of the self-adaptation capabilities must explicitly define all invariants and their exact mappings to appropriate remediation strategies. This does not address unforeseen anomalies and fault situations, which frequently occur in a complex system, such as a cloud infrastructure.

Cheng et al. present an architectural approach that allows multiple autonomic self-management agents to cooperate in the management of supervised components [35]. The approach is based on a common interface to the supervised components, which exports measurements, resource discovery, and allows executing actions. The common interface is used by all autonomic agents, which allows consistent and coordinated management decisions. The resulting benefit is that multiple autonomic agents can be used in a modular way, resulting in a flexible and extensible architecture. The drawback of this approach is that all autonomic agents must use the provided interface, which has limited applicability in systems with legacy support for aspects related to self-healing, like auto-scaling or automated crash

fault handling.

Kaiser et al. present their KX system [87], which is designed to add autonomic capabilities to legacy systems. In order to support arbitrary software components and architectures, KX uses the abstract notion of *probes* and *gauges*. While probes collect data from the supervised system, gauges analyze the incoming data and publish their results for consumption by a centralized controller, which in turn executes actions through *effectors*. This basic architecture is inspired by the general autonomic computing loop and has a high resemblance to the approach proposed in this thesis. However, Kaiser et al. do not explicitly account for the deep technology stack in modern cloud infrastructures, and ignore dependencies between system layers and components. Furthermore, the KX system works with events instead of continuous data streams, which makes it less applicable to other types of input data and limits the types of usable data analysis algorithms.

Ad-hoc multi-owner systems, such as grid infrastructures or collaborating agents, allow a bottom-up approach to self-healing [154]. Instead of a centralized or hierarchical organization, every entity is responsible for its own operation and self-healing capabilities. This extended self-organization is more scalable and allows for more diverse problem resolution strategies but can also produce unpredictable and chaotic results. Furthermore, the bottom-up approach is less applicable to systems with a deep technology stack and shared virtual resources, such as cloud environments. The complex inter-dependencies between different system layers are unknown to atomic components of the system. For example, a VM is by design isolated from its hypervisor and neighboring VMs, and therefore cannot properly contribute to the remediation of problems on the physical layer.

In the architectural pattern of *Embryo-ware*, each component of a system can dynamically switch between different roles as needed, depending on the current situation [122, 123]. In the example of a standard three-tier web architecture, a VM image would contain all software components necessary to execute the load balancer, web server, or database, and act as one of these roles as needed based on the load situation in the different tiers. We consider such capabilities as less relevant in a cloud environment, since the elasticity of clouds allows to simply replace a resource instead of converting it to another type.

3.1.2 Design-Time Self-Healing

Most self-healing strategies try to remediate problems that occur during the execution of an application or a system. However, since rigorous testing and robust

application design are one form of design-time fault removal, that process can be extended to *automatically* repair certain parts of an application's source code. In this case, anomaly detection is the process of finding programming errors, and the remediation is the process of replacing faulty code with a corrected version of equal functionality. Usually, developers of an application would perform these tasks manually, but when the entire process is automated, it can be seen as part of the self-healing system. Shehory implements such a scheme to automatically resolve concurrency and functional problems in source code [159]. However, this approach is not applicable generically to applications running on cloud platforms, and we see it as a complementary strategy for making applications more robust.

3.2 Dependability in Cloud Computing Systems

Due to the distributed and multi-layered nature of cloud systems, traditional dependability assessments and models from literature [25, 162, 180] are not directly applicable to cloud systems. Several dedicated dependability analyses for clouds [15, 42, 72] have been conducted to fill this gap. Similarly, existing reviews of data center network failures [63] have been extended with a special focus on cloud computing data centers [142]. Literature shows that causes for failures of services deployed on cloud infrastructures are very diverse, and the root causes include broken hardware, network errors, and software bugs. In addition, a high percentage of failures is caused by human error, misconfiguration and faulty upgrade routines. However, cloud platforms offer stronger fault handling mechanisms when compared to grid or cluster computing, partially due to the rapid elasticity and the possibility to migrate live VMs between physical machines [27]. Generally, in large and critical infrastructures, most bugs cause software to behave in a way that leaves reboot or restart as the only means of restoring the system [25]. This includes fault scenarios such as crashes, deadlocks, spins, or leakage of memory.

Researchers and software designers have devised many approaches to increase the dependability of applications. Fault tolerance on the software design level [145] greatly increases the resilience to unforeseen fault scenarios but is very costly and requires a lot of discipline from the developers. No system can assume perfect bug-free software, therefore higher-level mechanisms are used to stabilize applications. Specialized replication schemes can protect an application against so-called byzantine faults: arbitrary faulty behavior and communication [32]. The Recovery Oriented Computing (ROC) scheme focuses on reducing the "mean time to repair" of atomic infrastructure components, such as network switches, in order to greatly increase the overall availability [139]. A proactive strategy for protecting against

fail-stop faults consists of proactive overprovisioning triggered by low-level health-alerts [132]. The dependability of data storage can be increased through replication and a distributed integrity assessment scheme [22, 23, 166]. Similarly, the reliability of distributed computational jobs can be calculated and maximized even in the presence of propagating failures [33, 156]. The concept of *recursive restartability* [29] describes an application design that minimizes the impact of faults and therefore increases reliability and dependability of the application.

The listed approaches have in common that they solve individual problems on different system layers but fail to consider the full technology stack of a cloud infrastructure as a whole. Additionally, most academic approaches consider crash faults only, while ignoring other degraded states that impact the service quality. The following section covers more holistic approaches to IT operations.

3.3 AIOps Systems

The emerging paradigm AIOps extends traditional IT operations by methods from recent advancements in the field of machine learning. The main idea is to overcome current limitations in automation and monitoring systems through data-driven techniques. AIOps systems attempt to support the administrator by presenting digested information and by replacing the selection of repair actions and their trigger conditions with machine learning algorithms. The research and advisory firm Gartner, Inc. has coined the term *Algorithmic IT Operations* in 2014, which they later renamed to *AIOps*. The official definition is as follows:

“AIOps platforms utilize big data, modern machine learning and other advanced analytics technologies to directly and indirectly enhance IT operations (monitoring, automation and service desk) functions with proactive, personal and dynamic insight. AIOps platforms enable the concurrent use of multiple data sources, data collection methods, analytical (real-time and deep) technologies, and presentation technologies.” [62]

Even before the elasticity of cloud computing platforms, web services implemented concepts for automatic reconfigurations based on service quality and system load [101]. In the context of the cloud, auto-scaling is a commodity and has been addressed by numerous approaches [101, 103, 107, 108, 149]. The predominant approach is to use expert-defined thresholds for system resource utilization to trigger the addition or removal of new replicas. Some approaches use more advanced

models to predict the service workload and use it to infer resource utilization [149]. Based on above definition for AIOps, machine-learning-driven auto-scaling can be seen as a form of AIOps platform, albeit limited to a very specific part of overall IT operations.

The research community has only recently started to adopt the term AIOps [9, 43, 97, 110, 124]. However, this particular branch of the IT industry has seen an explosive growth, and new products, open source projects, and service provider companies continue to emerge. Gartner predicts that the percentage of large companies actively developing and deploying AIOps platforms will grow from currently 5% to around 40% by 2022 [9]. Numerous industrial products advertise AIOps capabilities.

As with any industrial solution, the products on today's AIOps market do not publish details on the underlying methodologies such as used machine learning models and data processing strategies. Furthermore, in this fast-moving industry, products are continuously extended by new features and methods. Moogsoft¹ offers solutions for ingesting monitoring data, reducing the number of alerts, finding the root cause of correlated incidents, and finally enabling collaboration for problem resolution through an integrated platform. The underlying algorithms include clustering and Long-Short Term Memory [79] for anomaly detection and data reduction, and temporal or topology-based correlation techniques for probable root cause analysis [127]. User feedback on incident refinement and resolution can optionally be incorporated to increase the future precision of the algorithms. However, the publicly available information does not mention automatic infrastructure-level incident resolution. The focus of the Moogsoft platform is to reduce and refine the data presented to system administrators, while relying on the administrators to use the presented information to quickly resolve the situation.

Other AIOps products have slightly different feature sets and focus on different aspects. FixStream² features a powerful auto-discovery of infrastructure and applications, but offers less analytical capabilities. BigPanda supports configurable levels of autonomy for first-level responses to technical incidents³. Other products such as DataDog⁴ focus more on application monitoring and provide integrations with large numbers of widely used services. The list of related companies and products can be continued, but to the best of our knowledge none of the currently

¹<https://www.moogsoft.com/product/>

²<https://fixstream.com/platform-capabilities/>

³<https://www.bigpanda.io/our-product/lo-autonomous-layer/>

⁴<https://www.datadoghq.com/product/>

available solutions claims to fully close the operations cycle by autonomically remediating unforeseen anomalies.

3.4 Cloud and Data Center Monitoring

Today's market offers a wide range of tools for monitoring data centers and cloud infrastructures. On a high level, there is a differentiation between general-purpose monitoring tools, and tools or services that specialize on cloud infrastructures. General-purpose monitoring tools usually consist of data collection agents and centralized components for storage and visualization. Examples include Nagios [17], Zabbix [136], Collectd⁵, and Ganglia [111]. These tools are usually open source and support monitoring infrastructures and services on both private and public infrastructures and clouds. Many tools offer some limited support for storage, analysis and visualization of the collected data, but additional specialized software is often used for these tasks. Most of these tools are not designed for immediate, low-latency processing of the collected data. Therefore, the collected data is recorded with high sampling rates of above one second and must be retrieved asynchronously from a database for further analysis.

Monitoring tools specialized on cloud infrastructures are either offered directly by public cloud providers, or integrate with specific Application Programming Interfaces (APIs) of public clouds or open source cloud operating systems. Examples include CloudWatch from Amazon AWS [8], AzureWatch⁶, Monitis⁷, LogicMonitor⁸, and PCMONS [44]. Since many of these tools are commercial and not open source, they usually offer rich visualizations and dashboards, but varying data analysis and alerting capabilities.

Surveys of the cloud monitoring field report that only a small number of monitoring platforms and services are able to monitor service dependencies [52] or autonomously act in case of anomalies and fault situations [3]. Along with deeper statistical analysis of the data, these capabilities are required for a self-healing cloud platform. Monitoring in cloud computing systems has special requirements due to the multitude of system layers and stakeholders, and high elasticity and dynamism of the overall system [38]. Data can be collected on the physical infrastructure layer, and on various levels of the virtualization and service software. Furthermore, the service provider and the clients have different requirements and

⁵<https://collectd.org/>

⁶<http://www.paraleap.com/azurewatch>

⁷<http://portal.monitis.com/>

⁸<http://www.logicmonitor.com/>

views on the data [125]. More specialized approaches in the field of cloud monitoring target monitoring specifically for enforcing security properties [66], or to associate monitoring data from multiple distributed data sources with a single client request [85].

Newer research emphasizes the growing complexity of data centers and recommends to integrate data from supporting systems, such as cooling, lighting, or power supply, with higher-level data collected from the actual software components into a full-stack data analysis [98, 99]. The domain of data center monitoring generally covers deeper parts of the overall infrastructure than monitoring of the software and services [129]. However, the state-of-the art for analyzing this monitoring data is still based on predefined thresholds and alarms [5].

Aside from infrastructure-level and network-related metrics, data from the application-level delivers important insights about the health of the running service. Application-specific metrics include request latency, error rate, throughput, or the latency of downstream requests. For comparability, some of these metrics can be normalized by job size. Several approaches suggest using such application-specific data for optimizing various aspects of cloud-based services, such as scale-outs or the service quality [38, 144, 157]. However, relying on application-specific metrics is not generic and is only applicable in a subset of real-world scenarios. A more generic approach is to use resource utilization data from the guest operating system to infer the state of hosted applications [49]. This approach still requires an agent with elevated privileges running inside all guest VMs.

3.5 Data Analysis Platforms

A major part of our self-healing cloud architecture consists of analyzing monitoring data obtained from the supervised system. This data analysis has certain requirements that increase the responsibility and resilience of the entire self-healing cloud platform, such as low-latency processing of data streams, scalability, and low resource consumption. Many academic approaches and industrial products are capable of analyzing data streams, but only few meet these requirements.

The research direction of *Big Data* focuses on analyzing data with volumes that are not possible to process on a single machine [58, 115, 179, 187]. The characteristics of Big Data are described by the “HACE” Theorem: Big Data starts with large-volume, **h**eterogeneous, **a**utonomous sources with distributed and decentralized control, and seeks to explore **c**omplex and **e**volving relationships among data [58].

Platforms for Big Data implement different programming models with varying fea-

tures and levels of flexibility. Googles MapReduce platform [36, 46] implements massively parallel execution of the well-established map-reduce model which allows data transformation through a user-defined `map()` operation and subsequent aggregation through a user-defined `reduce()` operation. This simple abstraction allows users to express and implement a wide variety of algorithms, such as the K-Means clustering algorithm [186]. Other widely used platforms support more general data processing workflows on batches of data (e.g., Spark [181]), or both batch and stream-wise processing (e.g., Stratosphere [6] or Asterix [18]). Such platforms usually offer high abstraction levels for data processing. On the one hand, these abstractions allow concise representation of the data analysis operations, and on the other hand they allow the platform to optimize the underlying transportation and computation of the data. Other platforms are optimized for graph mining algorithms [88]. All big data platforms have in common that they assume complete ownership of the underlying compute resources. This assumption enables many built-in optimizations, such as colocation of data and processing operators, or optimized scheduling of the job processing graph.

Astrolabe [171] is a hierarchical system for storing and aggregating highly distributed data records. All participating nodes form a distributed tree structure, where dynamic data records are stored in the leaves. Users of Astrolabe submit an SQL-like query syntax to describe a desired aggregation of the data. The query is propagated downwards through the tree, and the aggregation results are then computed in parallel and returned to the user. The design of Astrolabe is based on a randomized peer-to-peer protocol for robustness. The hierarchical and scalable approach of Astrolabe is similar to the data analysis architecture proposed in this thesis, but limited to aggregations, and not directly applicable to continuously updated streams of data. Compared to big data platforms, however, Astrolabe offers more potential for co-existence with other workloads.

Intanagonwiwat et al. have proposed *directed diffusion*, a communication paradigm for scalable data aggregation, where the data and the processing workload is hierarchically distributed [84]. While the main use case for directed diffusion is communication within sensor networks, the underlying concepts are similar to Astrolabe and generally applicable to any distributed system. The main difference between directed diffusion and Astrolabe is that the former is specifically designed to handle data streams, which could be emitted by a surveillance camera or other sensors. The TAG (Tiny AGgregation Service) [105] system builds on similar concepts as directed diffusion, but offers a simpler, declarative interface for describing user queries.

Amongst the proposed platforms for monitoring or managing computer systems, only few include a data processing strategy specifically designed for this use case. Moore et al. propose Splice, a knowledge plane for data centers that couples static and dynamic metrics of various infrastructural elements with physical meta information, such as location and energy consumption [128]. Splice relies on a relation database for storage and query-based analysis of collected data. The data collection itself is designed for extensibility and scalability, but ultimately all data must be transported and stored in a centralized storage system, which limits scalability and data analysis latency. Dean et al. propose to use *residual* resources for analyzing data that is collected on the hypervisors of a cloud data center. The analysis is performed by VMs with special scheduling constraints to avoid over-utilization of the hypervisor resources by the data analysis. These computational VMs are live-migrated to other hardware when utilizing too many resources. This approach of residual data analysis is similar to the in situ data analysis strategy presented in this thesis, but suffers from very coarse-grained scheduling, because each computational VM is responsible for *all* data gathered on one hypervisor.

Chapter 4

Zero-Touch Operations

Contents

4.1	The Zero-Touch Operations Loop	35
4.1.1	Zero-Touch Across System Layers	38
4.1.2	Fault Model	40
4.2	Detailed Architecture	43
4.2.1	Modeling the System Topology	47
4.3	Data Analysis Pipeline	51
4.3.1	Overview	51
4.3.2	Anomaly Detection	55
4.3.3	Anomaly Classification	60
4.3.4	Root Cause Analysis	62
4.3.5	Remediation Workflow Assessment	63
4.3.6	Controlled Level of Automation	67

Distributed systems suffer from various causes for potential outages or reduced service quality, including software aging, resource leaks, version mismatches, and hardware problems. These problems are exacerbated in modern distributed systems, which are usually built on a deep technology stack and include multiple layers of virtualization. System administrators, that are responsible for the design and maintenance of critical computer systems, carefully select and test the components, their versions and compatibilities. However, performance anomalies, overload situations, and failures of individual components can never be entirely eliminated in the design phase of a computer system. The sheer number of interacting entities causes unforeseen situations that need to be resolved to ensure a high level of service quality.

When a task, which is traditionally performed by a person or a team, exceeds human capabilities, the natural solution is to implement automation. Automation plays an important role in the field of network and system administration. Depending on the number and nature of automated tasks, the function of a system can range from supporting humans to being fully self-organizing and self-maintaining. Typical automation systems consist of binding trigger conditions derived from monitoring data to certain actions. These conditions are static; examples include predefined thresholds for resource usage, the results of a service-layer probe, or network latency measurements. Actions that are bound to such trigger conditions range from simple notifications, over auto-scaling, up to more complex workflows that include migrations, version updates and reconfigurations. Such automation tools suffer from the following limitations:

- Trigger conditions, including threshold values, must be chosen manually,
- automation is limited to a set of predefined actions that do not cover unforeseen situations, and
- automated actions usually recover from a situation, instead of preventing it (reactive instead of proactive).

We propose the concept of *zero-touch operations* as an extended automation platform that mitigates these limitations. The term zero-touch does not have a widely accepted and well-defined meaning in the computer science community and IT industry. Intuitively, it means that human interaction (or human “touch”) with a system is not only reduced, but entirely eliminated. In practice, of course, a computer system that operates without *ever* requiring *any* human interaction is neither possible nor desired. Therefore, we consider techniques and approaches under the umbrella *zero-touch* as efforts aiming to reduce human interaction with a computer system. We define *zero-touch operations* as the capability of a computer system to autonomously manage itself and continuously uphold predefined operational goals, even in the presence of anomalies and faults.

The objective of the zero-touch operations system has large overlaps with self-healing in autonomic computing. In order to continuously uphold operational goals, any anomaly or fault must be detected and handled, which is a form of self-healing. Therefore, we use the terms zero-touch operations and self-healing interchangeably in this thesis. A successfully running zero-touch operations system makes regular human interaction with the platform unnecessary. An administrator merely defines the *goals* of the platform, which is usually to execute a given set of services within given service quality constraints. Further interaction is only necessary when these goals must be changed, or when the system fails to resolve

a situation automatically.

The remainder of this chapter introduces our architectural concept that serves as a blueprint for a practical self-healing cloud system and strives to fulfill the “zero-touch” promise. In Section 4.1 we give an overview over the zero-touch architecture, discuss how it can be applied on different layers of the cloud stack, and define a fault model. Section 4.2 covers the architecture in more detail. Finally, Section 4.3 describes the data analysis tasks and algorithms that play a central role in the system design.

4.1 The Zero-Touch Operations Loop

Figure 4.1 visualizes the high-level control loop of our zero-touch operations system.

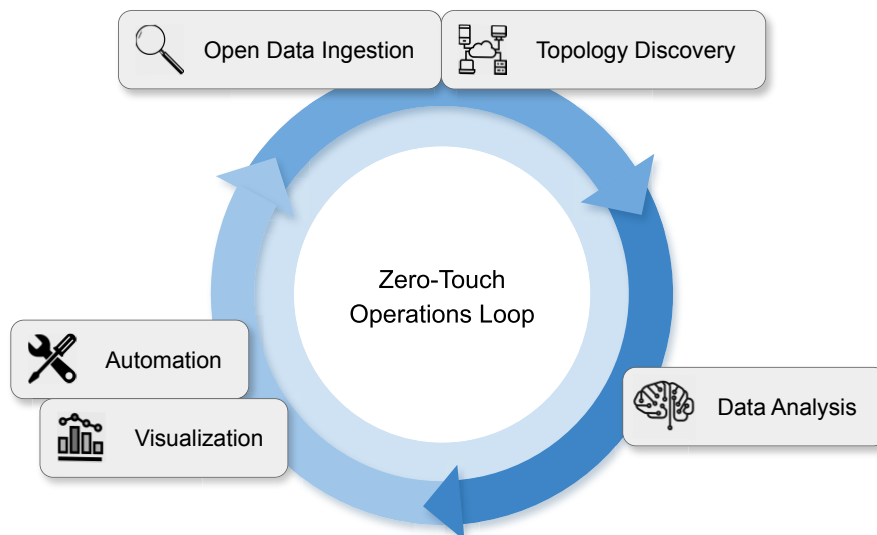
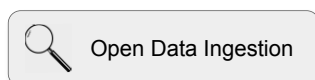


Figure 4.1: High-level view of the zero-touch operations control loop

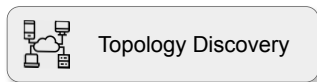


Open Data Ingestion

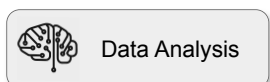
The *Open Data Ingestion* component is the basis for extracting knowledge about the supervised systems, and for any automated reasoning and decision making.

Cloud platforms usually include a variety of monitoring systems, out of which we prefer those, that produce low-latency and high-frequency time series data. Other types of data can be leveraged as well, such as event or logging data. As with any type of data analysis, the success of the later algorithms

highly depends on how well the modeled data represents the observed system. The more detailed the monitoring data, the more knowledge can be distilled from it.



Besides ingesting data, the zero-touch platform depends on a dynamic and up-to-date view of all system components and their dependencies. This task of *Topology Discovery* captures all accessible system layers, and the vertical and horizontal dependencies between the components. These dependencies are initialized and continuously updated from a number of data sources. Managers of virtualized resources are able to provide this kind of information. The cloud operating system knows where it schedules its virtual machines, and what virtual networks they are connected to. A management system for containers (such as Kubernetes) has similar information, but on another layer of the technology stack. We mine these information sources and combine them into one knowledge repository.



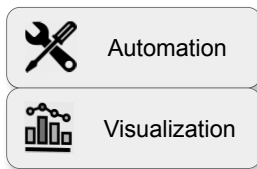
The *Data Analysis* component of the zero-touch control loop consists of algorithms that ingest the collected data in real-time and produce more high-level information about the state of the monitored components. The high-level information includes detected and predicted performance anomalies, localization of the anomaly root cause, and the selection of an appropriate remediation workflow to handle the anomaly situation. Cloud platforms and applications can exhibit very dynamic and unpredictable behavior, which is exacerbated by the high dimensionality of the analyzed data streams. Therefore, we leverage advanced statistical methods, and techniques from the field of machine learning, in order to represent such data streams and extract the knowledge contained in them. To produce accurate results, the algorithms continuously learn and improve their models of the data. Machine learning algorithms that work in a supervised or semi-supervised fashion obtain the ground truth from two sources:

1. The feedback loop through the streamed monitoring data (including domain-specific service quality measurements), and
2. input of a human expert.

The monitoring data contains information about the success of a remediation workflow, which allows to draw conclusions about the performance of all data analysis tasks. Algorithms for anomaly detection are expected to operate fully unsupervised by building a baseline model of the observed data and signaling deviations from it. Occasionally, a human expert might intervene and manually flag an anomaly that

remained undetected, or, on the contrary, notify the system about an expected anomaly situation in case of a software update, reorganization or migration of services, or similar external conditions. The process of notifying the self-healing platform about external conditions can also be automated. For example, a Continuous Deployment (CD) server knows about planned version updates and can cooperate with the zero-touch platform to avoid false anomaly alarms during update periods.

An important aspect of the Data Analysis component is that most of the processing is done directly on the hardware resources where the data is collected. The opposite approach, which relies on dedicated external compute resources for the analysis, leads to overhead in terms of network traffic and impacts data analysis latency. To achieve a scalable execution, yet extensive and precise results, the data analysis runs in multiple stages that build atop one another. Lower analysis stages cover individual components, while higher-level stages cover groups of components and finally the entire system. Since lower stages forward intermediate data analysis results to higher stages, the data granularity decreases progressively, while the analysis context is enriched through additional components. The richer analysis context allows to perform root cause analysis and to correlate alarms from multiple components in case of anomalies that propagate through the system. Chapter 5 introduces the design of the *in situ* data analysis engine that meets these requirements.



In the last component of Figure 4.1, *Visualization & Automation*, the self-healing functionality interacts with the monitored system and human administrators. The selection and execution of automated remediation actions closes the control loop that started with the monitoring of the cloud platform. The algorithmic challenge of the *Automa-*

tion component is to learn the most effective remediation workflows when encountering anomaly situations over time. The execution of such workflows must be strictly controlled in order to avoid further deterioration of the anomaly situation. The workflow execution engine provides important feedback to the workflow selection, including the execution time of the workflow and whether the workflow was executed successfully. The remediation workflow selection uses the feedback data to learn and improve the selection process. A User Interface (UI) component gives visual insights about the decisions of all involved algorithms, and allows administrators to insert their own expertise to improve the models.

The set of available remediation actions, and the flexibility of how they can be combined, controls the *level of automation* of the self-healing system. Different

deployment strategies favor different levels of automation. The most conservative configuration disables any intrusive remediation workflows and limits the platform to the visualization of suggestions and data analysis results. This configuration eliminates many benefits of self-healing. Intermediate levels of automation allow the automatic execution of selected remediation workflows when the decision making process reports a high confidence in the results. A high confidence value means that there is a high probability to make the optimal decision in the current situation. Finally, the highest level of automation gives the system complete freedom over the selection and execution of remediation workflows. The key aspect to a practical zero-touch operations system is comprehensive control over the level of automation.

4.1.1 Zero-Touch Across System Layers

The exact layout of the zero-touch architecture depends on the system layers it manages. The more layers and components are integrated with the self-healing system, the more data can be analyzed and the better the remediation workflows can be selected and coordinated. In other words, a more holistic view of a system enables a more complete closed-loop control. In this thesis, we assume that a provider offers cloud services on different abstraction layers to its customers, who execute their own software on top of the provided cloud stack. Figure 4.2 shows a selection of cloud services typically offered by modern cloud providers. The higher the level of the offered service, the more parts of the technology stack are managed directly by the cloud provider.

In traditional IaaS platforms, customers usually configure virtualized compute nodes, storage resources, and networks, and deploy their own applications on top of these resources. This makes it challenging to collect performance metrics and gain insights from the services owned by the customer. However, modern cloud platforms offer more high-level abstractions that allow customers to describe their application as a graph of replicated containers, which are scheduled and maintained by the cloud provider. Going even further, serverless Lambda computing is a paradigm that gives the cloud provider complete freedom over the execution of the customer's code. Here, customers submit their applications in the form of numerous small source code or deployment artifacts. In such scenarios, the cloud provider “owns” the entire system stack from the bottom to the very top and has access to a big variety of monitoring information and remediation workflows.

We envision a service model where the customer has the option to book self-healing capabilities for cloud services of any layer. The cloud service provider, on the other

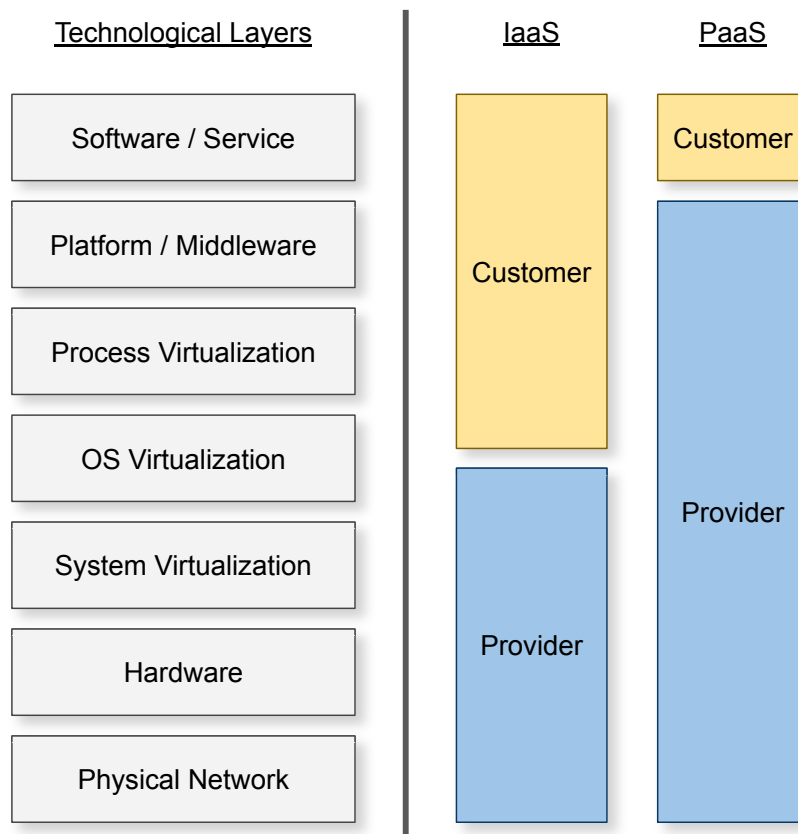


Figure 4.2: Layers of a cloud system and associated cloud services. Note: OS virtualization (also called container virtualization) in the IaaS model is commonly encountered both on the provider and on the customer side.

hand, benefits from the zero-touch operations system for all technology layers that are not directly managed by the customer.

For the sake of simplicity, the remainder of this chapter refers to the customer-owned part of the technology stack as the *service layer*, regardless of the actual cloud service model. In the traditional IaaS model, the *service layer* equals the entire operating system and software inside of virtual machines; in the PaaS model it refers to software running on the provided platform; in the serverless computing model, the *service layer* is limited to compact software modules managed entirely by the cloud platform.

Technically, the zero-touch operations architecture is not limited to the scenario of a cloud provider offering services to a customer. On the one hand, the zero-touch

paradigm is also applicable to an entirely self-hosted and self-managed computer infrastructure, from the hardware up to the service software. This scenario gives the data analysis algorithms the most holistic view on the system and the most freedom when composing remediation workflows. It is, however, increasingly rare in practice, due to the success of the public cloud service model. On the other hand, customers of cloud services can opt to deploy their own self-healing system running entirely outside of the scope of the cloud provider. This confines the zero-touch functionality to the customer-managed parts of the technology stack, while the cloud provider is entrusted with managing the lower layers only.

One important property of cloud services is that the concern of providing the cloud infrastructure, and the concern of maintaining the service layer software, are distinct and can be handled by unrelated parties. This assumption, however, does not always hold in practice: The layers of a cloud architecture are highly interdependent and some anomaly scenarios can be remediated or prevented by collecting and analyzing monitoring data across all system layers. Due to these complexities, a cross-layer self-healing system promises to be more effective. Examples for problems that span over multiple layers include misconfigured hypervisors that reduce the performance of the service layer, or VMs that overutilize a virtual resource and starve co-located VMs.

Although a self-healing system can integrate with different parts of the technology stack, the basic principles of the architecture and operations remain the same. While available data sources and possible remediation workflows change, the components and interfaces binding the two together, can be reused.

4.1.2 Fault Model

In this section we categorize the types of faults that the zero-touch operations system aims to handle. In other words, we define the *fault model*. Avizienis et al. provide an extensive framework to categorize faults and failures [14]. We utilize this classification framework to increase the comprehensiveness of our fault model. The *effect* of a fault is a failure. Avizienis et al. define three big classes of failures, which are service failures, development failures, and dependability failures. The focus of the zero-touch system is the autonomic operation of a service, therefore we do not handle development failures. Dependability failures are defined as causes of service failures, therefore we consider service failures only.

Since the actual impact of a service failure is highly specific to the respective system, the categorization of such failures is rather abstract. The main differentiation

is the *domain* of a failure, which can be one of the following:

Content failure: The content of the information delivered by the service deviates from its specification.

Timing failure: The response time or another timing aspect of the service deviates from its specification.

Halt failure: The system's activity can no longer be perceived by clients – i.e., the system becomes unreachable or unresponsive.

Erratic failure: The service does otherwise not comply to its specification, e.g., by sending information without being requested to do so.

Many application-specific mechanisms exist to detect and recover these failure domains. Zero-touch functionality operates in a service-agnostic fashion and does not aim to replace these failure handling mechanisms. Instead, the goal is to *prevent* failure conditions by detecting changes in the internal behavior of an application, before it results in an externally visible failure. We call such behavioral changes *anomalies*. The main precondition to detect anomalies is that they manifest in observable data. Therefore, an autonomic self-healing system cannot prevent failures that are not preceded by any observable behavioral changes whatsoever. In case of timing or erratic failures, the failure condition can continue long enough to allow data-driven detection and recovery. In case of halt failures, service-specific mechanisms must subsequently recover the situation. Therefore, the zero-touch operations system complements such traditional dependability measures, instead of completely replacing them.

Besides a categorization of failures, Avizienis et al. define a classification framework for faults. Figure 4.3 lists the eight elementary fault classes that allow to classify the *origin* of a fault based on a number of criteria. The classes marked in red are the ones that are explicitly out of scope of the zero-touch operations system. In the remainder of this section we explain each decision, and how the excluded fault classes should be handled in practice.

Regarding the *Phase of creation or occurrence*, the zero-touch operations platform is mainly concerned with faults that occur during the operation of a system. However, the origin of such faults often lies in the development phase, such as with software bugs that are introduced by developers and remain dormant until their activation at run-time. Self-healing explicitly aims to remediate situations caused

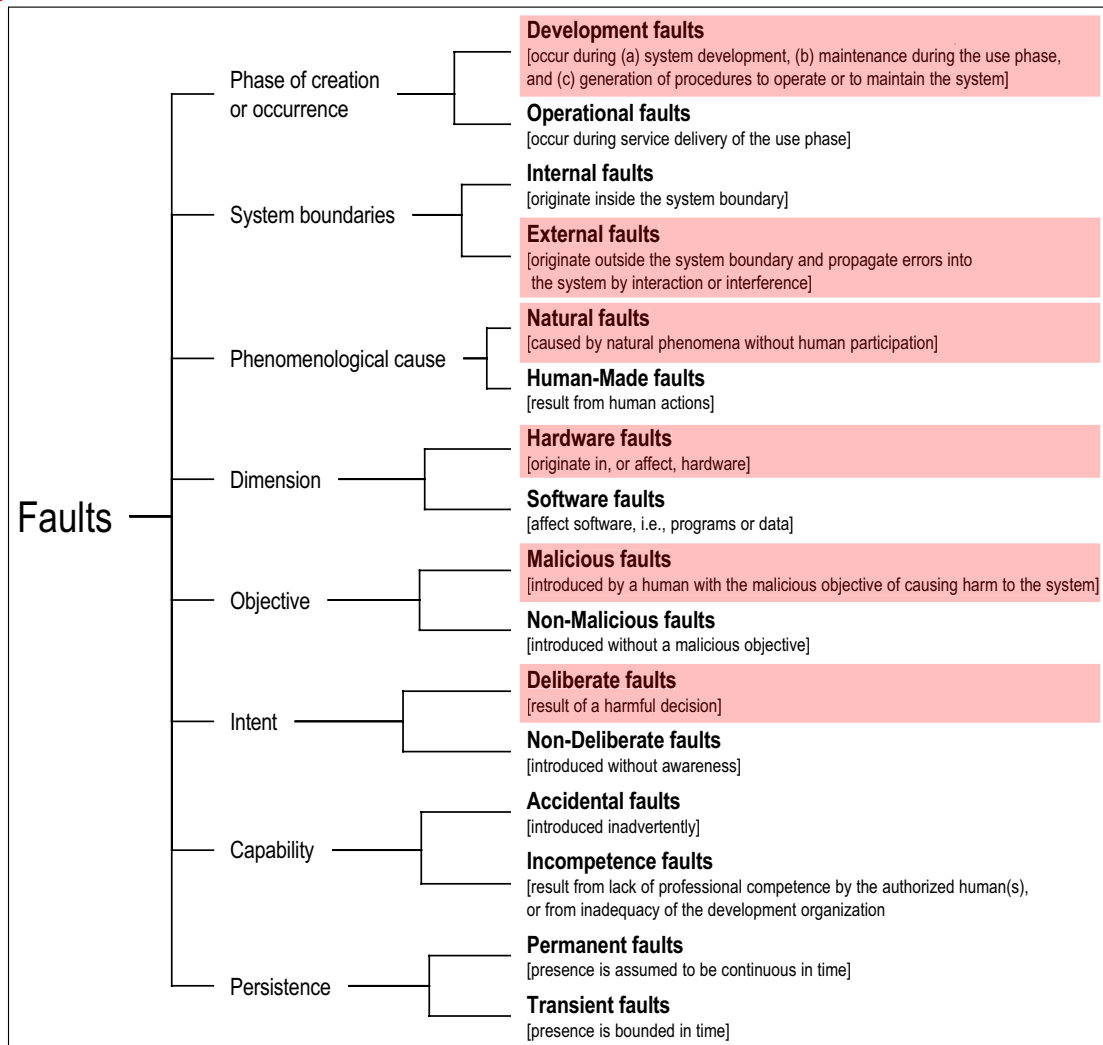


Figure 4.3: Elementary fault classes defined by Avizienis et al. [14]

by software bugs, but possible countermeasures such as reboots, simple reconfigurations, or version rollbacks, can never completely eradicate the bug: That must be done by human developers. The platform plays a supportive role in that task by pointing out bugs in versions of software components, where an anomaly situation keeps reappearing regularly. Avizienis et al. extend the class of *Development faults* by including faults that originate from system maintenance (see “(b)” and “(c)” in Figure 4.3). Such faults are hard to distinguish from regular operational faults, and we therefore consider them as operational faults that are covered by

self-healing.

External faults are out of scope for the zero-touch operations platform, since communication with external systems is application-specific, and related error handling must be built directly into the application software. Similarly, situations resolved by a self-healing platform are directly or indirectly caused by humans. Natural phenomena such as physical deterioration or natural disasters cannot be fully resolved without human intervention. The same mostly applies to hardware failures. Techniques like masking the outage of a hard disk are out of scope, as described above.

Malicious and *deliberate* faults lie in the domain of computer security and are handled by systems like firewalls or IDSs. Just as with traditional techniques for enhancing availability and reliability, we consider our zero-touch operations platform to be complementary to security systems, instead of replacing them.

From the point of view of the zero-touch operations platform, there is no reason to distinguish *accidental* from *incompetence* faults, since the impact of the fault remains the same. Finally, both *transient* and *permanent* faults are handled, because both fault classes can potentially manifest in the observable behavior of the application.

This section described how a self-healing platform complements other techniques for ensuring dependability and security. This multitude of management components introduces complexities of its own, such as potentially contradicting recovery routines or management decisions. Currently, this problem must be solved through careful engineering, but in the future such complementary systems might merge into one instance that manages all dependability, autonomicity, and security aspects of an application. Such a symbiosis would be one further step towards truly autonomic computing.

4.2 Detailed Architecture

The zero-touch operations architecture is an extension to the classical cloud infrastructure. Typically, a human administrator is responsible for receiving alarms, understanding and investigating them, and finally acting on them, when necessary. The components of the zero-touch architecture mimic this workflow and make use of machine learning techniques wherever it is necessary to learn from historic data or human input. Figure 4.4 shows how the components of the zero-touch architecture interact with the traditional cloud infrastructure.

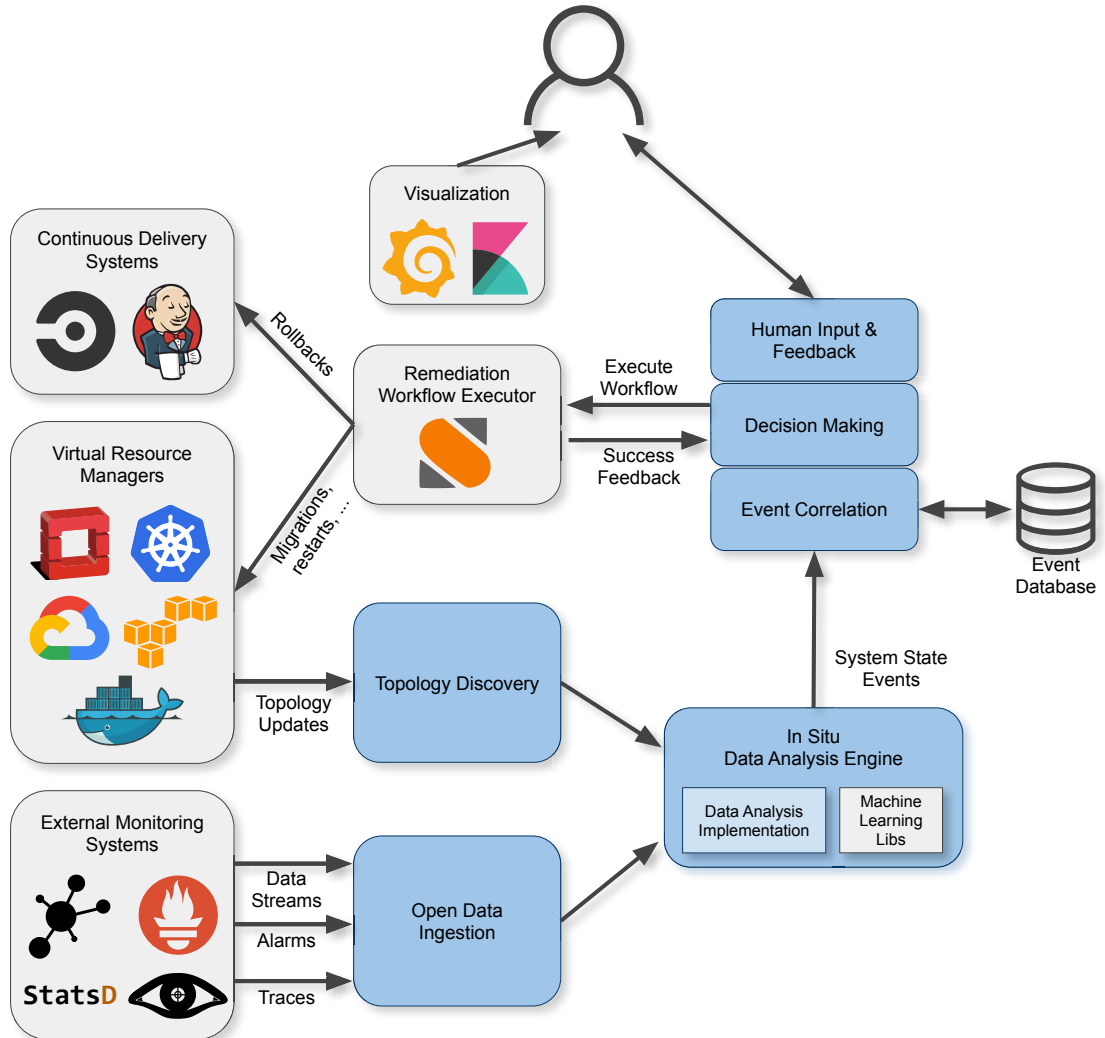
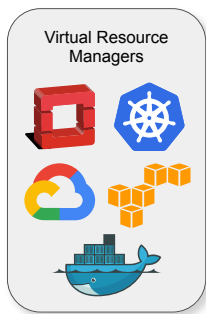
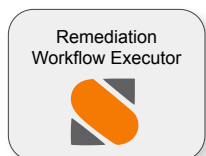


Figure 4.4: Interactions of zero-touch operations platform components with the cloud infrastructure. The zero-touch components are placed on the right side of the diagram in a different color than the cloud infrastructure on the left side.



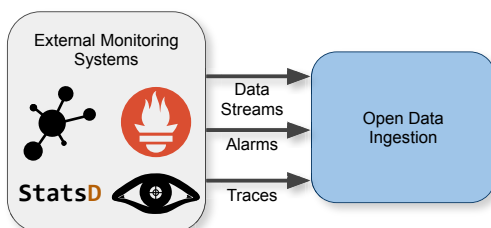
The core of the cloud platform are the *Virtual Resource Managers*. This umbrella term denotes all public and private cloud technology providers and their respective services. The cloud services span over all system layers, from simple virtual machines, up to domain-specific software solutions. Examples for Virtual Resource Managers suitable for private clouds include OpenStack, Docker and Kubernetes, while public cloud providers include Amazon AWS and the Google Cloud Platform. Users of such cloud services typically rely on continuous

delivery systems such as Jenkins to automatically build, test and deploy their systems. An interaction with the continuous delivery system allows to automatically roll back to previous versions of the user system in case of detected anomalies or repeated failures.



An important component for the zero-touch architecture is the *Remediation Workflow Executor*. Its responsibility is the reliable execution of remediation and/or repair actions on the monitored resources. Several systems, such as StackStorm¹, are openly available to fulfill this task. Depending on the underlying cloud

technology, different remediation workflows are available. For example, when working with a bare IaaS platform, the VMs can be restarted, scaled up or down, or migrated to a different hypervisor or cloud region. Similar actions are available when the user application is running in containers instead of VMs. Working with higher-level services enables service-specific remediations, such as service restarts, reconfigurations, or changes in the load-balancing strategy. An open execution engine such as StackStorm allows users of the zero-touch operations platform to implement user-specific remediation workflows. This increases the platform's flexibility.

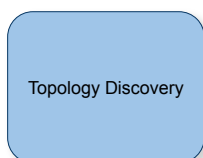


The bottom left corner of Figure 4.4 symbolizes the main sources of information used by the zero-touch components. Many monitoring systems are available and widely used in existing cloud infrastructures. The type and properties of the data sources have a big impact on the per-

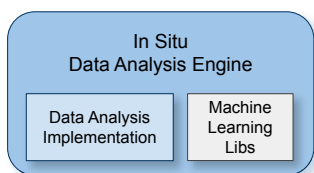
formance of the data analysis and remediation action selection. Real-time data streams can be analyzed with low latency, in order to deliver fast, high-frequency

¹<https://docs.stackstorm.com/overview.html>

results. This allows to rapidly detect and remediate local anomalies that do not require further offline processing and have a limited scope. To guarantee good analysis results, the data must actually contain a valid representation of the system state. However, a richer data stream also leads to more overhead when transporting and analyzing the data. Besides time series data, other types of data include events and traces. Events, also called alarms, are produced by monitoring systems and typically rely on fixed thresholds for metrics, which are predefined by human administrators. A big problem in traditional data center monitoring systems is the large number of alarms that are produced and must be handled by administrators. Trace data consists of application logs or high-frequency events that constitute state transitions of the application, including the timing characteristics of said transitions. The *Open Data Ingestion* converts all types of input data to a common internal data format, which can be analyzed coherently.



The second source of information is a dependency model of the application and infrastructure topology. In most cases, the interconnections between different system components are so numerous, that it is not possible to manually create and maintain such a model. The automated topology discovery component locates and aggregates dependency information from multiple sources and represents them in one consistent graph structure. The resulting graph structure allows to query for neighbors, cliques or similarities between components, which are used by different data analysis algorithms, but mainly for root cause determination. Section 4.2.1 describes how dependency information can be obtained automatically, and how it is represented in a graph structure.



The *in situ data analysis engine* orchestrates distributed data analysis processes, which in turn consume the available data to learn and to produce knowledge about the state of the observed system. The *in situ* property of the data analysis means that data is not transported to a dedicated processing infrastructure, but is instead processed as close to its origin, as possible. Other requirements of this data analysis engine include scalability to support analysis of large numbers of data streams, extensibility to cope with rapid advancements in the field of machine learning, and bounded resource usage to minimize the impact on the actual cloud workload. Chapter 5 describes the design of this data analysis engine, which is the second contribution of this thesis.

The actual algorithms performing data preprocessing, anomaly detection, root

cause analysis, and finally decision making, are subject of active research. Section 4.3 proposes a portfolio of data analysis techniques that fulfill the requirements of the zero-touch architecture. Our basic approach is the following: An efficient anomaly detection is performed per atomic system component, then a more resource-intensive anomaly classification categorizes the situation, before a multi-layered Root Cause Analysis (RCA) pinpoints the origin of a propagated anomaly. Finally, all available remediation workflows are assessed based on collected knowledge from workflow executions in the past.

The data analysis pipeline produces system state events that contain all information needed to select a proper remediation workflow. This includes the following information:

- The target component for a remediation workflow
- A confidence value that describes the aggregated probability of correct decisions by all data analysis steps
- Assessment scores for each workflow that describe its quality in the given situation

The latter two pieces of information allow the administrator to configure the system's level of automation as desired. Section 4.3.6 describes the detailed configuration mechanism behind selecting and executing remediation workflows.

4.2.1 Modeling the System Topology

A good understanding of the monitored system is crucial for any self-healing architecture. After detecting the anomalous state of one or multiple components, it is important to analyze their dependencies, group correlated anomaly events, and determine their probable root cause. This requires detailed information about the connections and dependencies between the system components. Besides the data analysis, the system topology is also used by other parts of the zero-touch architecture. When optimizing the scheduling of data processing tasks, the data analysis engine requires information about the network proximity between data sources and potential data processing hosts. When planning and executing remediation workflows, the execution engine requires knowledge about the dependencies of hypervisors, virtual machines and services. For example, when the selected remediation action is to restart a hypervisor node, all VMs and services running on it are migrated to other nodes before executing the restart. Other workflows that benefit from detailed topology information include reconfiguring a load balancer, scaling a service up or down, and a coordinated rollback of the version of a software

component.

The collection of topology and dependency information must be automated. Depending on what system layers are managed by the zero-touch architecture, the number of components and dependencies can be very high, and the resulting dependency graph very dynamic. Automated *topology discovery agents* run on relevant parts of the infrastructure to detect updates and push them into a logically centralized graph database. This database must be distributed, scalable, and accessible by all zero-touch components. Despite the automated run-time updates to the topology database, the stored model cannot be assumed to always perfectly represent the current system state. When a topology discovery agent detects a change in the system topology, it must process the new information and store the resulting update in the database, which causes a delay.

Due to the logically centralized nature of the topology database, the number of updates that can be performed per time interval is limited, and only data with a certain longevity is stored there. Metrics that change with high frequency, such as the throughput or latency of a network connection between two services, are covered by the real-time data streams produced by monitoring systems.

Figure 4.5 shows an excerpt of a topology graph covering hypervisor nodes, virtual machines and containers running in the VMs. The graph is *directed* and the nodes and relations are annotated with key-value properties that help data analysis processes navigate through the graph. The shown example graph is stored and visualized by Neo4j, a graph database that is used by the prototypical ZerOps platform presented in Chapter 6. The visualized nodes represent hypervisors, VMs, containers, as well as the dependencies between them, within an actual testbed containing the prototype.

The basic system layers form a directed, acyclic graph, i.e. a tree, through relations like “runs-on” or “is-divided-into”: A cloud site is divided into availability zones, which contain several hypervisor nodes, which host a number of VMs, which execute containers, and so on. These relations form the *vertical dependencies*.

Adding *horizontal dependency* information breaks the acyclic property of the graph. Horizontal dependencies mainly represent network connectivity between hosts on different layers. On the hypervisor layer, physical networks and virtual overlay networks form a network topology that is reflected in the dependency graph. On the VM layer, there are two types of connections. On the one hand, virtual tenant networks form a topology similar to physical networks. This topology is usually rather static and changes only when components are added or removed from the

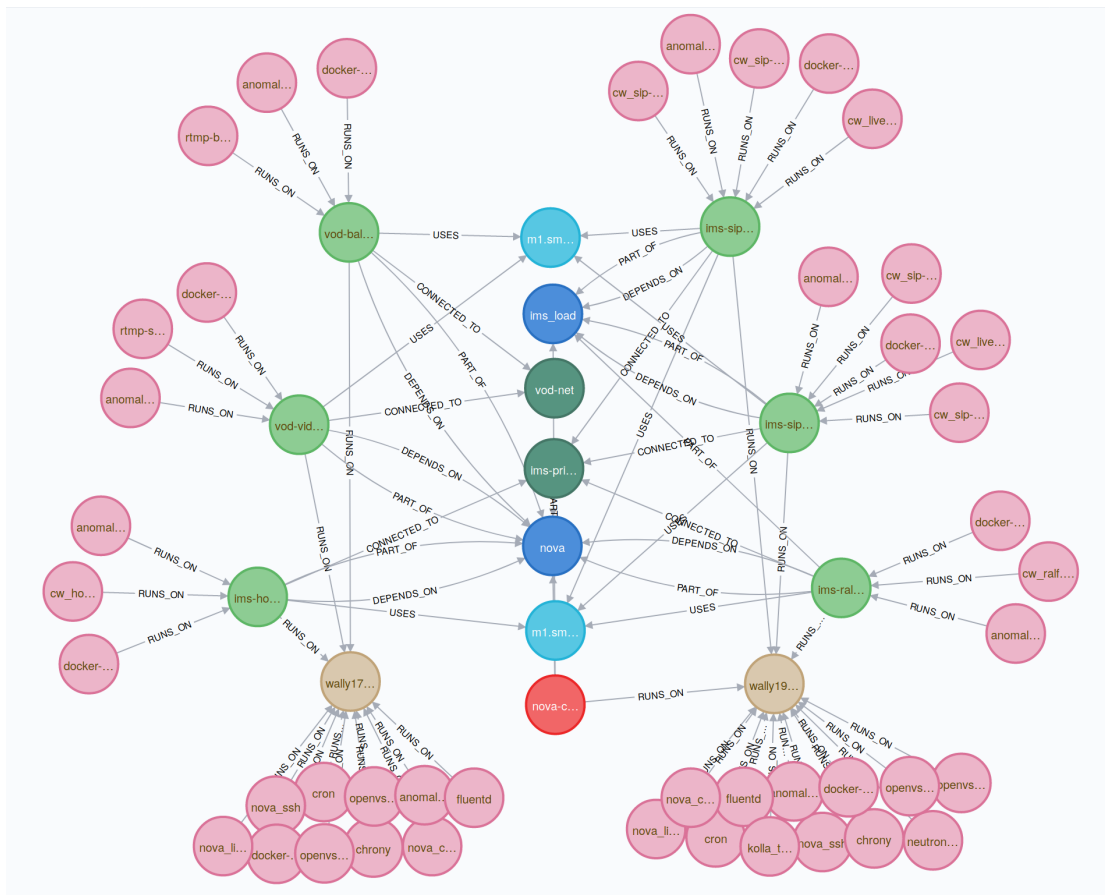


Figure 4.5: Excerpt of a topology graph. Brown: hypervisors, light green: VMs, pink: containers. In the middle: virtual networks and availability zones.

deployment. On the other hand, *actual* communication patterns between services indicate how the services depend on each other. This part of the topology is more dynamic and resembles a call graph between the different services and their running instances.

The various parts of the dependency graph are based on information from different sources:

Vertical dependencies between physical and virtual hosts, containers, and service instances are obtained from virtual resource managers, such as cloud operating systems. In order to evenly balance the workload and deliver a high-quality experience,

rience to the users, the cloud operating system maintains a clear mapping between hypervisor nodes, virtual machines, and higher-level services like PaaS offerings, containers, virtualized network functions, and so on. This dynamically updated mapping forms the base of the dependency graph. Any additional information provided by the cloud operating system is annotated within the graph, in the form of properties attached to nodes or relations. These properties include the allocated resources for virtual machines, the VM state and age, as well as any available meta information about the hypervisor nodes.

The physical network topology forms the horizontal dependencies between hypervisor nodes. Inference of network topologies is an ongoing field of research [74]. Existing topology inference tools for local networks are generally based on standardized tools like `traceroute` [86, 116]. Modern Software Defined Network (SDN) based networks are managed by a “network operating system” that contains a complete view of the entire physical network. Similar to querying the cloud operating system, this management entity delivers the most holistic view on the network topology. In case of faults on the network layer, additional diagnostics and topology inference tools are necessary.

The virtual network topology horizontally connects virtual machines. Users of IaaS cloud platforms directly or indirectly manage their virtual tenant networks. When using high-level PaaS offerings, networks are usually automatically configured by the cloud provider. In both cases, the cloud operating system always delivers an up-to-date view on the virtual network configuration. However, while this information gives an idea about what VMs and application have the *possibility* to exchange packets, it does not provide an accurate view on the *actual* communication topology. More detailed horizontal dependencies can be obtained by analyzing virtual **firewall access rules**. The best practice for firewall rule configuration is to be as restrictive as possible: In the optimal case, only expected traffic is actually allowed. Such firewall rules often include filters based on the communication partner - e.g., by virtual subnet or port number. Analyzing such well configured rules results in a more detailed service dependency graph without having to access and parse actual network traffic. An example for a firewall service that supports such an analysis are security groups in the OpenStack cloud operating system.

The network call graph Network traffic analysis is one way to obtain actual communication patterns between services. Deep Packet Inspection (DPI) is a technique to extract detailed information from captured network packets. While this technique can deliver a complete and continuously updated picture of the

network communication patterns, it has several drawbacks and is rarely used in productive systems. The main difficulty is the resource demand for analyzing high bandwidth network streams. This problem can be mitigated by sampling from the network, instead of analyzing *all* packets, but for complex virtualized networks this delivers a rather incomplete picture. Another technique for obtaining the actual network communication graph is to query the SDN switches that make up the virtual network fabric. Physical and virtual switches can be instructed to store high-level statistics about traffic on the network flows that they transport. While not acceptable in all deployments, this approach provides the necessary information with manageable overhead.

4.3 Data Analysis Pipeline

This section presents the design of our algorithm pipeline that solves the data analysis tasks of the zero-touch architecture. For most parts of the algorithm pipeline, we use techniques that were developed specifically for the zero-touch operations platform. For other parts, we apply existing techniques to our use case. Due to the fast advancements in the area of machine learning algorithms, new techniques will likely be proposed in the near future, that might provide properties better suited for the intended practical application. Therefore, an important design goal behind the algorithm pipeline is to separate the individual parts by clean interfaces and keep all parts interchangeable.

4.3.1 Overview

Figure 4.6 shows the individual data analysis steps of our approach. The design goal behind this pipeline is to keep each step interchangeable. The data exchanged between the steps is defined in an abstract way that allows using different specific algorithms.

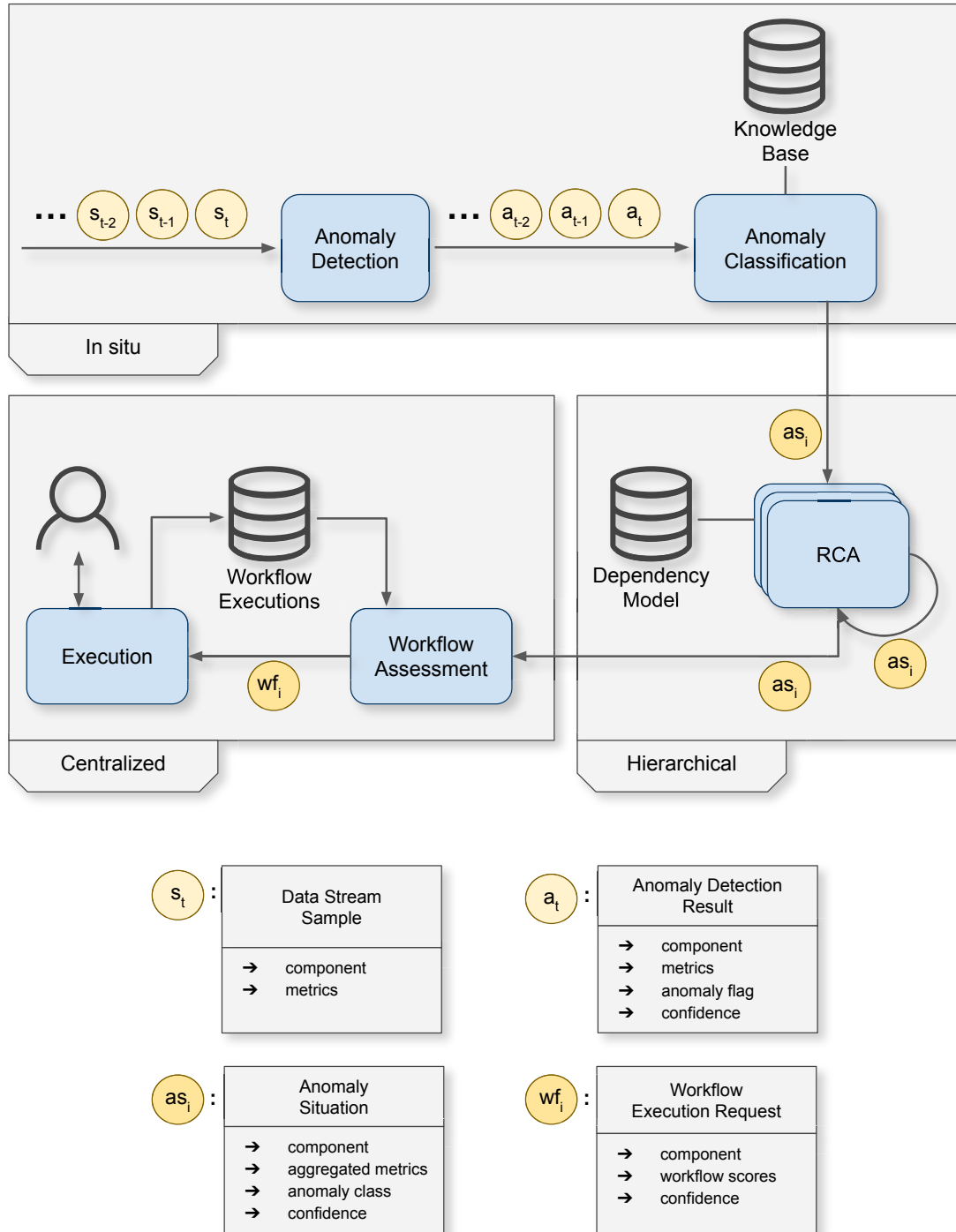
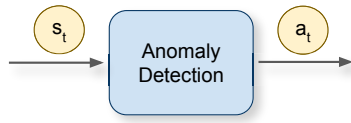
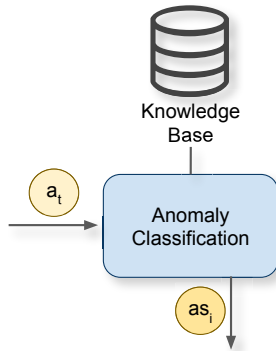


Figure 4.6: The data analysis pipeline of the zero-touch operations system



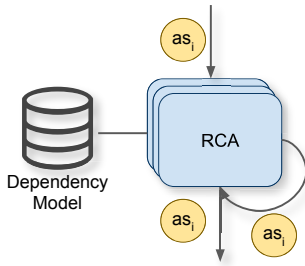
The anomaly detection step receives a stream of samples $s_t \in \mathbb{R}^n$, where n is the number of metrics that are delivered by the monitoring systems for a given component. For each sample, the anomaly detection computes a binary result in $\{0, 1\}$ and a *confidence value* in $[0, 1[$. The statistical confidence describes how likely it is that the computed result is correct. The output of the anomaly detection is a tuple consisting of the analyzed component, the input samples, the binary anomaly flag, and the confidence of the decision.



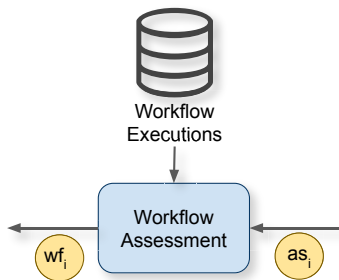
The anomaly classification receives the output of the anomaly detection but remains dormant as long as no anomaly is detected. Once an anomaly is reported, it analyzes the current situation and tries to classify it based on a database of historic observations. The output of this classification is an *anomaly class*. The anomaly class represents a class of potentially recurring anomalies that lead to similar behavior of the anomalous component. By itself, this class identifier has no meaning that humans can interpret, but administrators can choose to associate a label with the identifier that clarifies its origin or impact. For example, a common class of recurring anomalies are memory leaks, so an administrator could assign this label to an anomaly class that exhibits slowly rising memory usage. However, this label is for human-readability only and not evaluated algorithmically. Certain labels with clear behavioral patterns (such as memory leaks) could be assigned or suggested automatically, but we leave that out of the scope of our approach. If the anomaly classification encounters an entirely new anomaly, or if it cannot clearly classify the situation, it creates a new anomaly class identifier and reports a previously unknown situation. The anomaly classification also computes a statistical confidence of its decision, similarly to the anomaly detection step. The resulting confidence is the product of the anomaly detection and anomaly classification confidence values, and therefore represents the probability of a correct data analysis so far.

In addition to these results, the anomaly classification outputs an aggregated representation of the state of the anomalous component. This aggregation includes a distribution of the most recent data stream and other features that were extracted from the data during the anomaly classification. At this point the in situ data analysis is finished and the analysis results are potentially transported over the network. The data aggregation avoids transporting a history of the entire data

stream. At the same time, the data stream analysis is concluded as well, and the remaining steps operate on the basis of events. In Figure 4.6, this is indicated by as_i and wf_i , which have an i as index instead of a t : Events are identified by their index, while elements of a data stream are identified by time. We call the events emitted by the anomaly classification *Anomaly Situation* events, and they contain all information available at this point: the affected component, the anomaly class, the analysis confidence, and aggregated data describing the components behavior right before the anomaly.



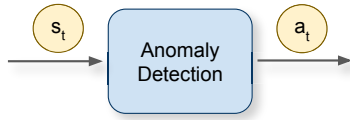
The next analysis step is the RCA, which receives *Anomaly Situation* events. This step is outside of the in situ execution, which means that it receives input events from a number of associated components. The view on multiple components allows to correlate situations in different components in order to decide where the problem originates. In systems that are too large for a single RCA instance, a hierarchical RCA provides better scalability. To achieve this, the system must be divided into a logical hierarchy of RCA domains. Data center sites, server racks, or availability regions are good candidates for such groups of tightly coupled components. Continuing up the system hierarchy, the VMs on a hypervisor can be grouped as well. The output of the RCA are again *Anomaly Situation* events, except that RCA forwards these events only for the component(s) that it considers the origin of the anomaly. This common interface also allows short-circuiting the RCA component in simpler setups that assume that anomalies do not propagate and are handled on every component individually. When the RCA computes a confidence value for its decision, it multiplies that value with the received confidence value to maintain an aggregated confidence for the entire data analysis pipeline.



The final part of the data analysis is the workflow assessment. Based on the anomaly class, this step looks up a list of remediation workflows that have been executed in the same situation before. Every workflow exhibits a collection of key criteria that describe aspects like its success, execution time, or impact on the service quality. Based on these values, the assessment step computes a score for every available workflow. The final output of the data analysis is a *Workflow Execution Request*, which contains the target component, the confidence of the data analysis pipeline,

and the computed scores for all workflows. Section 4.3.6 describes how the system selects a workflow based on that information.

4.3.2 Anomaly Detection



The anomaly detection step receives a live stream of metric data and enriches it by a binary anomaly detection flag and a statistical confidence. The main requirement for this step is to function in an unsu-

pervised fashion and to have a low resource consumption. Most fitting algorithms realize this by creating a *normal behavior model* based on the observed data and without any labeled training data. The underlying assumption is that most of the time, all components behave normally, and that statistically, anomalies occur rarely. In case of an anomaly, the algorithm detects a deviation from the normal behavior model. It is important to note, that load changes (both seasonal and exceptional) should not be interpreted as anomalies. We identify multiple strategies to avoid such *false positives*, which are described in the next section. After detecting an anomaly, the model continues to adapt to the incoming data stream, although it might stay in the anomaly state for a while. Eventually, the anomaly will be remediated. If the remediation works as expected, the time the component spends in anomaly state is short and has a low impact on the model, compared to the normal behavior data that is constantly collected. The same applies when the anomaly is transient and resolves itself after a short period of time.

Handling False Positives

When a component permanently changes its behavior, this change does not represent an anomaly, but a change of its normal behavior. Examples for such an event include upgrades to a new software version, or permanent reconfiguration of the application. From the point of view of the anomaly detection, such events highly resemble anomalies, although they are expected, and the self-healing system should not try to remediate the situation. Such *false positives* can also occur for shorter time intervals, for example due to seasonal load variations or rare events during the systems normal operation.

When there is no way to algorithmically tell the difference between an anomaly and normal state change, the anomaly detection relies on external knowledge to correct its false positive decisions. Many anomaly detection algorithms support such external feedback, which allows them to learn from wrong decisions and improve their false positive rate. The following sources of external knowledge are

available to the anomaly detection:

Human administrator Asking for human input is the safest way to tell whether the system is running normally or not. In conservative or critical setups, where wrong decisions could lead to catastrophic results, this would be an acceptable strategy, but it mostly strips the self-healing system of its automatic features. Section 4.3.6 describes our approach for deciding whether to add a human into the loop, or to automatically execute a remediation workflow.

Input from the Development Process Modern development processes use CD servers to automate the task of testing, building and deploying applications. By listening for events, such as upgrades of individual services or other maintenance activities, the anomaly detection determines when the behavior of a component is expected to deviate. After an upgrade event, the algorithm will observe the changed components for a period of time to build a stable model of its new normal behavior.

It is possible that the application consumes more resources after an upgrade, due to an introduced bug. We consider this a *development fault* (see Section 4.1.2), and therefore not the responsibility of the anomaly detection. If the buggy behavior of the application remains constant over a longer period of time, we consider it the new “normal” behavior, until the developers of the software resolve the problem.

System Load Metrics Variations in client-induced load are the main drivers for changing resource consumption of a service. For different services, the usage of different resources correlates in certain ways with the number of client requests. For example, a computation-intensive HTTP-based service might linearly increase CPU utilization with the number of requests, while a database server would generate disk access as well. The load level is represented as a metric such as `client-requests-per-minute`; When the algorithm detects an anomaly while the value of `client-requests-per-minute` is higher than previously observed, it can deduce that it is simply observing a higher load than before, and avoid flagging an anomaly. This way, the algorithm builds different normal behavior models for different values of `client-requests-per-minute`.

It is important to note that this strategy assumes rather specialized services that generate a consistent resource consumption pattern for different load levels. Traditional monolithic applications perform a variety of tasks with

different resource usage impacts, that are hard to represent as a single metric. However, an application constantly working at a certain capacity will still produce a consistent resource usage pattern over a sufficient time span. When a single metric does not adequately represent the system load, multiple load metrics are used for more precision. For example, the load of a database is better represented by separate metrics for read and for write requests.

When no application-specific load metric is available, we use the incoming network traffic as a fallback. Since requests to services are sent over the network, the number of received packets or bytes is an approximation for the number of client requests.

Subsequent Analysis Results When the anomaly detection emits an alert, follow-up analyses are launched to provide further insight into the situation. These analyses can be more resource-intensive, since they are executed less frequently than the anomaly detection. As such, they often produce a more accurate classification of the current situation, including the possibility that the situation is, in fact, not an anomaly. In case of such a result, the appropriate feedback is propagated back to the anomaly detection in order to refine the model that produced the false positive result.

IFTM-based Anomaly Detection

We have contributed to the development of Identity Function Threshold Model (IFTM) – an anomaly detection algorithm designed specifically to fulfill the requirements of the zero-touch operations platform [151]. Figure 4.7 visualizes the IFTMs approach, a combination of IF (Identity Function) and TM (Threshold Model).

In the context of IFTM, an identity function $\xi : \mathbb{R}^n \rightarrow \mathbb{R}^n$ attempts to *reconstruct* each input sample $s_t \in \mathbb{R}^n$ based on a model that represents the history of the data stream. *Reconstruct* means that the data point is first compressed into a latent data space, and then decompressed back into the original representation. The operations for compression and decompression share an underlying model that represents the history of the observed data stream. By carefully selecting the compression and decompression algorithms, and the training procedure of the underlying model, the reconstruction can adapt to the data stream over time. As long as the incoming data points resemble one another, the reconstruction works well. Data points that deviate from the history of the stream result in a *reconstruction error*: a distance between the input data point and the result of the

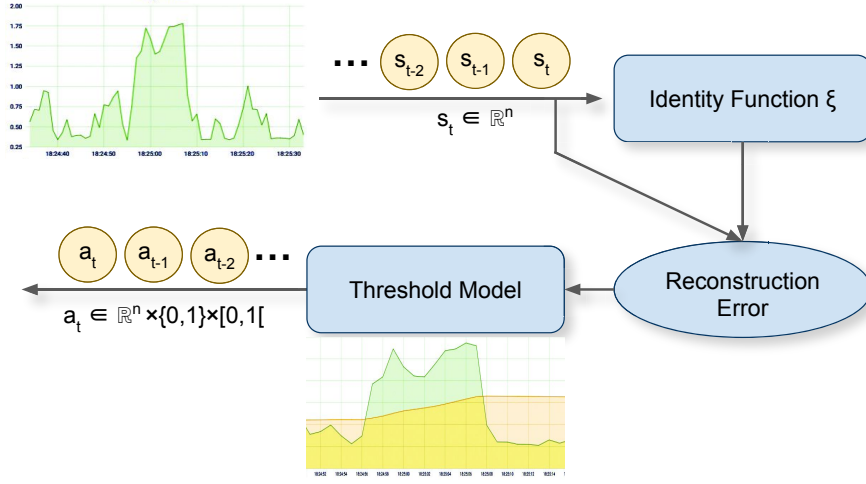


Figure 4.7: The basic principle of IFTM-based anomaly detection

reconstruction. A high reconstruction error indicates a deviation from the normal data points in the stream, and is therefore interpreted as an anomaly. In other words, given a data point $s_t \in \mathbb{R}^n$, the identity function computes a reconstruction value $s'_t = \xi(s_t)$, where $s'_t \in \mathbb{R}^n$. Based on s'_t , a reconstruction error Δ can be computed using the Euclidean distance:

$$\Delta = \|s_t - s'_t\|^2 \quad (4.1)$$

The identity function ξ adapts its internal model in order to minimize the reconstruction error iteratively over time:

$$\operatorname{argmin} \|s_t - \xi(s_t)\|^2 \quad (4.2)$$

The second part of IFTM, the Threshold Model, decides what reconstruction error must be exceeded in order to report an anomaly. The threshold adapts to the reconstruction error over time, depending on how well the identity function represents the data stream. The more the reconstruction error exceeds the threshold, the higher the confidence of the anomaly detection.

Figure 4.8 shows an example of the principle behind IFTM-based anomaly detection. Four normalized input metrics form the input for the identity function and the threshold model. The computed reconstruction error and the threshold can be plotted to get a good visual understanding of the development of the data stream. In the example, the reconstruction error exceeds the threshold for a period of time, which is reported as an anomaly alert.

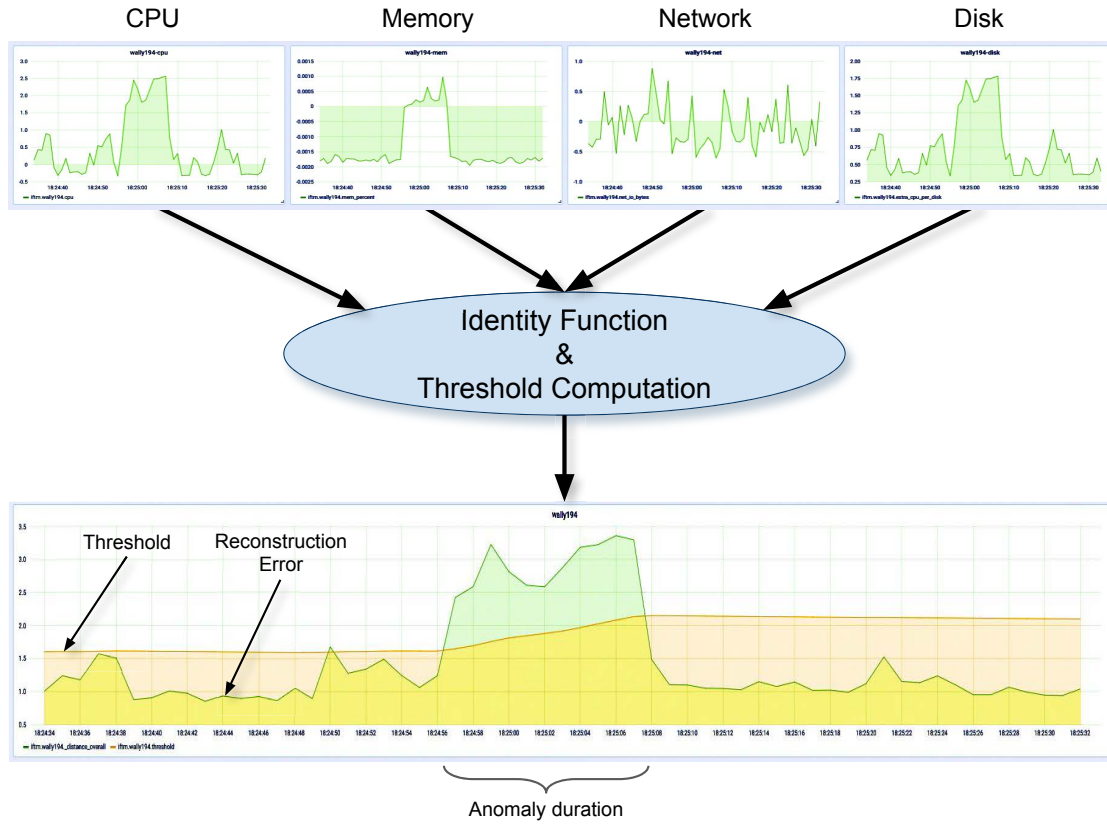


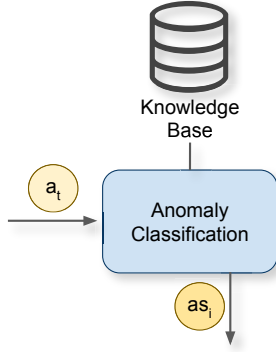
Figure 4.8: Example visualization of an IFTM-based anomaly detection

The IFTM approach allows to select individual algorithms to implement the identity function and the threshold model. Schmidt et al. evaluate multiple possible identity functions [151]. Autoencoders are neural networks that first compress the input data into a smaller number of neurons, before decompressing it into the original representation. This combination of a compression and a decompression function makes them a natural candidate for the IFTM identity function. By using Long-short term memory (LSTM) neurons [79], the network additionally represents a notion of time that is capable of capturing trends and seasonalities in the data. An alternative approach uses online clustering algorithms such as BIRCH [185], which creates a memory-efficient summary of a continuous high-dimensional data stream. Spherical clusters form around the observed data points and separate normal from abnormal regions in the value range [68]. The identity function then consists of mapping an incoming data point to the closest existing

cluster. For further algorithmic details and evaluations we refer to [71, 151, 152, 153].

Since the IFTM approach to anomaly detection works entirely unsupervised, it fulfills the main requirement to be used in the zero-touch architecture. Regarding limited resource consumption, this approach generally offers high efficiency, but also depends on the specific underlying identity function. The IFTM approach is robust against changes of the processing frequency. This means that in high-load situations, the algorithm can drop intermediate processing requests to reduce the overall resource consumption in exchange for a better data analysis latency.

4.3.3 Anomaly Classification



The task of the anomaly classification is to observe an anomalous component and determine, whether the current anomaly has been previously observed. The goal is to use this knowledge to select an appropriate remediation workflow based on “experience”. This mirrors the behavior of administrators in a similar situation: The first step is to classify the situation (e.g., “The service is experiencing a memory leak”), the second step is to choose an appropriate countermeasure based on experience (“In the past, rebooting the VM has helped to resolve this”). Since anomaly

classification does not continuously analyze data streams, it does not have the same strict requirement of low computational overhead. A bounded processing time is still beneficial for a timely remediation of anomalies.

We have contributed to the approach of *density grid mapping*, which we leverage for the purpose of anomaly classification [4]. Density grid mapping consists of two steps, as visualized by Figure 4.9.

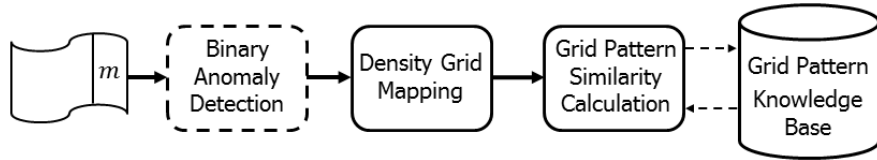


Figure 4.9: Steps of the *density grid mapping* approach for anomaly classification [4].

First, we load a part of the time series data that describes the behavior of the anomalous component directly before the anomaly. This data consists of samples $s_t \in \mathbb{R}^n$, where n denotes the number of collected metrics. The high-dimensional value space \mathbb{R}^n of that data is then partitioned into a defined number p^n of *grid cells*. Every resulting grid cell is assigned a *weight*, which equals the number of input data points that lie within the bounds of that cell. The result is a density grid that represents the behavior of the component at the respective point in time. A series of density grids is called a *grid pattern* and summarizes the components behavior right before the anomaly was observed. Figure 4.10 shows four examples for two-dimensional density grids with color-coded weights. The shown grids were recorded while injecting different anomaly situations at the same observed component.

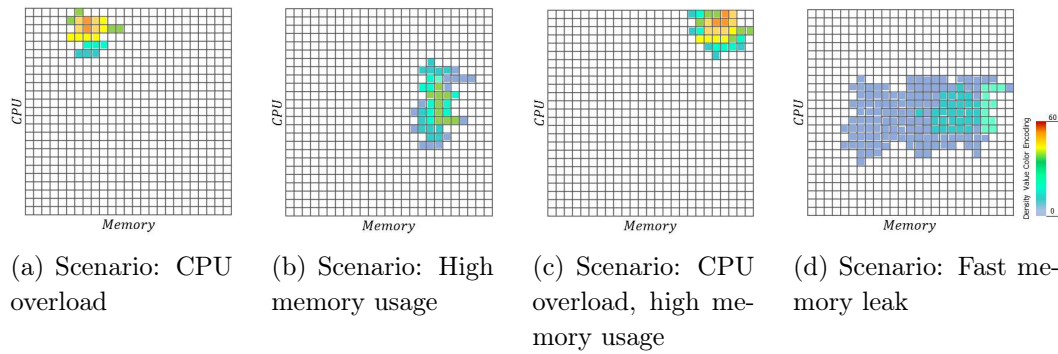
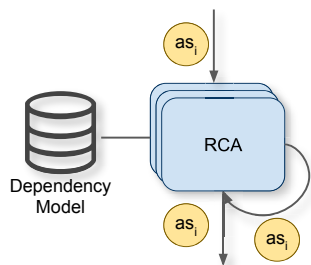


Figure 4.10: Examples for two-dimensional density grids based on the usage of the CPU and main memory. The underlying resource usage data was recorded during different anomaly scenarios [4].

The second step of the density grid pattern approach calculates a fuzzy similarity value between the grid pattern of the current anomaly situation and all historically recorded grid patterns for the anomalous component. The similarity calculation is fuzzy since it is unlikely that two anomaly situations lead to the exact same behavior. The fuzziness allows for small differences and behavioral deviations, while still yielding larger similarity values for grid patterns that resemble each other in their form and magnitude. The final output is a list of previously recorded anomaly situations that resemble the current ongoing anomaly. The combination of these recorded situations represents the *anomaly class*, while the confidence of the algorithm is derived from the fuzzy similarity values. For further details and evaluations of the density grid pattern approach, we refer to [4].

4.3.4 Root Cause Analysis



The purpose of the RCA component is to pinpoint the origin of an anomaly, in case multiple components exhibit anomalous behavior within a short time interval. This task involves combining anomaly situation events to correlated groups. In other words, both the input and the output of the RCA are streams of anomaly situation events, but the output events are possibly filtered or grouped. The RCA can hold off anomaly situation events for a period of time in order to verify, whether another correlated event will arrive in the near future. Since both the anomaly detection and the anomaly classification deliver results annotated with a level of confidence, the RCA component can use that information to weigh the received information, or to partially ignore it.

For reasons of scalability, the RCA works hierarchically: Each instance of the RCA has a number of components in its scope, which it is responsible for analyzing. In case a clear identification of the root cause is not possible, the request for further analysis of the situation is forwarded to a higher level RCA instance that is responsible for a larger number of components. The hierarchy of RCA instances is constructed depending on the structure of the underlying cloud platform.

The most trivial implementation of RCA is one that simply forwards all incoming anomaly events unchanged. This allows to use the remaining parts of the zero-touch operations system under the assumption that anomalies do not propagate between system components. As a result, it is possible, that more remediation workflows will be executed than necessary to resolve each problem. We consider such a “no-op” RCA as a valid choice for non-critical systems: The self-healing functionality eventually brings the system back to a normal state, but the additional remediation workflows have a higher impact on the application software, and possibly the service quality.

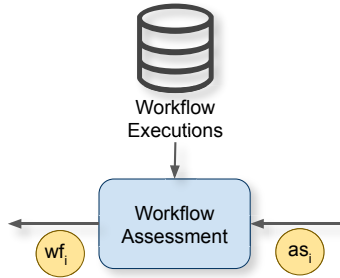
Nguyen et al. have proposed an approach for online RCA that relies on the timestamps of anomaly events and the dependencies between anomalous components [133]. We leverage this approach as a simple RCA with low resource overhead. Upon receiving an anomaly event, we wait for further events for a predefined amount of time. The received events are then ordered by time and split into groups, based on the proximity of the anomalous components on the dependency topology graph. The resulting groups are not directly connected in the dependency

graph, which makes it unlikely that the anomaly has propagated between them. For every such group, the earliest anomaly event is reported as the root cause.

Although this approach considers both the anomaly timing and the system topology, it fails to correctly identify the root cause when the symptom of a fault is detected before its origin. More recent advancements in the field of RCA do not rely solely on the order of anomaly events [40, 178]. Instead, these approaches try to determine how the behaviors of the different components influence each other. A high correlation (computed by correlation measures such as the Pearson correlation coefficient [19]) between the resource usage metrics of two components indicates an interdependence between the two components. Since this interdependence is based on actually observed data during the anomaly situation, it is more reliable than the static information in the dependency graph. Anomalous components, that have a high interdependence with many other components, have a higher chance of being the root cause.

Unfortunately, such approaches are computationally expensive and require the centralized collection of data from all involved components. As of the time of writing, to the best of our knowledge, there has been no contribution that proposed a truly scalable and practically applicable approach to solve this problem. We expect further developments in this field.

4.3.5 Remediation Workflow Assessment



After the anomaly detection, anomaly classification, and RCA, the final step is to assess each remediation workflow in the current situation. We formulate this task the following way: Given a set of alternative remediation workflows, each annotated with a set of criteria, assign a *workflow score* $WS \in [0, 1[$ to each workflow.

The WS value represents the expected effectiveness of the workflow in the given anomaly situation. A value of $WS = 1$ is not allowed because a probabilistic assessment can never be entirely certain. The workflow with the highest WS value is regarded as the *optimal* decision. An optimal decision cannot be made based on a single criterion alone. For example, while the success rate of a remediation workflow is important, other factors such as the execution time or the impact on the service quality should be considered as well. We define the following criteria that characterize each workflow in the context of an anomaly class:

Remediation success rate. The share of total executions of this workflow that lead to a resolved anomaly.

Execution success rate. The share of error-free executions of this workflow, regardless of whether the anomaly was resolved.

Execution time. The average time from starting the remediation workflow until its completion. It is possible that the anomaly is not yet remediated during this time – i.e., the remediation workflow might take some more time to show its effect.

Mean time to repair (MTTR). The average time from starting the remediation workflow until it resolves the anomaly. This value is undefined if the anomaly is not actually resolved by the workflow.

Complexity. A static value that represents the complexity of the workflow. This can be expressed in the number of individual steps that are needed for the execution. A higher workflow complexity is generally associated with a higher failure probability and longer execution time. While failure probability and execution time are also measured individually, the complexity is a predefined criterion that is available even prior to the first execution of each workflow. It is therefore an important criterion early during the system bootstrapping.

Service quality impact. An average value that expresses how strongly the workflow disturbs the affected component. Even in the presence of anomalies, the component possibly still delivers its service, and a remediation workflow such as a restart might disrupt that. This criterion implies that the service quality can be measured.

Number of executions. The number of times a remediation workflow has been executed in the past. A higher number means a richer experience, and should therefore be preferred when other criteria values are high.

Please note that some of these criteria favor small values (e.g., mean time to repair), while other criteria favor large values (e.g., remediation success rate). This difference in significance must be considered when implementing the assessment.

We leverage existing methods from the field of Multiple Criteria Decision Making (MCDM) [173] to implement the computation of the WS value for each remediation workflow. The following two sections introduce the weighted sum model and the TOPSIS method for solving MCDM problems. We use one of these two

methods, depending on the available amount of historical data. These two methods are rather straightforward to understand and implement, and require a clear set of input parameters. More importantly, their simplicity makes them computationally inexpensive, which is important for applying these methods in the self-healing architecture. Since MCDM problems are an active field of research, other approaches from the literature might provide similar results.

Weighted Sum Model

The weighted sum model [65, 77] is a straightforward way of solving an MCDM problem. Given a number of alternatives and criteria, first all criterion values are normalized based on their distribution. Criteria that favor a small value for the optimal solution, are inverted. Next, each criterion value is multiplied by a predefined *weight*. These weights must be statically defined before executing the algorithm, and denote the importance of the respective criterion for the overall result. In the simplest case, the administrator can select a value of 1 for all weights, resulting in an equal influence of all criteria of the final result. Since the primary target of the decision is to select a successful remediation, the administrator might select a higher weight for the two criteria *remediation success rate* and *execution success rate*. However, there is no computationally inexpensive way to automatically determine appropriate weights.

For each remediation workflow, we compute the WS value as follows. The first step is to normalize the criteria values in the observed value range for each observed criterion, resulting in a normalized criteria vector $c \in [0, 1]^n$. The criteria values that favor small values (such as execution time) are inverted by subtracting them from 1. Given a vector of corresponding weights $w \in [0, 1]^n$, WS is the weighted average of the criteria vector:

$$WS = \frac{\sum_{i=0}^n c_i * w_i}{\sum_{i=0}^n w_i} \quad (4.3)$$

When all weights are set to zero, WS is instead defined as 0. To allow a meaningful normalization of criteria vectors, only workflows with a configurable minimum number of executions are considered. For other workflows, WS is defined as 0.

Table 4.1 shows example remediation workflows and values for the respective decision criteria. The criteria values were obtained experimentally in a video streaming service with an artificially injected anomaly that increased the network latency on a specific hypervisor. In this example, the only remediation workflows that suc-

cessfully fixed the anomaly were migrating the VM to another hypervisor, and scaling up the service. After collecting this data, the resulting ranks in this example situation show that the *migrate* workflow would be executed.

Table 4.1: Example remediation workflows and decision criteria, collected through experimentation. Weights were set to 1, except for the success rate (weight 5).

Remediation Action	Execution time (s)	Complexity	Impact	Execution success	Remediation success	Rank
reboot	47.18	3	79228.1	99.962	0	4
migrate	23.00	6	66416.8	99.991	1	1
resize	84.58	4	9242.9	98.123	0	5
scale up	154.25	5	113904.0	99.996	1	3
restart service	2.66	1	0.0	98.584	0	2

TOPSIS

The Technique for Preference by Similarity to the Ideal Solution (TOPSIS) [93, 134] is another method for solving MCDM problems. The input data is the same as for the weighted sum approach, but it is not necessary to manually define weights for the criteria. After normalizing the criteria, TOPSIS computes a theoretical *ideal* alternative based on the value ranges of all input values. In the case of selecting a remediation workflow, the ideal workflow would be the one with the smallest execution time, highest remediation success rate, smallest impact on the service quality, and so on. The real alternatives are then ranked based on their Euclidean distances to the ideal solution, and the *WS* value is obtained from normalizing these distances.

The advantage of this approach is that it requires no manual configuration. However, it does not work well when the number of historic observations is low. When each workflow has only been executed a few times, or even never before, the ideal workflow cannot yet be reliably derived. Therefore, we use the weighted sum model with manually adjusted weights, until the zero-touch system has executed each workflow a minimum number of times. The required number of executions is an input parameter defined by the administrator. This way, administrators can give preference to their manually defined weights, if so desired.

4.3.6 Controlled Level of Automation

An important property of the zero-touch operations architecture is to allow comprehensive control over the level of automation. The main aspect to configure is: Given an anomaly situation and all data analysis results, should a workflow be executed automatically, or given to the administrator as a suggestion?

For this decision, we use the two key figures delivered by the data analysis pipelines: the analysis confidence and the highest score assigned to a workflow. For both values, the administrator can set a *minimum value* that must be exceeded in order to automatically execute the workflow. Should any of the two values fall beneath their respective threshold, the workflow will not be executed and instead only displayed as a suggestion for the administrator, accompanied by all retrieved information.

We call the minimum analysis confidence configured by the administrator $MC \in [0, 1]$, and the minimum workflow score $MS \in [0, 1]$. These thresholds can be configured individually for every remediation workflow, or uniformly for the entire system. The range of values for MC and MS allows for three distinct cases, which have different implications. These cases are analogous to levels of automation described in Section 2.3.

$MS = MC = 1$ (**Suggesting**): Workflows with both thresholds set to 1 are never executed automatically, since both WS and the analysis confidence are defined within the range $[0, 1[$ and therefore can never reach 1. Since the decision making algorithms are based on heuristics and probabilities, full certainty can never be achieved. When the system concludes that such a workflow is the best option for a given anomaly situation, it notifies the administrator and provides all available information as a suggestion for how to proceed. The administrator can then choose to execute the action, or to perform a manual remediation. Should the administrator manually trigger a remediation action through the zero-touch platform, the decision will be used for building a knowledge base and improving future suggestions.

$0 < MS < 1, 0 < MC < 1$ (**Assisted cold-start**): Intermediate threshold values allow to automatically execute workflows with a high confidence and score, while only displaying suggestions in other cases. Setting these values closer to 0 results in more automatically executed workflows and more autonomic decisions of the system. High values are more conservative. This spectrum allows administrators to start with a conservative system and increase the

level of automation once they gain trust in the system's autonomic decisions. This way, we solve the problem of an empty knowledge base, when bootstrapping the system, through human input: The administrator controls the remediation process until it reaches a satisfactory level of decision accuracy.

$MS = MC = 0$ (**Fully autonomic**): This setting gives the system full freedom over executing the remediation workflow in anomaly situations. With a sufficiently filled knowledge database, this option will continue to select the most appropriate remediation workflows without prior consultation with administrators. However, a freshly bootstrapped system will not have any experience to build upon, resulting in low workflow scores and confidence values. In such situations, the only fallback strategy is to randomly execute remediation workflows until an anomaly is resolved. Should such a workflow be triggered, the result will be recorded and the experience will improve the selection process during the next anomaly. Such a *cold start* mode of operation is applicable in few cases in practice, for example when the remediation action has a low impact on the service quality, and it is generally acceptable to execute it by mistake.

Chapter 5

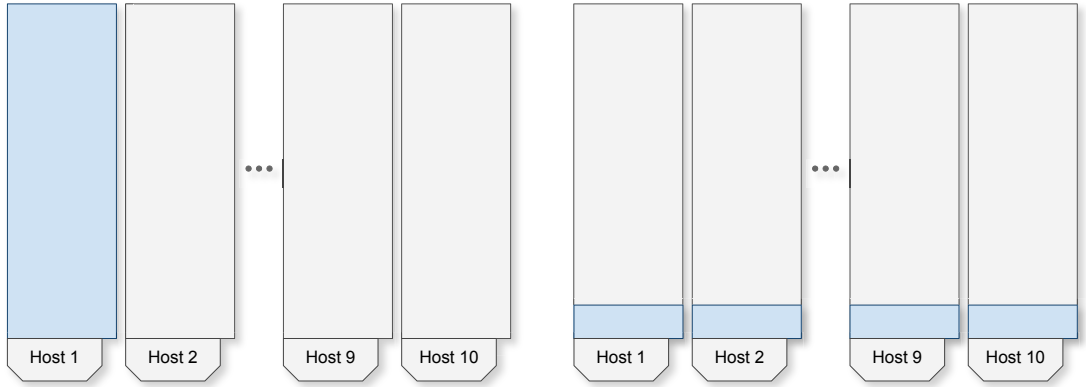
In Situ Data Analysis

Contents

5.1	Interfaces of Data Analysis Processes	71
5.2	Modeling the Data Analytics Pipeline	74
5.3	Enforcing Resource Limitations	78
5.4	Scheduling of Data Processing Steps	81
5.4.1	Static Scheduling	81
5.4.2	Adaptive Scheduling	82

An autonomic, self-healing cloud infrastructure depends on the analysis of monitoring data. Since the data needs to be analyzed online and with low latency, it is not feasible to record it and transfer it for external offline processing. Therefore, we reserve a part of the computational resources of the cloud infrastructure to the task of data analysis. There are two ways to allocate these resources, as visualized by Figure 5.1: on dedicated data analysis hosts, or partially on each host. *Dedicated* data analysis means that a number of the physical hosts works exclusively on the task of data processing. For example, if 10% of all resources are available for analysis, one out of ten servers is removed from the cloud platform and becomes part of a dedicated data processing infrastructure. In the opposite approach, 10% of the resources of *each* hypervisor are reserved for data analysis. We call this approach *in situ* data analysis (Latin for *in place*). Both dedicated and in situ analysis have the same absolute overhead of compute resources, but different implications on the system structure.

The dedicated data analysis approach has a number of drawbacks. It enforces sending the input data over the network, whenever an analysis is needed. This



(a) Dedicated data analysis: entire host(s) reserved for data processing

(b) Distributed resources, partially allocated for in situ data analysis

Figure 5.1: Computational resources, allocated for *dedicated* and *in situ* data analysis. The blue area is reserved for data analysis, the remaining area for general cloud workload. Assumption: 10% of overall resources are used for data analysis.

strains the network, limits the data collection frequency, and restricts the possible volume of the analyzed data. Transporting the monitoring data over the network increases the duration and fluctuation of each data processing task, depending on the speed and load of the network. This aspect is exacerbated in edge computing platforms, where different sites of the cloud are potentially far apart in terms of the network topology. This way the network has a particularly large impact on the transfer of analyzed data. Furthermore, a dedicated infrastructure for data analysis implies an organizational overhead, due to the required administration, and hinders an expansion of the cloud infrastructure. In contrast, by making use of the resources of each hypervisor, the data analysis naturally scales with the extent of the cloud.

Therefore, our approach to data analysis relies on each hypervisor's own compute resources. The main means for making the data analysis itself scalable is a hierarchical execution model that is based on forwarding intermediate results to higher-level analysis steps. This multi-level aggregation model spreads the overall computational load as much as possible over all available resources. The aggregation levels can mirror existing aggregation tiers in the infrastructure, as visualized in Figure 5.2. In a self-monitoring cloud infrastructure the lowest level is a single physical or virtual host that monitors itself locally and continuously executes a low-overhead anomaly detection algorithm. Aggregated results of this local anomaly

detection are forwarded to a second-level analysis that runs on each hypervisor and aggregates the results from all VMs running on it. The results can further be aggregated on the level of physical server racks, data center availability zones, entire cloud or edge sites, and finally the global scope. A well-chosen set of aggregation levels can greatly reduce the bandwidth used by monitoring data, for example by analyzing intermediate results of all VMs on their corresponding hypervisor, since this data does not have to be transported over the physical network.

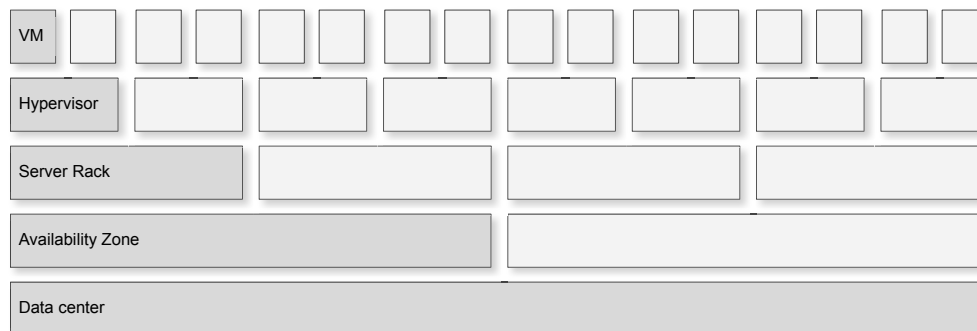


Figure 5.2: Possible aggregation layers in a cloud data center

Fast advancements in the field of machine learning and anomaly detection continuously produce new algorithms suitable for analyzing the operation of cloud resources. Therefore, a secondary design goal for the in situ analytics engine is extensibility. The engine should allow for easy integration of new data sources and types of data, as well as modular adjustments of the multi-level data analysis process.

The remainder of this chapter describes the design of a data analytics engine that meets the aforementioned design goals. Section 5.1 introduces the high-level components of the engine and the interfaces between running data analysis processes and their environment. Section 5.2 shows our modeling approach to describe and automatically orchestrate a data analysis pipeline on a given cloud infrastructure. In Section 5.3 we describe how the engine limits the resources occupied by data analysis processes, while Section 5.4 defines our scheduling approach.

5.1 Interfaces of Data Analysis Processes

The main task of a data analysis engine is to execute the requested data analysis processes. Therefore, we first describe the interfaces of such a process, and how

the process interacts with its environment. Figure 5.3 shows this high-level view.

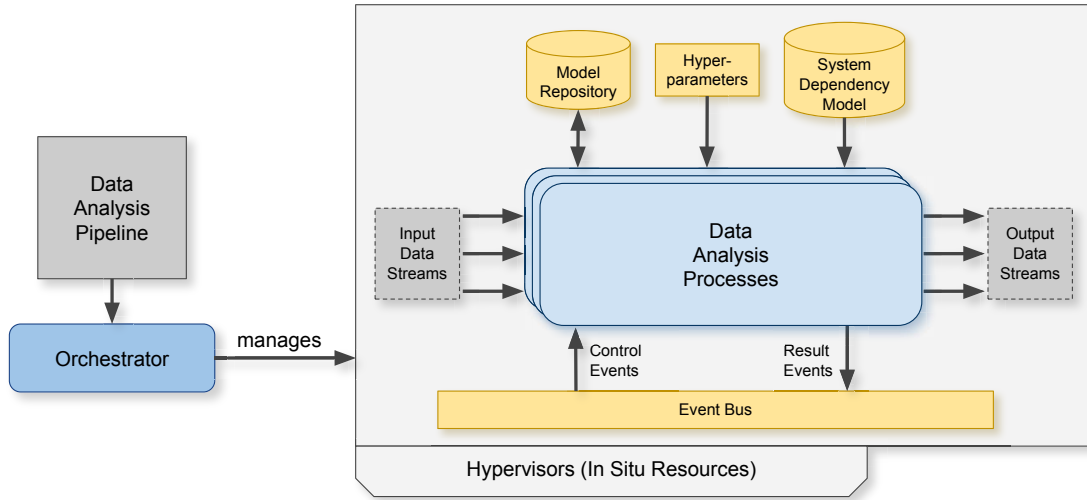
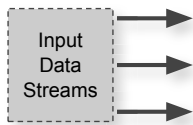


Figure 5.3: Interfaces between data analysis processes and their environment

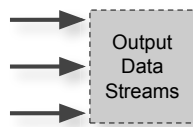
The *orchestrator* component is the core component of the data analysis engine. It is logically centralized, although a replicated set of orchestrators with a voting mechanism is preferred in practice for reliability reasons. The main input for the orchestrator is a model of the data analysis pipeline. The orchestrator uses this model to derive a list of required data analysis processes and schedules them on the available execution hosts. There can be an arbitrary number of running *Data Analysis Processes*, which interface with common external storage and communication systems.

The *hyperparameters* input data is defined individually for every process and contains values for the execution parameters of the used data analysis algorithm. The designer of the data analysis pipeline defines the values for the hyperparameters before the analysis process starts, and these values are not expected to change at runtime. The explicit mention of the hyperparameters is due to their importance for the performance of the used algorithms. In many cases, an expert must tune the algorithms and their hyperparameters to work well with a given domain. Some algorithms allow to automatically tune their hyperparameters at run-time, in which case the list of input parameters is reduced. *AutoML* is an emerging class of algorithms with such capabilities

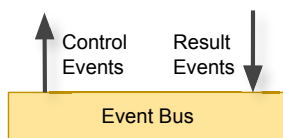
[53, 92, 135]. AutoML algorithms combine automatic hyperparameter tuning and feature selection by applying optimization procedures on batches of labeled input data. By repeating the optimization procedure in regular intervals, AutoML algorithms are also able to analyze data streams. Optimization procedures for the selection of hyperparameters and features include Bayesian optimization [53, 92] and genetic programming [135].



Aside from configuration parameters, a data analysis receives *input data* for processing. Since the objective is to detect and remediate anomalies with a low latency, the input data mostly consists of high-frequency data streams. In certain cases it is useful to receive a certain amount of historic data before reading the actual live data stream. Many algorithms can use historic data to better prepare for the actual operation, reduce warm-up phases, or improve accuracy through better data normalization [37, 141]. A single data analysis step can also receive multiple input streams. For example, it is often necessary to combine the data of multiple monitoring systems to obtain a richer representation of a single component. Another important use case for multiple input streams is higher-level data analysis, which further processes the intermediate results of lower-level analysis. Examples for such high-level data analysis include finding the root cause, when multiple anomalies are detected at the same time, or analyzing all VMs on a single hypervisor, which might discover anomalies that would otherwise remain hidden. Multi-level data analysis will be further covered later on in this chapter.

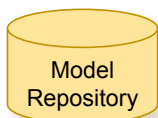


After processing its input data, a data analysis step typically produces an *output data stream*. A step without an output data stream often has a miscellaneous task, such as storing the data into a database, or sending out alert emails for certain events. Another task that does not produce any output data, is selecting a remediation action that will be executed as a reaction to a detected anomaly. The output of such a “decision making” step consists of notifying an external component of the selected remediation workflow. In general, however, most data analysis steps do produce one or more data output streams, which can be consumed by one or more other data analysis steps, or by data sinks outside of the data analysis engine.

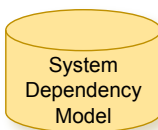


Besides data streams, a second important communication mechanism for data analysis steps are *events*. In contrast to data streams, which are continuous and carry homogeneous data, event streams have no expected timing characteristics and can carry arbitrary sequences of different classes of data. Events

can be used both to communicate data analysis results, and to control the execution of data analysis steps. An anomaly classification step issues an event when it successfully classifies an anomaly based on the received data stream. Likewise, the administrator of a data analysis pipeline can order the anomaly detection step to reset its model by sending it an according control event. To support this flexible usage of events, our data analysis engine uses a publish-subscribe event bus model based on *labels* and *selectors*. When publishing an event, the creator of the event attaches a list of labels (key-value pairs) to it. A subscriber listens for certain events by defining a selector, which is a Boolean expression that is evaluated against a set of labels. The selector determines, which events the subscriber is interested in, and hence the subscriber will receive all events that match the given selector. The labels and the selector can be chosen arbitrarily, which makes this mechanism flexible enough for different use cases, including the ones mentioned above.



The *model repository* module in Figure 5.3 allows the data analysis step to store or load machine learning models. Machine learning algorithms work on an underlying *model*, which represents the algorithm's internal state. The exact data in a machine learning model depends on the algorithm. To keep the algorithm implementation flexible, the step can freely store and load models at any time. This allows to load and store multiple models per analysis step, and to use stored models for various purposes. The model repository is a globally accessible storage system, which makes it a bottleneck in terms of scalability. Distributed read replicas can somewhat mitigate this problem, but consistent write access to the repository usually requires expensive locking. The implementer of a data analysis step should therefore restrict the number of writes to the model repository, compared to the number of local, non-persisted model updates.



The *system dependency model* contains the topology of the underlying cloud system. This is an important part of the zero-touch operations architecture (see Section 4.2.1). Data analysis algorithms can use the dependency model to obtain meta-information about the data they analyze.

5.2 Modeling the Data Analytics Pipeline

Deep insights from the analysis of monitoring data emerge after a sequence of interdependent analysis steps. To name an example, the first step when analyzing

a data stream is usually to preprocess, normalize, and filter the data. In our zero-touch operations system, this is followed by the local anomaly detection that assesses the state of every component individually. Subsequently, a number of higher-level analysis steps examine the dependencies between different groups of interconnected components to get a richer context for the anomaly detection and root cause analysis. In a cloud system, the groupings of interconnected components are given by colocation of physical resources (like server racks), network paths, or communication paths on the service layer. This sequence of analyses produces a system view that is finally consumed by a decision making step, which tries to recognize the current anomaly situation, and selects a remediation action that is most likely to resolve the situation. Figure 5.4 visualizes a physical deployment of the named example analysis steps in a small cloud system.

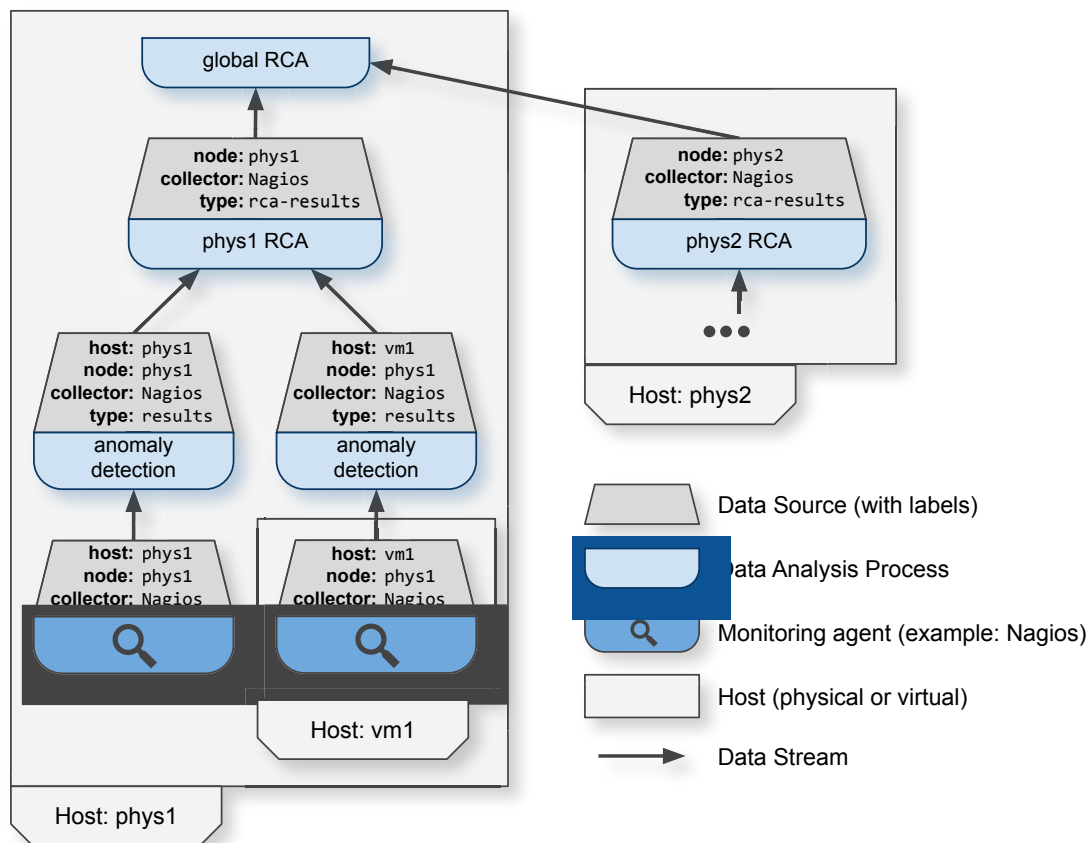


Figure 5.4: Example deployment of a data analysis pipeline for a small cloud system

The trade-off between a lower-level and a higher-level data analysis is between the level of detail in the data, and the number of correlated components. Local anomaly detection analyzes detailed data about each component, while the final decision making is based on aggregated (and less detailed) data. On the other hand, the global decision making step has aggregated data about a large number of correlated components, which results in a richer analysis context. Analyzing detailed data from a big number of components would combine both benefits, but does not scale indefinitely for large systems.

Defining data analysis pipelines requires a flexible, yet powerful execution model. The main task of the *analysis pipeline model* is to express arbitrarily nested data analysis pipelines, their interconnections, and input data streams. Our analysis pipeline model consists of a decoupled definition of *data sources* and *data analysis steps*. Figure 5.5 gives an abstract description of the proposed data model.

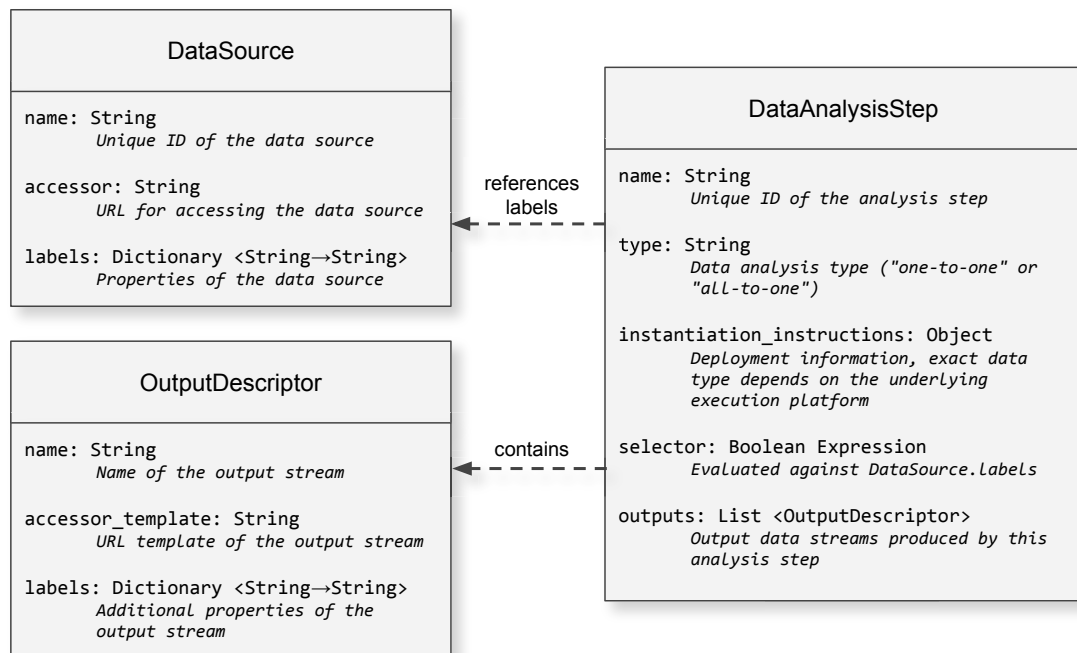


Figure 5.5: Analysis pipeline data model with decoupled data sources and analysis steps

The main property of a data source definition is a data stream accessor: the URL where the data stream can be obtained. Aside from the accessor, a data

source definition contains metadata in form of a key-value list of `labels`. These `labels` contain machine-readable properties of the data stream that are used to identify the stream and find all data analysis steps that are able to analyze it. Useful data source labels include the monitoring system that collected the data, the system layer of the monitored component (e.g., physical, virtual, application), the component where the data originated, possible parent components, and the data sampling frequency.

On the other hand, an analysis step definition contains all instructions necessary to deploy an instance of the underlying algorithm. In practice this includes the name of a container image or virtual machine image with all accompanying configurations. Besides the execution instructions, the analysis step definition also contains initial values for all relevant hyperparameters of the algorithm, which have to be defined by a domain expert. Finally, the *data source selector* is a function that receives a dictionary of strings and returns a Boolean result. The selector function is evaluated against the labels of each data source and the result determines whether a data source is compatible with the respective analysis step. Compatible pairs of data sources and analysis steps result in the deployment of an analysis process. For example, assume the following data source selector in pseudo-code:

```
1 def select(labels: Dictionary<String, String>):  
2     return labels["layer"] == "physical" &&  
3         labels["nagios_version"] != ""
```

Listing 1: Example data source selector function

This selector matches all data sources with the label `layer = physical` that also have the label `nagios_version` with a non-empty value. In other words, this data analysis step analyses data about hypervisors provided by the Nagios monitoring system.

The analysis pipeline model supports both low level analysis steps that operate on a single data source, and higher-level analysis steps that consume multiple input data streams. This is realized by two classes of analysis steps: *one-to-one* and *all-to-one* steps. When a matching data source is found for a one-to-one step, the analysis engine deploys a new process and instructs it to analyze the matched data stream. An all-to-one step, on the other hand, has at most one running instance at a time. That instance is instructed to analyze *all* data sources that are matched by the data source selector of the step. This type of data analysis step realizes high-level analysis algorithms, such as root-cause analysis.

Once an analysis process is deployed, it starts processing input data and usually produces one or more output data streams. These output streams are documented in the analysis step description as a list of outputs. Each entry in this list will be treated like a data source once an analysis process is running. In order to differentiate these analysis results from external raw data, additional labels are attached to the output data streams. In addition to these new labels, the output data streams inherit the labels from the analyzed input data streams. In case of multiple input data streams, only the labels that are shared by all input streams are inherited.

Figure 5.6 shows an example of a hypervisor and a VM that each provide a data source with monitoring data. The data analysis pipeline on the left consists of three steps. First, an anomaly detection step analyzes the two data streams individually. A second step processes the results of all anomaly detection steps running on the hypervisor `phys1`. Finally, the `global RCA` step consumes all intermediate results, which includes other hypervisors (not shown in this example). On the right, Figure 5.6 shows the labels of all data sources and output data streams. This example demonstrates how our modeling approach allows to create deep, hierarchical pipelines of interdependent data analysis steps.

5.3 Enforcing Resource Limitations

Besides executing the analysis pipeline model, the in situ data analysis engine has to minimize the disturbance of the main cloud workload. The main technique to this end, is to enforce resource consumption limits on all data analysis processes. The main configuration parameters for this limitation are upper bounds for the share of resources allocated for data analysis processes. These bounds cover all relevant system resources, including CPU utilization, allocated memory, hard disk access, and network bandwidth. In order to make the configuration task easier for administrators, they can configure these bounds by providing a single figure that determines the global share of all resources reserved for data analysis (e.g., 10%). Alternatively, administrators can express the resource bounds individually, either in relative or in absolute terms.

The data analysis engine is responsible for enforcing the configured resource bounds. The means for this depend on the underlying execution platform, but resource usage restrictions can be defined for bare operating system processes, containers, and virtual machines. For example, the container virtualization engine Docker¹

¹<https://www.docker.com/resources/what-container>

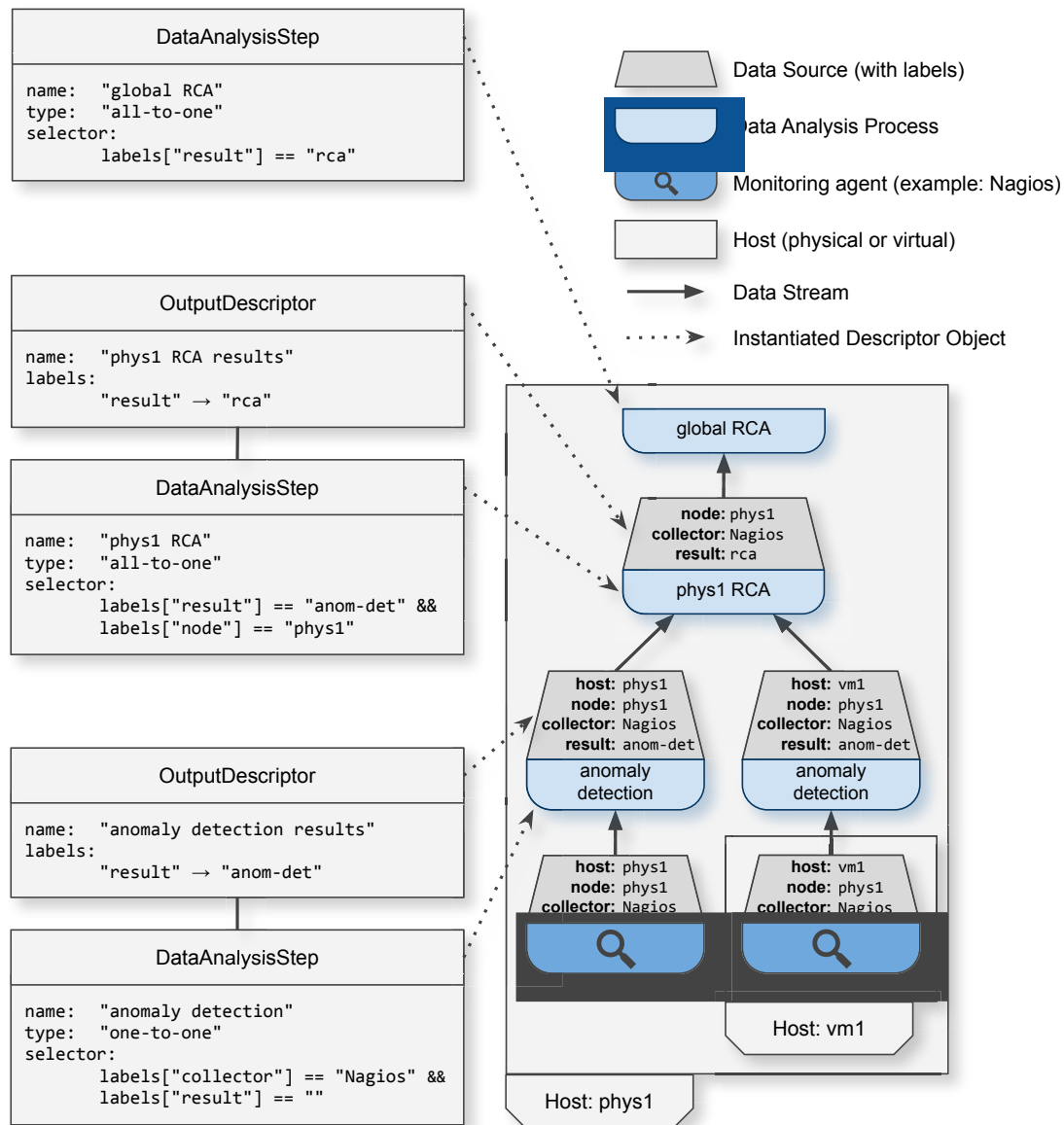
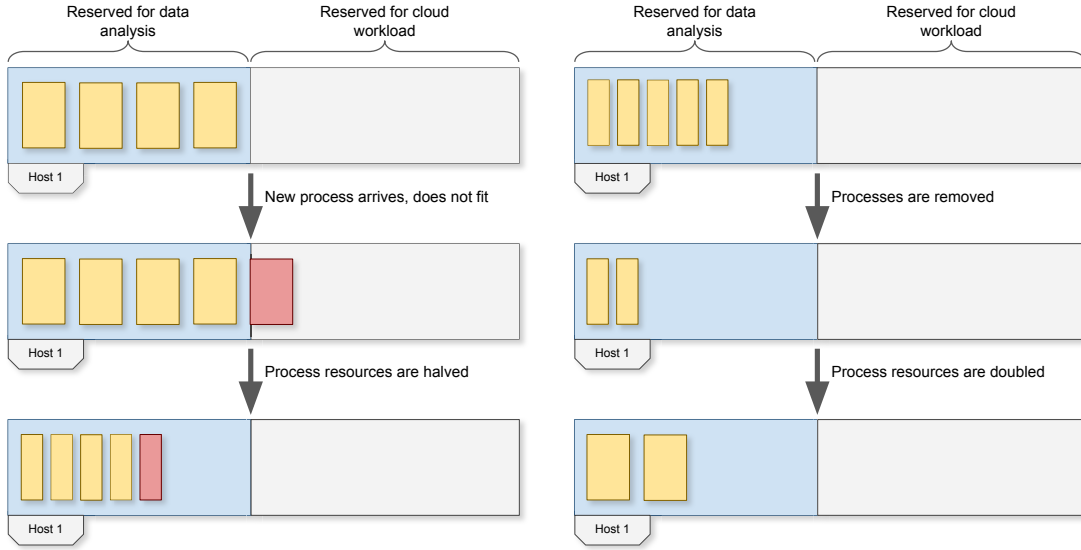


Figure 5.6: Example pipeline model and the resulting deployed analysis processes

uses the Linux kernel feature “cgroups” to define limits for the resource usage of containers [119]. Other virtualization techniques like KVM, Xen, or virtualization libraries like Libvirt [21], have similar features.

The techniques for limiting resource consumption do not allow online updates of the enforced limits. Constrained processes must be restarted when these limits are changed. To allow restarts, data analysis processes must be stateless from the engine’s point of view. We achieve this by storing snapshots of the internal data analysis model in the model repository in regular intervals. Upon being restarted, a process checks the model repository and loads the latest model snapshot. Updating the resource usage limits becomes necessary when too many analysis processes are instantiated on one host, so that the already running processes must be “shrunk” in order to fit all processes within the available resource bounds. In such a situation, the share of resources for every analysis process is recomputed and all processes are restarted. This can lead to a cascade of restarting processes when the number of processes changes frequently. We mitigate this problem by using an *exponential growth policy* when computing the share of resources given to each process. Instead of splitting *all* resources between the running processes, we leave spare resources for a number of additional processes to be started. When the spare room is filled up, the share of resources given to each process is halved, resulting in twice as many slots for running processes. The same procedure is applied when processes are stopped: Once three quarters of the reserved resources are free, the number of reserved slots is halved, resulting in a new situation where half of the resources are allocated, while the other half is reserved for starting new processes. As a result, all processes receive twice as many resources as before. Figure 5.7 demonstrates this procedure, both when increasing and decreasing the number of reserved slots for processes.

The dynamic allocation of resources can lead to a situation where the algorithm does not have enough resources to perform the necessary computations in time. To avoid an infinitely growing queue of input data, we use load shedding: We limit the size of the input for each data analysis process. In case of overload situations, samples in the queue are dropped to avoid running out of memory. In order to resolve overload situations on individual hosts, the data analysis engine implements an adaptive scheduling algorithm, as described in the following section.



(a) When resources are depleted, the number of slots for processes is doubled. As a consequence, the resources for each process are halved.

(b) When resources become available, the number of process slots is halved, and the process resources are doubled.

Figure 5.7: Exponential growth policy when starting and stopping data analysis processes

5.4 Scheduling of Data Processing Steps

An important technique for minimizing the network overhead and data processing time, is intelligent scheduling of data analysis processes. The optimization objectives of in situ data analysis differ from scheduling objectives for dedicated computational resources. Scheduling algorithms for dedicated resources (such as big data frameworks [10, 109, 168]) try to optimize the performance and fairness, while in situ data processing on shared resources needs to ensure low average resource usage to reserve sufficient resources for the main workload. We consider other secondary scheduling concerns, such as energy efficiency, out of scope for the in situ data analysis, as it only uses a small fraction of the overall compute resources.

5.4.1 Static Scheduling

For our basic scheduling approach, we treat the network topology of the hypervisors as a flat topology with equal distances between all nodes. Further, we assume that each hypervisor can host an unlimited number of processes due to the adap-

tive resource limitations described in Section 5.3. These assumptions result in a straightforward scheduling algorithm, since we only have to differentiate two cases:

One-to-one processing steps are placed on the node that contains their data source.

All-to-one processing steps are placed on the least occupied node from the list of their data sources.

5.4.2 Adaptive Scheduling

To mitigate overload situations for individual analysis processes, we add an adaptive extension to the static scheduling algorithm. When an analysis process suffers from an overload situation, it can be beneficial to dynamically move it to another node, in order to distribute the load more evenly. After such a migration, the input data stream must be forwarded from the data source to the new execution host, which results in an increased data processing time. However, the evenly distributed load, and the better resource availability on the migration target, potentially outweigh this performance penalty. Therefore, we define the following dynamic scheduling criterion: A data analysis process is migrated, when the migration would result in a lower data processing time, despite the added time for transporting the samples over the network.

In order to test our migration criterion, we have to compare the current data processing time of the processing step with the predicted processing time after the migration. We predict this processing time using a formal *processing time model*. Before defining the processing time model, we declare the following notation.

Let $T(H_1, H_2)$ denote the time to transmit a sample over the network from host H_1 to host H_2 . For the sake of simplicity, we assume a constant network latency and speed over time, and also that each sample has the same size.

Let the function $send(\mathcal{H}, H)$ denote the time to send a sample from all hosts in the host set $\mathcal{H} = \{H_1, \dots, H_n\}$ to the host H . This time is zero when forwarding the sample within the same host, and it is equal to the largest forwarding time, when sending data from at least one remote host, since all samples are sent in parallel:

$$send(\mathcal{H}, H) = \begin{cases} 0 & \text{if } \mathcal{H} = \{H\} \\ \max_{H_i \in \mathcal{H}} (T(H_i, H)) & \text{else} \end{cases} \quad (5.1)$$

Let $compute_A(R)$ denote the time for the algorithm A to compute the results for one sample, when it is executed on the resources R .

Now we can define the data processing time of an algorithm A as the function $P_A(\mathcal{H}, H, R)$, which is the sum of sending all samples from the n data sources $\mathcal{H} = \{H_1, \dots, H_n\}$ to host H , and the time to compute the result:

$$P_A(\mathcal{H}, H, R) = \text{send}(\mathcal{H}, H) + \text{compute}_A(R) \quad (5.2)$$

If we can calculate P_A for every combination of hosts, algorithms, and resources, we can use it to test our migration criterion the following way:

1. Compute $P_A(\mathcal{H}, H, R)$, where H is the current execution host and R the currently available resources.
2. Compute $P_A(\mathcal{H}, H^*, R^*)$ for every potential migration target H^* , setting R^* to the resources that would be available on that host.
3. If one of the migration targets leads to a lower result than the current execution host, migrate the process there.

In order to calculate $P_A(\mathcal{H}, H, R)$, we need to compute the sub terms $\text{send}(\mathcal{H}, H)$ and $\text{compute}_A(R)$. Computing $\text{send}(\mathcal{H}, H)$ depends on the values $T(H_i, H)$ for all $H_i \in \mathcal{H}$, which we obtain by observing the data stream characteristics during runtime. Due to the assumption that the network latency and size of samples remains constant, we do not model dynamic characteristics of the network. We model the compute time $\text{compute}_A(R)$ with the following parametric function:

$$\text{compute}_A(R) = a * (R + b)^{-c} + d \quad (5.3)$$

This polynomial function with four parameters models a declining compute time with rising resources. Figure 5.8 illustrates this parametric function, and how it affects our migration criterion for a one-to-one step that runs on the same host as its data source. Intuitively, the time to compute a result increases with decreasing resources. With zero resources, the processing time is infinite. However, the processing time also never falls below a threshold defined by the parameter d , which is bounded by the algorithm's complexity and by the underlying hardware. The resources can be reduced without any implications, until $\text{compute}_A(R)$ starts rising. At this point, we can continue reducing the resources, until $\text{compute}_A(R)$ increases by more than the network transportation time to a remote host. Now, if we have to reduce the resources even further, it is more beneficial to migrate the analysis process to a remote host.

In order to compute the parametric function $\text{compute}_A(R)$, we have to find fitting values for the parameters a , b , c , and d . These values are specific for a combination

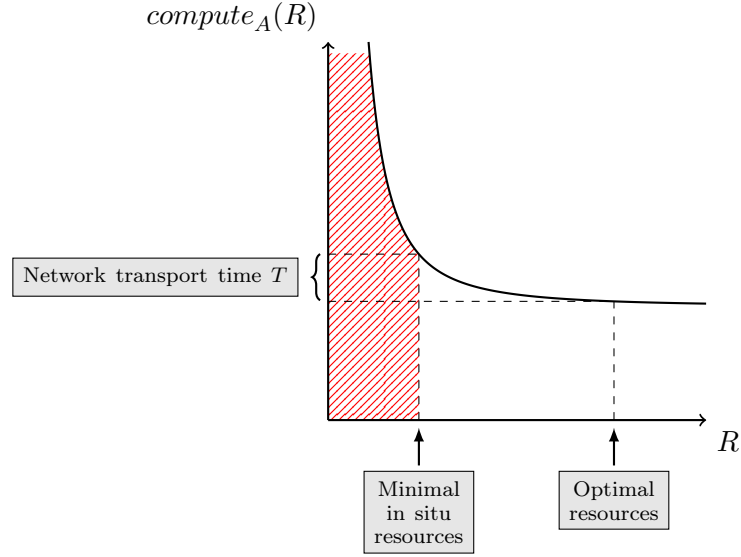


Figure 5.8: Predicting the processing latency after migrating the analysis process

of an algorithm and an underlying execution platform. The procedure to obtain the parameter values consists of a series of controlled experiments. First, the algorithm is executed with the maximal available resources, while measuring the average time it takes for it to process one sample. Then the resources are reduced in regular steps, while continuing to take the processing time measurements. This procedure produces a set of sample points for the $compute_A(R)$ function. In the final step, we fit the parametric function $compute_A(R)$ to the collected points, using the Levenberg-Marquardt algorithm for non-linear least squares optimization [104, 130]. The result is a $compute_A(R)$ function with specific parameter values, which predicts the processing time of the algorithm for arbitrary resources on the underlying hardware.

To demonstrate the applicability of our approach, we have performed the procedure to obtain the parameters for $compute_A(R)$ for three different algorithms. The evaluated algorithms are based on the *Identity Function and Threshold Model* (IFTM) [151] principle, which we use for anomaly detection in the zero-touch operations system (see Section 4.3.2). IFTM allows to select different underlying models. The first evaluated model is based on a neural network with Long-short term memory (LSTM) neurons [79]. The second model uses multivariate online ARIMA [153], an approach for modeling and predicting periodic time series. The third model is not based on neural networks, but instead uses a variation of the

clustering algorithm BIRCH [185], optimized for evolving data streams. We performed the experiments to obtain the necessary data on a physical machine with a Quadcore Intel Xeon CPU (E3-1230 V2 3.30GHz), 8 virtual cores, and 16 GM of RAM. In each experiment run, a data set consisting of 10.000 samples, with 28 metrics per sample, was processed by the respective algorithm, while measuring the average processing time per sample. Starting with 8 virtual CPUs (which results in no resource limitation at all), the number of CPUs allocated for the process were reduced in steps of 0.3 after each experiment run, until reaching the minimum assignable resources at 0.1 CPUs.

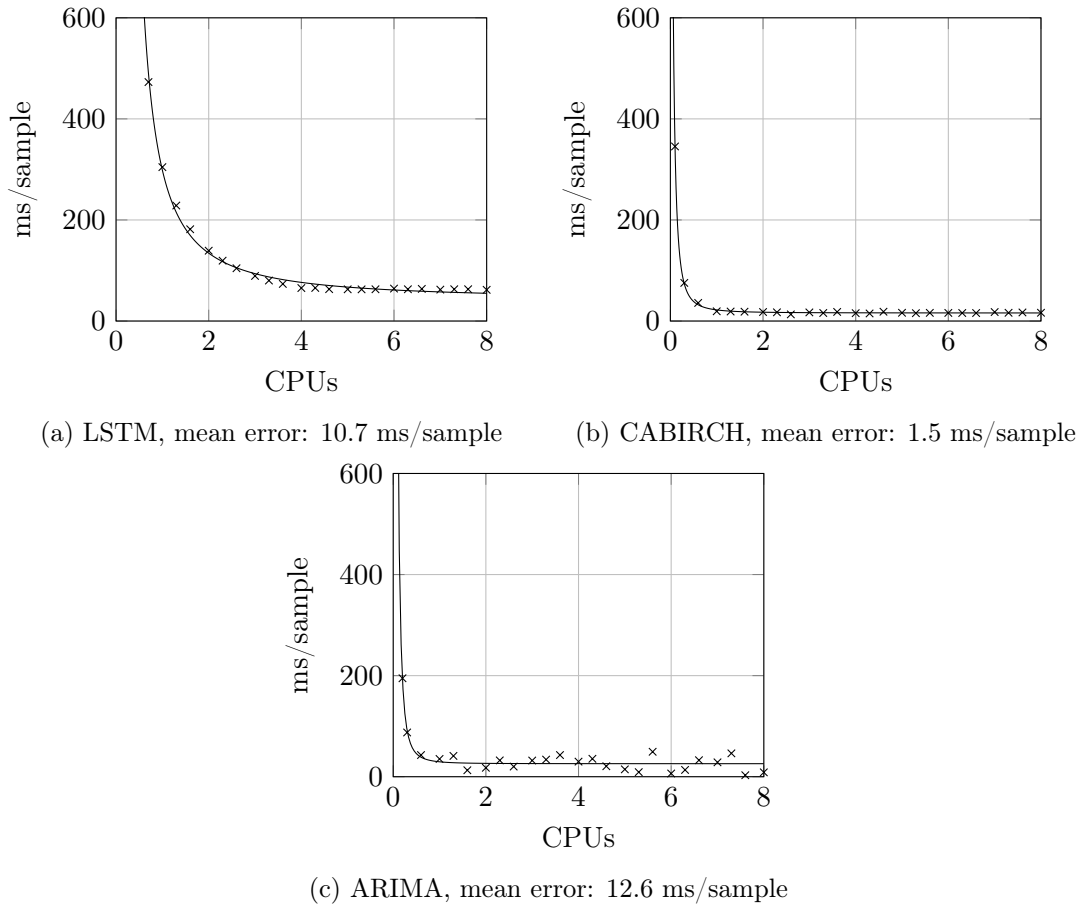


Figure 5.9: Modeling the processing time of different classes of algorithms with the parameterized $compute_A(R)$ function

Figure 5.9 shows the results of our experiments. The crosses show the experimentally collected samples, while the lines show the resulting parameterized

$compute_A(R)$ function. All algorithms show the expected behavior of rapidly increasing processing time with dropping resources. For every algorithm, we computed a *mean error* values, which is the mean absolute deviation of the fitted $compute_A(R)$ from the actual measurements. The low mean error values, which we obtained for the three algorithms, confirm that our parameterized model of $compute_A(R)$ is well-suited to represent their behavior.

To summarize, we propose a dynamic scheduling approach that optimizes the data analysis time of resource-constrained processes, while considering both the network transportation of the input samples and the computation time itself. Our scheduling criterion can predict overload situations that originate from resource limitations, instead of handling such situations reactively. Our approach is based on a data-driven model of the algorithm's computation time, depending on the allocated CPU resources. Using three example algorithms, we showed that our computation time model is applicable to a wide range of algorithm classes.

Chapter 6

ZerOps: A Self-Healing Cloud Platform

Contents

6.1	Use Cases	89
6.2	Components of the ZerOps Platform	92
6.2.1	High-Availability Deployment of OpenStack	94
6.2.2	Topology Discovery	95
6.3	Bitflow Stream Processing Framework	96
6.3.1	Bitflow Controller	97
6.3.2	Bitflow Stream Processing Runtime	100
6.3.3	Bitflowscript Language	103
6.3.4	Bitflow Data Collector	104
6.4	Experimental Evaluation	107
6.4.1	Tenant Service Scenarios	107
6.4.2	Anomaly Injection	110
6.4.3	Testbed and Experimental Procedure	113
6.4.4	Evaluation Results and Discussion	115

In order to evaluate our architectural concepts, we derive an implementation of the zero-touch operations architecture, which is usable in a number of specific use cases. This system – *ZerOps* – implements our architecture from Chapter 4, as well as the in situ data analysis engine from Chapter 5.

We base ZerOps on a number of assumptions that drive the selection of system components. First, we assume that the operator of the cloud infrastructure, includ-

ing the hardware and the virtualization layer, installs and maintains the ZerOps system as well. The service model of the cloud platform is IaaS, meaning the clients (or tenants) manage and connect VMs, and install their application software on VMs as well. Furthermore, from the cloud provider's point of view, client VMs are black boxes and cannot be accessed internally. Finally, every client VM runs one instance of an application service and the services are considered stateless. These assumptions support a number of use cases that differ in the size of the cloud platform and in the relationship between the cloud provider and the owner of the application.

The ZerOps platform allows us to perform both a qualitative and quantitative evaluation of our conceptual architecture. For the qualitative evaluation, we refer to the requirements and challenges that we identified in Chapter 1, and discuss, to what degree the zero-touch operations architecture fulfills them. A series of experimental measurements in a running instance of ZerOps provides further evidence for the fulfillment of the requirements. Specifically, the quantitative evaluation covers the following:

- The resource overhead introduced by the data collection and data analysis processes, and
- the amount of network traffic that is saved by working with the in situ data analysis engine.

All components of ZerOps are available as open source software¹. We have implemented the stream processing engine *Bitflow*, which provides data collection and data analysis capabilities for ZerOps. Besides the core component Bitflow, ZerOps also includes a number of tenant service scenarios and an anomaly injection service for conducting controlled experiments. Therefore, ZerOps also functions as an experimental platform for evaluating algorithmic solutions for different tasks in the zero-touch operations architecture. ZerOps contributes to the research of such algorithms by facilitating the collection of realistic data that is suitable for training and evaluation purposes. Since this data is collected in a functional infrastructure, it is preferred over simulations or emulations of similar cloud platforms. In experiments with the ZerOps platform, we have evaluated algorithms that were specifically designed for the zero-touch operations use case. Furthermore, we have tested the ZerOps platform on the infrastructure of a large vendor of telecommunications equipment. ZerOps was also adapted for multiple industrial research projects in the domain of IoT and smart factory.

¹We provide links to the publicly available repositories within this chapter.

Section 6.1 defines the use cases that form the scope of ZerOps and drive its design decisions. Section 6.2 gives an overview over the ZerOps platform and describes the underlying cloud infrastructure. Section 6.3 documents the stream processing framework Bitflow, which is at the core of the ZerOps platform. Finally, Section 6.4 presents how conduct evaluation experiments can be conducted with the ZerOps platform, and discusses our evaluation results.

6.1 Use Cases

This section lists and describes a number of use cases that operate on the assumptions behind the ZerOps platform. The purpose of these use cases is to illustrate that the system design of ZerOps applies to a variety of infrastructures and scenarios used in practice.

Use Case: Public Cloud Provider

The first use case for the ZerOps system is a public cloud provider that offers IaaS capabilities to its clients. In this use case, the cloud provider maintains one or more centralized data centers with virtualization servers. The physical machines that run the cloud are commercially available off-the-shelf servers. The cloud provider uses the self-healing capabilities of ZerOps to supervise the cloud infrastructure and cloud services, and offers the client the optional service to put individual VMs under supervision as well. Since the VMs are black boxes and contain private data of the respective client, their behavior must be analyzed in a non-intrusive way, without service-specific metrics, and without monitoring agents running within the guest operating system. The first important data source for VMs is the virtualization engine, which provides resource usage data. The communication behavior on the virtual network provides additional black-box metrics that describe the VMs behavior.

Because of the privacy constraints of tenant VMs, the set of possible remediation workflows is limited in this use case. Application-specific workflows are not supported for the same reasons that service-specific data cannot be extracted from the VMs. However, this scenario still allows for generic remediation actions on the virtualization layer, such as restarting or migrating a virtual machine, or changing the number of replicas of a service. For more critical applications or services, the tenant can choose to receive notifications from the data analysis part of ZerOps, in order to handle or post-process such alarms.

Use Case: Telecommunication Service Provider

The networks of telecommunication service providers (or carriers) are amongst the most reliable computer and communication systems. Telecommunication services are *critical services* in the sense that they have very high requirements regarding reliability, availability, security, and performance. The term *carrier-grade* availability, when applied to common systems in order to measure the percentage of uptime, denotes an especially high level of availability. Over decades, service providers have engineered an extensive range of sophisticated features into their networks in order to guarantee an uptime of 99.9999%, which corresponds to a downtime of as little as 32 seconds per year.

This high level of reliability is achieved through specialized *appliances*: combinations of hardware and software that are designed and tested to fulfill very specific tasks. The *hardware-software co-design* is the key to the reliability of appliances. The software is entirely optimized for the underlying physical components and does not share any resources with arbitrary tasks, as in general purpose operating systems or hypervisors.

The obvious downside of appliances is their cost. Development cycles for co-designed hardware and software are very long, as well as testing phases and ultimately the time-to-market for new services. On today's fast-paced market, the time to release a new product or service is critical for the commercial success of a telecommunication service company.

As an effort to decrease the time-to-market and increase cost-efficiency, telecommunication service providers have started to investigate cloud platforms as a potential deployment strategy [13, 182]. The telecommunication sector of the IT industry is one of the last to migrate their services to virtualized infrastructures, due to the services' critical nature. This movement is entitled Network Functions Virtualization (NFV) [2, 51, 76, 121], and introduces a significant degree of flexibility and chances for cost reduction. So-called Virtualized Network Functions (VNFs) are services that were migrated from their traditional appliance-based implementation to a cloud-based, virtualized environment. Typical VNFs operate on lower layers of the network stack, redirect network streams (virtual routers [50, 172]), secure them (virtual firewalls [47, 114]), virtualize them (virtual overlay networks), or coordinate network flows between autonomous segments of the internet (virtual BGP routers). Some VNFs operate on the application layer and provide communication related services, such as the Virtualized IP-Multimedia Subsystem (vIMS), a suite of protocols for dialing and establishing connections between communication peers

[31].

For ZerOps, the use case of a telecommunication service provider differs from the use case of a public cloud mainly in the relationship between the maintainer of the cloud infrastructure and the application developer. In order to guarantee a high level of reliability, and to comply with privacy-related SLAs, the service provider owns and maintains the cloud infrastructure. The virtualized network functions deployed on this infrastructure are also developed, or at least used, by the same company, but it is likely that different teams within the company handle the aspects of cloud infrastructure and application development. Furthermore, this scenario does not have the restriction of black box VMs from the cloud provider's point of view. This allows for an extended set of service-specific metrics and remediation workflows executed directly within the anomalous VMs and services.

Use Case: Edge Computing

Technological advancements in the field of telecommunication services, such as the future mobile network standard 5G [7], enable the development of a new class of applications that benefit from geographical proximity to the user. Having the server side of an application close to the client has multiple advantages:

- lower network latency between the client and the server,
- less network hops, hence a more predictable connection quality,
- less data transferred through the backbone network to the centralized data center or cloud, and
- increased geographical scale-out of the application.

Many applications benefit from such an architecture. Virtual reality applications require very low update latencies to be perceived as natural [164]. Critical applications such as self-driving cars, controllers in industry 4.0 factories, smart cities, and wearable healthcare devices, all benefit from stable server communication and quicker response times.

We envision cloud providers or Internet Service Providers (ISPs) offering special computational resources at the edge of the network. Such edge clouds could be located at broadband base stations or access networks of ISPs and consist of a combination of commodity hardware and open source software. However, due to the favorable positioning, resources in edge clouds are costly and limited in their capacity. This enables the ISP to offer a new class of services for monitoring, anomaly detection, and automated remediation. Of course, such services must still comply with the service agreements between the ISPs and their customers. These

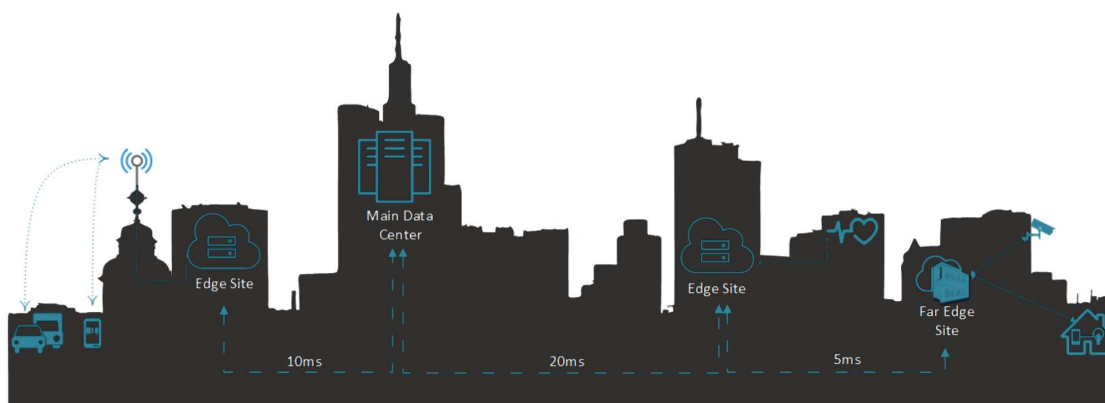


Figure 6.1: Overview of an example edge computing infrastructure in the context of smart cities. It shows several edge sites, a main data center, the interconnectivity between them, as well as several use cases.

agreements prohibit the ISP from directly accessing their customers' computing resources and stored data, which means that the monitoring services must treat the customer software as black boxes.

Effectively, this scenario resembles the IaaS use case for ZerOps, except that the cloud infrastructure is geographically distributed and consists of more heterogeneous servers. The ZerOps platform supports this scenario due to the adoption of our distributed, hierarchical in situ data processing engine. ZerOps treats edge clouds with limited resources differently from cloud resources in the central cloud. Furthermore, ZerOps is designed to limit monitoring data sent over the network.

6.2 Components of the ZerOps Platform

ZerOps combines multiple standalone open source software solutions into one coherent platform. Figure 6.2 shows the high-level components of the ZerOps platform. The blue boxes represent components of ZerOps that we implemented specifically for this purpose, while the light boxes represent the cloud platform and other existing third-party software.

ZerOps uses the open source cloud operating system OpenStack. Tenants and clients of the ZerOps platform use the default OpenStack dashboard and API for managing their VMs and workloads. In fact, ZerOps makes no assumptions about the underlying IaaS cloud platform, except for using the OpenStack-specific HTTP-based API. Therefore, other cloud systems can be integrated with ZerOps.

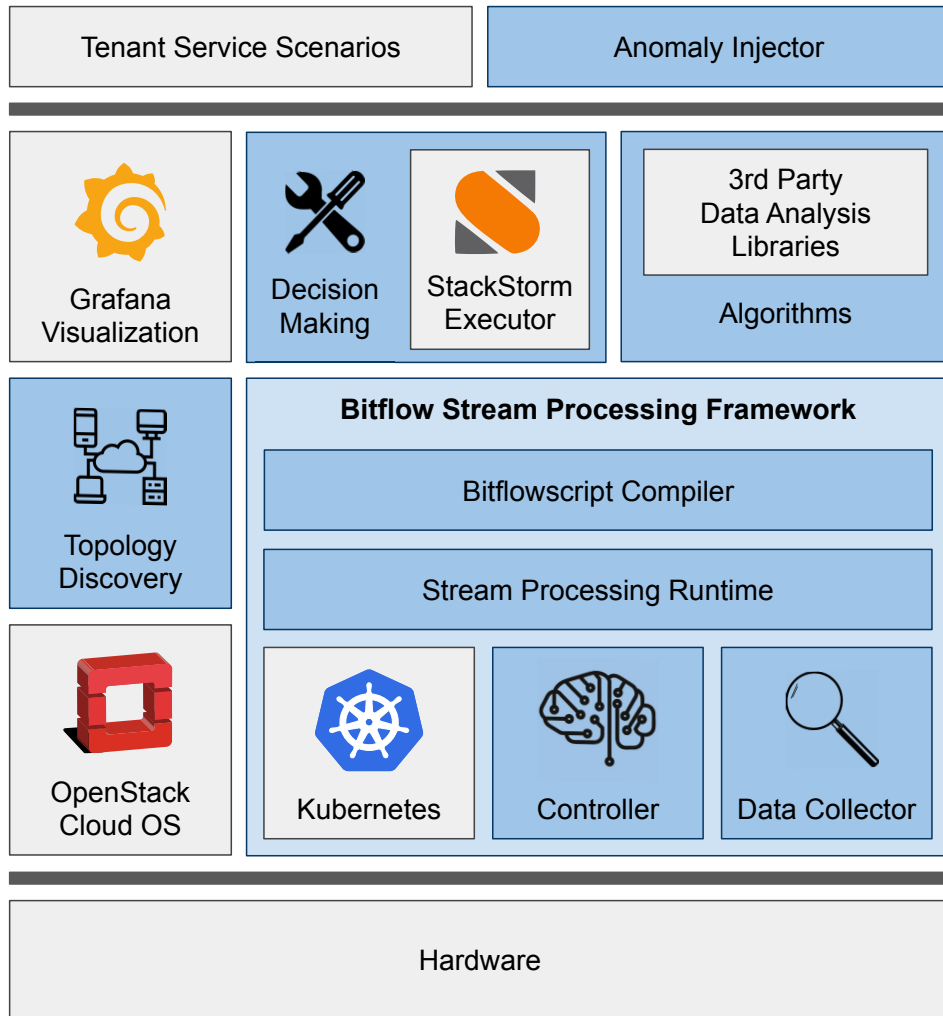


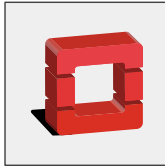
Figure 6.2: ZerOps High-level Architecture

We implemented the stream processing framework Bitflow to provide the data collection and stream data analysis capabilities of ZerOps. Bitflow follows the design of the in situ data analysis from Chapter 5. The Bitflow Controller implements distributed stream processing on top of the Kubernetes container orchestration platform². Therefore, ZerOps runs Kubernetes on the compute nodes of OpenStack, parallel to the cloud platform and its tenants. The Bitflow Data Collector consists of an agent that runs on every hypervisor node and provides data streams that describe the behavior of the hypervisor and the contained VMs.

²<https://kubernetes.io/docs/home/>

On the service level, ZerOps features a number of benchmark workloads to generate realistic resource usage patterns on the cloud. To facilitate experiments with anomalies and failing services, we added an Anomaly Injector service to ZerOps specializes on resource anomalies.

6.2.1 High-Availability Deployment of OpenStack



ZerOps relies on the open source cloud operating system OpenStack to handle the management of tenant VMs and virtual networks. OpenStack implements an IaaS cloud platform that can be used both in a private cloud and in a public cloud scenario. This supports all defined use cases for ZerOps.

A number of cooperating services provide the core functionality of OpenStack. This functionality includes user and session management (implemented by a service named Keystone), VM management (implemented by Nova), image storage (implemented by Glance), block storage (implemented by Cinder), and web frontend (called Horizon). In order to guarantee high availability of the entire cloud platform, ZerOps replicates all of these services individually. ZerOps uses the application-layer load balancer HAProxy³ to distribute requests between the OpenStack services. To eliminate the load balancer as a single point of failure, the load balancer itself is replicated following an active-passive scheme. One load balancer is active at each time, and the active load balancer occupies a virtual IP address. This virtual IP address is used by clients of the cloud platform to access all OpenStack services. In case of a load balancer failure, the virtual IP is reassigned to the standby load balancer instance using the *keepalived*⁴ service. The *keepalived* daemon monitors the active load balancer and detects failures in a very short time span, in order to ensure a fast failover and transfer the virtual IP. As soon as the IP address is transferred, new client requests are routed to the now active load balancer. Clients might have to resubmit a currently pending request, which is unproblematic since the OpenStack API is stateless.

To further increase OpenStack's reliability, all services use a Ceph⁵ installation as a sophisticated distributed storage backend. Ceph runs on separate storage hosts and replicates every stored data item to ensure no data is lost when a disk or storage node fails. OpenStack stores all data related to the cloud platform in a database backed by Ceph. Furthermore, when OpenStack boots VMs, it

³<http://www.haproxy.org/#docs>

⁴<https://www.keepalived.org/manpage.html>

⁵<https://docs.ceph.com/docs/master/>

provides them with a virtual network volume that is effectively stored in Ceph. This feature decouples the execution host of the VM from the VM storage and enables live migration of VMs between different hypervisors. Live migration is an important feature for some remediation workflows.

Figure 6.3 visualizes ZerOps’ highly available, bare-metal OpenStack deployment with the aforementioned features. We have evaluated the visualized deployment of OpenStack and concluded a sufficiently low failover time of the chosen components [106]. Furthermore, the impact of failures of cloud operating system components on the actual service layer is very low. Even when the controller services of OpenStack fail, the tenant VMs and virtual networks continue to operate normally.

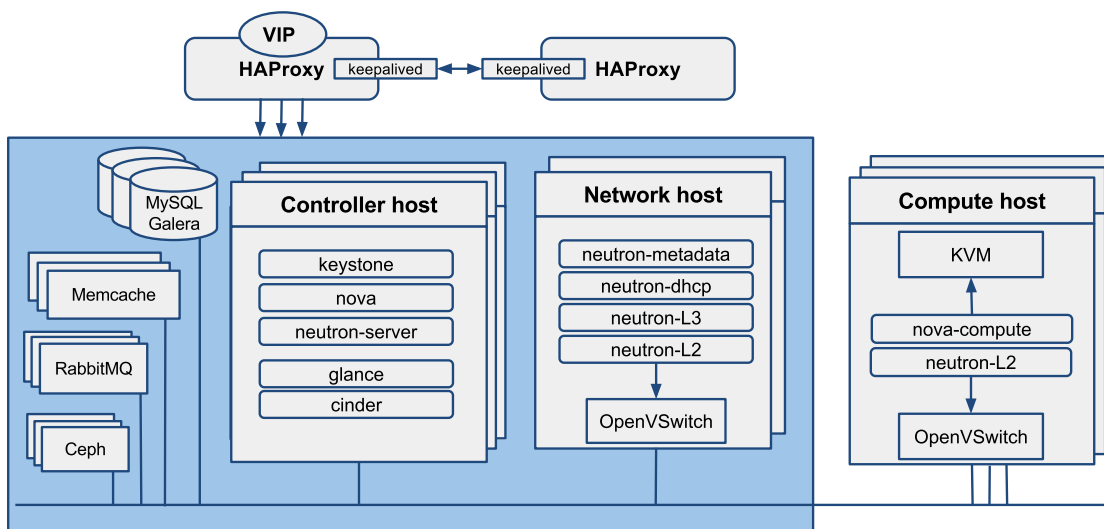
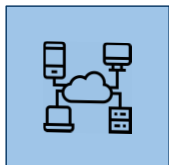


Figure 6.3: Redundant OpenStack cloud platform in ZerOps

ZerOps implements all communication with OpenStack through an abstract cloud communication layer. Therefore, other cloud systems can be integrated with ZerOps with limited development effort.

6.2.2 Topology Discovery



Various parts of ZerOps depend on an up-to-date view of the monitored cloud system’s topology. The cloud system consists of physical nodes, VMs, as well as physical and virtual networks. ZerOps’ topology discovery component collects this information and provides it in

a central graph database for easy access by all other components. ZerOps uses the graph database Neo4j to store topology information. Neo4j allows to store graph nodes and edges, as well as arbitrary properties attached to them.

The topology discovery uses OpenStack as main source of information. The OpenStack API provides access to hypervisors, VMs, and virtual networks, VM images, users, and more. ZerOps maps this data to a coherent graph representation. All available information about the mapped entities is included in the Neo4j graph model. Due to this dependency on the OpenStack API, the topology discovery component is tightly coupled to OpenStack, unlike any other component in ZerOps.

ZerOps splits the process of mapping the topology graph in two phases. First, the entire OpenStack data model is loaded, and the Neo4j database is updated. This process is rather expensive and puts a lot of stress on the cloud system due to the many requests to the API. Therefore, this “full scan” of the topology is performed once initially, and from then on in regular time intervals. To keep the topology graph up to date with changes in the cloud system, ZerOps relies on OpenStack’s event mechanism. ZerOps configures the OpenStack platform to send events on its internal message queue, whenever the topology of the cloud changes. By subscribing to the relevant message queue channel, ZerOps receives all update events from OpenStack as push-notifications. These update events are directly mapped to updates of the Neo4j graph model. This mechanism allows to maintain an up-to-date topology graph of the cloud infrastructure, without frequently parsing all information delivered by the OpenStack API. Full scans of the topology are still repeated regularly to make sure the graph model remains consistent with the real system.

6.3 Bitflow Stream Processing Framework

The Bitflow framework was specifically designed to provide the data collection and data analysis for ZerOps. The source code for Bitflow is available online⁶, as well as the documentation⁷. Bitflow implements a stream processing runtime, a data transmission format, a domain-specific script language, a controller for distributed processing, and a time series data collector.

⁶<https://github.com/bitflow-stream>

⁷<https://bitflow.readthedocs.io/en/latest/>

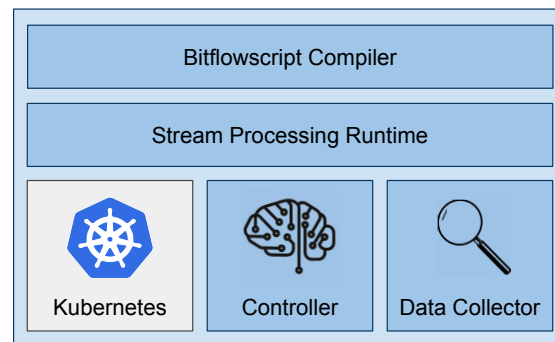


Figure 6.4: The components of the Bitflow Stream Processing Framework

6.3.1 Bitflow Controller



The Bitflow Controller implements the in situ data analysis engine following our design introduced in Chapter 5. The in situ design allows ZerOps to execute Bitflow parallel to the OpenStack cloud platform.

Bitflow uses Kubernetes as the execution platform for data analysis containers. Kubernetes is a distributed orchestration platform for containers and supports many different abstractions for expressing, where and how containers are executed and connected. The built-in capabilities of Kubernetes include the execution of individual containers, automatic replication and scaling of identical container instances, precise control over the target execution node, and hard resource usage limitations for each container.

Bitflow extends the native capabilities of Kubernetes by making use of the Kubernetes *operator pattern*⁸. A Kubernetes operator is a component that continuously and automatically manages Kubernetes resources. The most common use case is to automatically create, schedule and configure containers based on some domain specific conditions. To support this, Kubernetes offers the feature of *custom resource definitions*. It allows storing user-defined JSON data structures in Kubernetes' native, distributed object store, where they can be accessed by the operator process. The user-defined objects describe the desired state of managed Kubernetes resources in a declarative way. The operator then compares this description to the actual runtime state of the system and performs the necessary actions to approximate them. These actions usually involve the creation, deletion, or reconfiguration of containers, but can also involve other resources like networks, DNS

⁸<https://kubernetes.io/docs/concepts/extend-kubernetes/operator>

entries, or load balancers.

The Bitflow Controller works with two custom resource types, which are based on the data analysis pipeline model described in Section 5.2. A `BitflowSource` describes the location and meta data of a data source, while a `BitflowStep` defines how to instantiate a data analysis container and what data sources it must be attached to. Listing 2 shows examples in YAML syntax for a data source and an analysis step. In this example, the `BitflowSource` object has the label `layer: physical`, which is matched by the `ingest:` section of the `BitflowStep` object. The ZerOps operator will spawn a container based on the `spec.template` section inside the `BitflowStep` definition.

```

1  apiVersion: bitflow/v1
2  kind: BitflowSource
3  metadata:
4    name: bitflow-collector-node01
5    labels:
6      collector: bitflow
7      component: node01
8      layer: physical
9      node: node01
10 spec:
11   url: tcp://192.168.0.30:9000

```

(1) BitflowSource

```

1  apiVersion: bitflow/v1
2  kind: BitflowStep
3  metadata:
4    name: anomaly-detection
5  spec:
6    ingest:
7      - key: layer
8        value: physical
9    outputs:
10     - name: results
11       url: "http://0.0.0.0:9000"
12       labels:
13         type: anomaly-detection
14    template:
15      metadata:
16        labels:
17          app: anomaly-detection
18      spec:
19        containers:
20         - name: anomaly-detection
21           image: anomaly-detection
22           imagePullPolicy: Always
23           ports:
24             - name: results
25               containerPort: 9000
26         command:
27           - "-input"
28           - "${BITFLOW_SOURCE_URL}"
29           - "-output-port"
30           - "9000"

```

(2) BitflowStep

Listing 2: Example definition in YAML syntax of a `BitflowSource` and a `BitflowStep`

Bitflow supports three types of processing steps with different semantics regarding the amount and location of the resulting containers:

one-to-one steps: Bitflow instantiates one container for every matched data source. Each container is placed as close as possible to its data source. In most cases this means that the data source and the processing step run on the same physical host, which minimizes the network overhead and the communication path between the container and its data source.

all-to-one steps: Bitflow instantiates at most one container that is instructed to analyze *all* matched data sources.

singleton steps: Bitflow starts the defined container, but does not connect it to any data source. This type of step is useful to implement miscellaneous containers, such as databases or web servers. This way, all tasks and containers in the ZerOps system can be defined and executed in the same format – not only tasks strictly for data analysis.

When spawning a container, the Bitflow Controller provides it with all contextual information that the container needs to perform its task. The information is injected by two means: by setting environment variables and by replacing placeholder tokens in the command section of the container template. Table 6.1 summarizes the information that is provided this way. In the example of Listing 2, the `{BITFLOW_SOURCE_URL}` token in the command list is replaced by the URL of the data source, which makes sure that the data analysis process knows what data stream to attach to.

`BitflowStep` objects offer great flexibility to the designer of the data analysis. By attaching `BitflowSteps` to the output data streams of other `BitflowSteps`, Bitflow supports building deep and dynamic data analysis pipelines. `BitflowSources`, on the other hand, are typically inserted into the Kubernetes object store by an automatic process. The Bitflow Collector, for example, creates a `BitflowSource` object whenever a new VM appears on the monitored hypervisor.

The Bitflow Controller implements automatic resource limitation of data analysis containers. For each physical node of the cloud platform, a configurable share of the available resources is reserved for the analysis processes. When spawning a container, the Bitflow Controller calculates the resources that the container receives, and adds them to the container definition that it passes on to Kubernetes. Kubernetes uses the “cgroups” feature of the Linux kernel [119] to implement the

Table 6.1: Contextual information provided to data analysis containers by the Bitflow Controller.

Field Name	Description & Example
BITFLOW_SOURCE_URL	The URL for connecting to the input data stream Example: <code>tcp://192.168.0.30:9000</code>
BITFLOW_SOURCE_NAME	The name of the BitflowSource object that caused this container to spawn Example: <code>bitflow-collector-node01</code>
BITFLOW_SOURCE_LABELS	The labels of the data source that caused this container to spawn Example: <code>node=node01, layer=physical</code>
BITFLOW_STEP_NAME	The name of the BitflowStep of this container Example: <code>anomaly-detection</code>
BITFLOW_STEP_TYPE	The type of the BitflowStep of this container Example: <code>one-to-one</code>

resource limitations for the processes it controls. When computing the resources assigned to each container, the Bitflow Controller uses an exponential growth strategy, which minimizes the number of restarted containers when analyzed data streams are added or removed.

6.3.2 Bitflow Stream Processing Runtime

The Bitflow Controller schedules and configures data analysis processes on distributed hosts, but it does not implement any actual stream processing capabilities – this is the task of the Bitflow Stream Processing Runtime (short: Bitflow Runtime). The Bitflow Runtime consists of three parts:

- a sample data type and a programming model to process samples in a graph of processing steps,
- a binary marshalling format and efficient protocol for transmitting or persisting samples,
- a domain-specific scripting language (Bitflowscript) that describes graphs of processing steps.

One instance of the Bitflow Runtime runs as a single process on one execution host, the Bitflow Controller starts many such processes to implement distributed

stream processing. This section describes the programming model and marshalling format of the Bitflow Runtime, while the following section defines the syntax and semantic of Bitflowscript.

To increase the flexibility of the Bitflow Runtime, we have implemented the programming model, network protocol, and Bitflowscript in three programming languages: Go (`go-bitflow`⁹), Java (`bitflow4j`¹⁰), and Python (`python-bitflow`¹¹). These implementations of the Bitflow Runtime are compatible, since they support the same Bitflowscript syntax and the same network protocol. samples can be seamlessly transported between data analysis processes implemented in different languages. Depending on the nature of the required data analysis, different languages offer different capabilities and libraries. The Go programming language offers good performance and primitives for parallel programming. Python and Java, on the other hand, provide sophisticated frameworks for data analysis and machine learning, such as Tensorflow¹².

The core data type of the Bitflow Runtime is a sample. A sample represents one data point or measurement, while a sequence of samples represents a multi-dimensional time series. Table 6.2 lists the data fields that are contained in a Bitflow sample. The Bitflow Runtime processes streams of samples in a pipeline-oriented programming model. A Bitflow pipeline can be composed of the following elements: data sources, data sinks, processing steps, forks, and batch steps.

Bitflow data sources parse data from external sources into the internal sample format. After a data source parses a sample, it passes it to the subsequent pipeline element. Bitflow can read samples from files, TCP and HTTP connections, and the process' standard input stream. In addition, Bitflow supports the time series data base InfluxDB¹³ and Elasticsearch¹⁴.

Data sinks in Bitflow mirror their data source counterparts – i.e., Bitflow can write data to files, network connections, the standard output, and external data bases such as InfluxDB. For transporting data over the network, or storing samples in files, Bitflow defines a dense binary marshalling format. A Bitflow binary data stream begins with a magic byte sequence, followed by the names of the fields transported in the stream. These header fields are only re-transmitted when the

⁹<https://github.com/bitflow-stream/go-bitflow>

¹⁰<https://github.com/bitflow-stream/bitflow4j>

¹¹<https://github.com/bitflow-stream/python-bitflow>

¹²<https://www.tensorflow.org/overview>

¹³<https://docs.influxdata.com/influxdb/>

¹⁴<https://www.elastic.co/guide/index.html>

Table 6.2: The sample data type of the Bitflow Runtime.

Field Name	Description	Example
timestamp	The time when the sample was originally recorded or generated	2019-11-05 20:05:56.984
header	A list of names of the data fields contained in this sample	cpu, mem-percent, net-io/packets
values	The list of floating point values of this sample	42.488, 58.166, 285751.408
tags	A dictionary of string key-value pairs that describe meta-properties of this sample	node=node01, layer=physical, collector=bitflow

transported fields change. The actual samples are marshalled by encoding their timestamp as a binary unsigned 64-bit integer, followed by their tags and by the binary representation of the actual sample values. The detailed specification of the marshalling format is available online¹⁵.

Regular processing steps in Bitflow receive a stream of samples, perform calculations or modifications, and forward resulting samples to the subsequent processing step. Bitflow supports a number of useful steps for transforming streams of samples. Samples can be filtered based on tags and values of fields. Individual tags or fields can be removed or modified. Samples can be sorted, shuffled, or otherwise reordered. Many mathematical operations are supported as well, such as computation of value distributions, normalization and smoothing of values, Principal Component Analysis, Fast Fourier Transformation, and others. More specialized data analysis algorithms can be added to the toolbox by implementing a simple interface in one of the supported programming languages.

Bitflow's pipeline model offers control flow and parallelization capabilities by defining *forks*. A fork splits a stream of samples into multiple parallel sub-pipelines. Incoming samples are distributed into one or more sub-pipelines based on their tag values. For example, the user can process data from different nodes individually by defining a separate sub-pipeline for every value of the node tag of incoming samples. Furthermore, Bitflow supports batch operations by buffering samples and passing them to the processing step implementation collectively. A batch pro-

¹⁵<https://bitflow.readthedocs.io/en/latest/data-format/>

cessing step can be defined with a fixed size, constructed based on the timestamp of samples, or simply built from all data in a finite data source.

Figure 6.5 shows how multiple Bitflow pipelines can be combined to form a complex data analysis chain. Each instance of the Bitflow Runtime runs as an individual process and can be started on the same machine as its predecessor, or remotely.

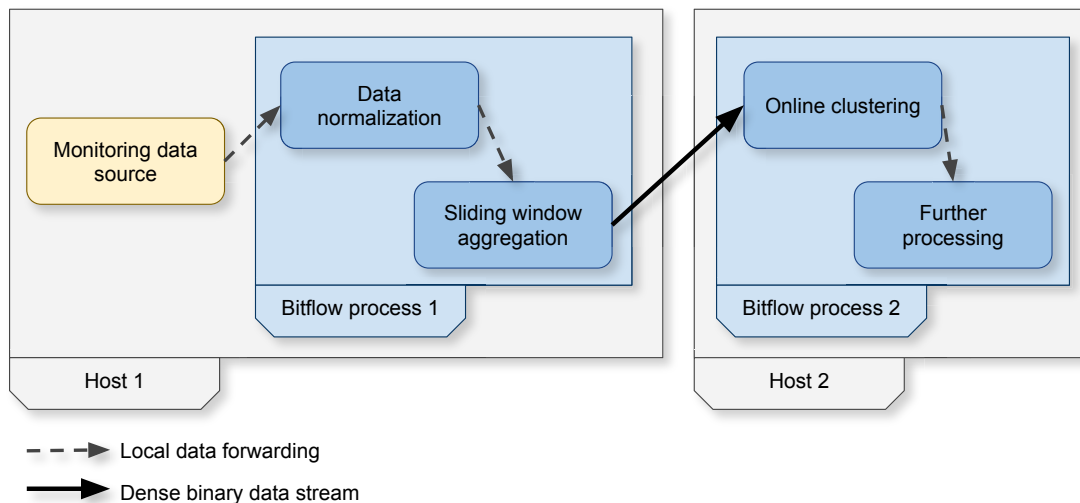


Figure 6.5: A chain of Bitflow Runtime processes analyzing a data source.

6.3.3 Bitflowscript Language

The Bitflow framework defines a light-weight Domain-Specific Language (DSL) for specifying data processing pipelines¹⁶, called *Bitflowscript*. Bitflowscript allows to construct flexible and expressive pipelines of data analysis steps from a toolbox of predefined components. Listing 3 shows a short example for the Bitflowscript DSL used for anomaly detection.

Bitflowscript consists of data sources and data sinks in URL syntax, as well as processing steps in a function-like syntax with parameters. These elements can be combined by the chaining operator `->`. Sub-pipelines in forks are surrounded by curly braces and separated by semicolons. The Bitflowscript compiler parses the script and instantiates every processing step as an actual object in the Bitflow Runtime. Executing a Bitflowscript with unsupported processing steps leads to a

¹⁶<https://bitflow.rtf.d.io/projects/bitflow-antlr-grammars/en/latest/bitflow-script>

```

1 listen://:9000
2 -> fork-tag(tag="component") {
3     * -> detect-anomalies-with-iftm(ensembler=50, opt=1500)
4     -> anomaly-smoothing(windowSize=20, minNumAnomalies=5)
5 }
6 -> write-to-influx-db(
7     databaseURL="http://storage.local:8086",
8     database="detected_anomalies")
9 -> tcp+csv://192.168.100.100

```

Listing 3: Example for the Bitflowscript DSL

well formatted error message, and a list of supported elements can be queried from the containers both in human and machine readable formats. Bitflowscript allows to express data analysis pipelines in a concise, declarative syntax, and facilitates online updates and patches in running pipelines.

6.3.4 Bitflow Data Collector



The Bitflow data collector `bitflow-collector`¹⁷ is ZerOps' main source of monitoring data for physical and virtual machines. The data collector operates in a similar fashion to related approaches for monitoring generic processes and applications [49]: It runs as a single process on a hypervisor machine and supports the collection of time series data from different sources in a modular way. For easier deployment and portability, the tool is also packaged as a Docker container.

The design goal of the `bitflow-collector` is to provide generic and *not* application-specific data about hosts and services. This supports the goal of ZerOps to monitor the infrastructure of an IaaS cloud provider, where multiple tenants share the resources and the provider does not have any access to the workload deployed in the VMs. The frequency of sampling data with `bitflow-collector` can be arbitrarily high and the data collection service is designed to minimize resource overhead induced by the sampling routine itself. The `bitflow-collector` provides all data in the Bitflow sample format, which integrates seamlessly with the Bitflow Controller and the Bitflow Runtime.

The `bitflow-collector` agent uses the three data sources `procfs`, `Libvirt`, and `OVSDb`. Table 6.3 gives an overview over all collected metrics.

¹⁷<https://github.com/bitflow-stream/go-bitflow-collector>

Table 6.3: Metrics provided by the Bitflow data collector.

Resource	Metrics	Data source granularity		
		procfs	Libvirt	OVSDB
CPU	cpu cpu-jiffies	Host/process	VM	—
Load	load/1 load/5 load/15	Host/process	VM	—
Memory	mem/percent mem/used mem/free	Host/process	VM	—
Disk Access	disk-io/byte disk-io/write-byte disk-io/read-byte disk-io/ops disk-io/write-ops disk-io/read-ops disk-io/time disk-io/write-time disk-io/read-time	Host/process	VM	—
Disk Space	disk-space/percent disk-space/used disk-space/free	Host/process	VM	—
Network	net-io/byte net-io/rx-byte net-io/tx-byte net-io/packets net-io/rx-packets net-io/tx-packets net-io/errors net-io/dropped	Host/process	VM	Network link
Other	num-procs num-threads	Host/process	—	—

The local `/proc` filesystem

The Linux kernel exposes information about running processes through the pseudo file system `procfs`, typically mounted at `/proc`¹⁸. The `procfs` provides a variety of static information, but also continuously updated values about resource utilization. The `bitflow-collector` samples these utilization values in a configurable interval to derive time series which can be analyzed for anomaly detection purposes. The main metrics analyzed in ZerOps cover the utilization of CPU and main memory, access to hard disks, and network activity. Additional metrics include counters of processes and threads, and detailed information about specific network protocols. The main drawback of this data source is that it is provided by the Linux kernel in a text format, which must be parsed at a certain performance overhead. By default, `bitflow-collector` parses this data for the entire operating system to ensure low resource overhead induced by the monitoring system. At the cost of additional parsing operations, the same data can be obtained for individual processes.

The Libvirt interface

The virtualization library Libvirt [21] provides unified access to virtual machines running on a hypervisor host. Through a standardized network protocol¹⁹, Libvirt allows to obtain information and data about managed VMs. Besides static meta information, this interface offers similar resource utilization data as `procfs`, namely time series of the utilization of CPU, main memory, disk access and network traffic. This data is available for every VM running on a hypervisor.

There are multiple benefits in obtaining this VM-specific data from Libvirt, instead of querying the `procfs` filesystem within the VM. Firstly, the Libvirt interface is independent of the used virtualization technology, and of the operating system in the virtual machine. Secondly, no access to the internals of the monitored VMs is required in order to obtain this data. Therefore, this approach works with a broad range of use case scenarios, and is a good approximation of a productive public cloud scenario, where the cloud service provider sees the customer VMs as black boxes.

¹⁸<https://www.kernel.org/doc/Documentation/filesystems/proc.txt>

¹⁹<https://libvirt.org/internals/rpc.html>

The OVSDB protocol

Besides resource usage of individual hosts, more detailed data about the network connections between hosts and services is useful when looking for abnormal behavior. Open vSwitch is a virtual switch that often implements the virtual network layer in open source cloud infrastructures. The main feature of Open vSwitch in this context is to establish network tunnels between physical hosts and to create tenant-specific overlay networks for the VMs. Open vSwitch implements the OVSDB protocol²⁰, which provides access to the virtual interfaces and bridges on a hypervisor. While the cloud stack controls the creation, deletion and configuration of the virtual network elements, the `bitflow-collector` uses the OVSDB protocol to query the network infrastructure and utilization of the virtual links. In order to utilize the obtained information, the virtual links must be associated with the VMs they belong to. The required associations are obtained from the cloud operating system.

6.4 Experimental Evaluation

The goal of this experimental evaluation is to measure a number of runtime properties of the ZerOps system. In order to make these measurements more realistic, we generate load on the underlying cloud system by simulating tenants. These tenants execute different types of services on the cloud, which stresses the resources of the hypervisors and feeds the data analysis tasks in the ZerOps system. We complete the tenant simulation by generating load on the services using external clients. Furthermore, we inject artificial anomalies during the experiment, in order to trigger all parts of the ZerOps system, that go beyond anomaly detection.

6.4.1 Tenant Service Scenarios

To facilitate experimentation with the ZerOps platform, ZerOps includes two tenant service scenarios that can be deployed on top of the OpenStack cloud platform. The two scenarios were selected to maximize the diversity of the resulting resource consumption. Both scenarios support arbitrary scale-out configurations, which can be chosen depending on the number and size of the physical hosts. Different service configurations lead to different load levels during an experiment.

Both service scenarios contain easy-to-use deployment artifacts. The necessary cloud resources, such as virtual networks and VMs, are provisioned using Open-

²⁰<https://tools.ietf.org/html/rfc7047>

Stack's Heat project²¹. OpenStack Heat implements a domain-specific language that allows to describe entire topologies of cloud resources. Using a single command, the Heat service rolls out the described topologies in a target OpenStack cloud. After the VMs have booted, ZerOps deploys the actual tenant service using Ansible playbooks²². Ansible is an orchestration tool that connects to the target hosts via SSH and performs a series of commands based on a declarative description of the desired host state. ZerOps' Ansible playbooks are idempotent and can be executed whenever a service must be restarted, repaired, or redeployed.

The deployment artifacts for ZerOps' tenant services are available on Github²³.

Tenant Service: Video Streaming

The first service scenario consists of two types of applications: video streaming servers and load balancers. Both services operate on the Real-Time Messaging Protocol (RTMP)²⁴. The video streaming servers use the Nginx web server²⁵ and its RTMP plugin²⁶ to serve sample video content with different lengths and resolutions. Clients of the video streaming service do not connect to the backend servers directly, but instead start streaming videos over one of the load balancers. When a video is requested, the load balancer randomly selects one of the backend servers, forwards the request, and redirects the video stream to the client. In this scenario, the load balancers do not implement any additional functionality such as caching or reliability mechanisms.

The two classes of servers in the video streaming scenario exhibit different resource usage patterns. The backend service mainly generates outgoing network traffic, and also some hard drive access to load the videos. The load balancers, on the other hand, generate both incoming and outgoing network traffic, and use the disk only for swapping.

The video streaming scenario includes a load generation client²⁷. This load generation tool was specifically designed to facilitate experiments with RTMP and HTTP based streaming services. The client is configured with a list of target HTTP or RTMP URLs. When the experiment is started, the client continuously sends requests to the provided URLs in configurable order and time intervals. To

²¹<https://docs.openstack.org/heat/latest/>

²²https://docs.ansible.com/ansible/latest/user_guide/playbooks.html

²³<https://github.com/citlab/testbed-scenarios>

²⁴<https://www.adobe.com/devnet/rtmp.html>

²⁵<https://docs.nginx.com/>

²⁶<https://github.com/arut/nginx-rtmp-module>

²⁷<https://github.com/antongulenko/stream-statistics-client>

make the experiments measurable, the client provides real-time statistics about the generated data streams. These statistics include transferred bandwidth, open connections and streams, as well as encountered errors and in case of video streams, the number of transferred frames and pixels. This data can be used to calculate service quality metrics that track how well the video servers are operating. For increased flexibility, the load generation client offers an HTTP API that allows to change its configuration during the experiment.

To make experiments as realistic as possible, the load generation clients are typically not executed on the same infrastructure as the services. Executing the clients on external hosts ensures that the clients do not affect the servers' operation, except through their requests over the network. Figure 6.6 shows all components of the video streaming scenario and their distribution on two separate infrastructures.

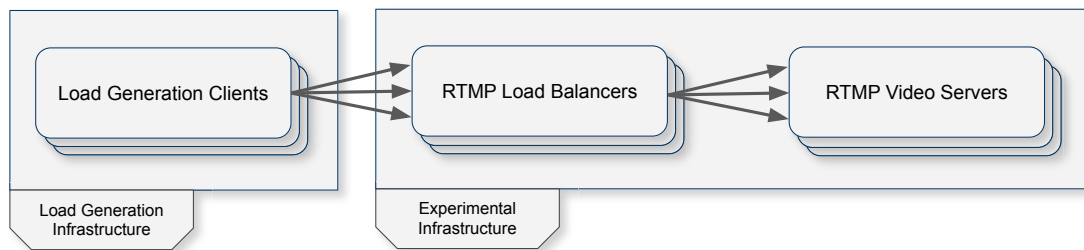


Figure 6.6: Parts of the video streaming tenant service and their separation

Both servers and the load generation client are packaged in Docker containers. The Ansible playbook for the video streaming scenario downloads and starts the respective containers with correct configuration.

Tenant Service: IP Multimedia Subsystem

The IP Multimedia Subsystem (IMS) is a standardized collection of services and protocols that enables various functionalities related to multimedia and IP telephony [1, 28]. One of the core use cases for an IMS system is to establish a communication session between two end users. Users can identify and contact each other using identification tokens such as phone numbers or user names. The IMS service is responsible for resolving these tokens to actual communication endpoints, such as IP addresses. When users roam and change their location, their IMS client registers their updated endpoint with the IMS system.

ZerOps includes IMS as the second service scenario, because the communication pattern of IMS is quite different from the video streaming scenario. IMS is not responsible for actually routing long-running connections, such as VOIP sessions or video calls. Instead of long running streaming connections, IMS traffic consists of short, bursty message cascades.

Project Clearwater²⁸ is an open source implementation of the IMS specification. Clearwater consists of over ten micro services that cooperate for delivering the IMS core functionality. The services of Clearwater are designed to be "cloud-native" – i.e., they are mostly stateless, resilient, and individually scalable. The internal communication of Clearwater consists mostly of UDP-based Session Initiation Protocol (SIP) messages, with some exceptions using HTTP.

In order to generate actual load in the tenant services, ZerOps simulates clients that continuously register with the system and establish IMS sessions. The IMS load generation client software, Sipp, is available online²⁹. Sipp performs various sequences of IMS requests, including operations such as registration, calling another user, answering a call, ending the session, and so on. By starting multiple instances with different configurations, Sipp can emulate a variety of end user behaviors. Similarly to the video streaming scenario, ZerOps runs the Sipp clients on dedicated physical hosts to minimize their impact on the actual cloud testbed.

6.4.2 Anomaly Injection

The anomaly injector component of ZerOps allows to perform controlled and reproducible experiments with real cloud workloads. The software is available on Github³⁰. Since anomalies are highly irregular in fully functional software, the ZerOps anomaly injector implements generic artificial resource anomalies that can be injected and reverted in a controlled fashion. The implemented anomalies follow two design goals:

1. anomalies are applicable to arbitrary injected services, and
2. anomalies can be reverted to bring the system back into a healthy state.

The ZerOps anomaly injector consists of two components, visualized in Figure 6.7. Every host that is part of the experiment, executes an *anomaly injection agent*. These agents implement an HTTP-based API that allows to remotely inject anomalies, revert them, and query their current status. Anomaly injection agents can

²⁸<https://clearwater.readthedocs.io/en/stable/>

²⁹<https://sourceforge.net/projects/sipp/>

³⁰<https://github.com/citlab/distributed-anomaly-injection>

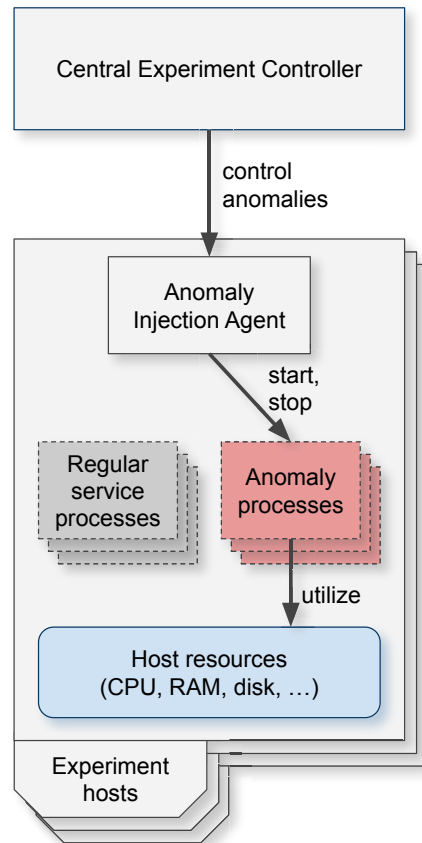


Figure 6.7: The ZerOps anomaly injector: distributed anomaly agent and central experiment controller

be started with a predefined time plan that defines when the agent has to inject and revert anomalies. However, this is not the preferred way of running anomaly injection experiments. In case an anomaly cannot be reverted due to a component failure, the distributed agents would continue to inject anomalies and potentially break the experiments, or render the testbed unusable. Therefore, the ZerOps anomaly injector contains a centralized experiment controller. When an experiment is started, the controller receives a plan for the anomalies to be injected. This plan contains the timing, intensity, parameterization, and targets of all planned anomalies. Every aspect of the plan can be randomized in order to increase the diversity of the experiment. Whenever the experiment controller proceeds with the plan, it checks the current status of the testbed by querying all anomaly injection agents. If any agent is not reachable, or reports a problem, the experiment is suspended, until the problem is resolved. The controller maintains a log of

every event throughout the entire experiment, to make long-running experiments comprehensible.

All anomalies implemented in the anomaly injection agent are idempotent. This means an anomaly will not be injected multiple times, even if the agent or host fail or are restarted. Most anomalies work by launching a process that consumes system resources, but some anomalies change certain configurations of the operating system. When configured correctly, these “anomaly processes” disturb the regular workload of the target host. ZerOps operates under the assumption, that tenants of the cloud platform execute one service per VM. Therefore, the anomaly injection processes appear to be part of the tenant service, when observing the VM from the outside.

Table 6.4: Building blocks for resource consumption anomalies.

Consumption Patterns	
Hogging	Allocate a configured amount of the resource without releasing it
Leakage	Allocate a configured amount of the resource in regular time intervals, until reaching a maximum value
Fluctuation	Same as leakage, but upon reaching the maximum value, start releasing the resource, and restart after releasing everything
Resources	
CPU	Perform mathematical operations to produce the desired CPU load
RAM	Allocate the defined amount of memory
Disk access	Repeatedly perform read or write operations to a disk
Disk space	Create a file of the desired size
Network	Send arbitrary network traffic from and to a remote host
PIDs	Create the desired number of child processes
File pointers	Open a number of files without releasing the file pointers

New anomaly implementations can be plugged into the anomaly injection agent by providing a script that responds correctly to standardized command line parameters. However, the agent supports a number of useful anomalies out of the box. Table 6.4 summarizes all anomalies that consume system resources in various

ways. The upper part of the table shows the three supported resource consumption patterns. These patterns can be combined with each of the system resources shown below. These building blocks allow to generate a big spectrum of anomalous behavior. For example, *leakage of file pointers* emulates a software bug where file pointers are not released and eventually cause new file operations to fail. *Hogging of RAM* triggers high swapping activity or causes out-of-memory errors in other processes. *Fluctuation of PIDs* emulates a repeated fork bomb attack that is intended to render the host unresponsive. More conservative anomalies, such as memory leaks or CPU overutilization, are supported as well.

In addition to the resource consumption anomalies, the ZerOps anomaly injection agent supports a number of specialized *network configuration anomalies*. Table 6.5 summarizes these anomalies. The implementation of these anomalies makes use of the Traffic Control³¹ feature of the Linux kernel. Network configuration anomalies allow to emulate anomalous behavior of network devices, such as overload on a switch or a router, which leads to packets being dropped.

Table 6.5: Network configuration anomalies.

Latency	Increase the processing time for packets on all external network interfaces by a defined value
Bandwidth	Artificially reduce the bandwidth on all external network interfaces to a defined value
Packet loss	Randomly drop a part of all incoming and outgoing packets on all external interfaces

6.4.3 Testbed and Experimental Procedure

In this evaluation, a cluster of dedicated physical commodity servers functions as physical infrastructure for the experimental ZerOps testbed. Table 6.6 summarizes the core properties of our testbed. Both OpenStack and Kubernetes are installed on the entire range of compute nodes, alongside all other ZerOps components. The two tenant service scenarios video streaming and IMS are deployed in OpenStack for load generation. A small number of external dedicated nodes are reserved for the load generation clients for the two scenarios.

³¹<https://lartc.org/>

Table 6.6: Properties of the ZerOps experimental testbed.

Physical CPU frequency	3.30 GHz
Number of cores per node	8
Main memory per node	16 GB
Network links between nodes	1 Gbit/s
OpenStack/Kubernetes compute nodes	12
Ceph storage nodes	3
Load generation client nodes	4
Video streaming VMs	12
IMS VMs	33

Experimental Procedure

For a period of 72 hours, we record various types of data in the ZerOps testbed. To avoid overly monotonous resource consumption, both classes of clients are configured to vary the level of load in regular intervals. A central coordinator randomly selects a new load level in intervals of ten minutes. For the video streaming service, the clients vary the number of videos streamed in parallel. The IMS clients, on the other hand, adjust the number of end users simulated at a time.

Table 6.7: Artificial anomalies injected during the experiment, with their respective intensities.

Anomaly	Video Streaming	IMS	Hypervisors
CPU hogging	90–100%	90–100%	90–100%
Disk hogging	Full utilization	Full utilization	Full utilization
Network hogging	Full utilization	Full utilization	Full utilization
Memory hogging	8000–1000 MB	400–500 MB	1500–2500 MB
Memory leak	90–240 MB/min	60–180 MB/min	90–240 MB/min
Reduced bandwidth	50–60 Kb/s	50–60 Kb/s	15–25 Mb/s

During the experiment, we use the ZerOps anomaly injection framework to create disturbances in the cloud workload. Table 6.7 summarizes the anomalies injected for the evaluation experiments, with their respective intensities. We manually tuned each intensity to ensure that the anomalies affect the quality of the services, without rendering them unresponsive, or even crashing them. The intensity of

every anomaly injection is randomly selected within the defined range. Every anomaly is injected for a duration of five minutes, followed by a cool down period of ten minutes. The controller randomly selects a combination of an anomaly and a target host, while making sure that no combination is repeated more than once. After injecting every anomaly on every target host, the experiment controller repeats the entire cycle.

Table 6.8 summarizes the main configuration parameters for the procedure of this experiment.

Table 6.8: Configurations of the experimental evaluation procedure.

Total duration	72 hours
Client load change interval	10 minutes
Anomaly injection interval	15 minutes
Anomaly injection duration	10 minutes
Cooldown period between anomalies	5 minutes

With two types of measurements, we perform a numerical evaluation of the ZerOps platform. The first measurement covers the resource consumption of all elements of the ZerOps platform. The objective of this measurement is to show that the zero-touch operations system induces a resource overhead that is low enough to be used in practice. One of the requirements defined in Chapter 1 is “Unintrusiveness”: The zero-touch operations platform must have a very low impact on the remaining cloud platform and its tenant workload. ZerOps can only fulfill this requirement with a low resource overhead. Therefore, the second type of measurements concerns the in situ data analysis engine and evaluates the volume of network traffic that the platform saves by analyzing the data streams directly at their origin. In this regard, we measure how much data the ZerOps data collector produces per monitored host.

6.4.4 Evaluation Results and Discussion

Requirement: Scalability

Scalability of the zero-touch operations platform means that its design naturally adapts to arbitrary sizes of the monitored cloud platform, and to arbitrary numbers of monitored hosts. This requirement mainly concerns the data analysis part of ZerOps. Scalability cannot be achieved by having a single, centralized entity

perform the data analysis for a potentially unbounded number of hosts. Instead, ZerOps spreads all data analysis tasks over the entire cloud platform. We achieve this by two means: in situ data analysis and hierarchical data analysis. All data processing tasks that do not require the context of multiple monitored hosts, are executed directly where the respective data stream originates. Wherever the data of multiple hosts is required (e.g., for RCA), we rely on a divide-and-conquer approach by splitting the analysis tasks in multiple layers, that each encompass a limited number of monitored hosts.

Requirement: Extensibility

One of our core design goals for the ZerOps and the underlying zero-touch operations architecture is to make it modular and extensible. Extensibility is first addressed on the overall architectural level. All components, including the data collection, the various data analysis tasks, and the selection and execution of remediation workflows, are decoupled from each other through clearly defined interfaces. This makes it possible to implement new data sources, design different data analysis pipelines, or define new remediation workflows, without rethinking the entire system architecture.

Another modular and extensible part of our architecture is the data analysis pipeline, as defined in Section 4.3. Every data analysis task is a standalone module with clear input and output data formats. A number of specific algorithms can be chosen to implement each part of the pipeline.

Requirement: Unintrusiveness

The third requirement, unintrusiveness, demands that ZerOps works as a seamless extension of a traditional cloud platform, without disrupting its main operation. ZerOps has multiple mechanisms to minimize a negative impact on the underlying cloud platform. One important mechanism is the capability of the in situ data analysis engine to limit the resources used by the data analysis processes. Administrators define upper resource consumption limits that they deem acceptable. However, some elements of the zero-touch operations architecture are not covered by these hard limits. One such element is the data collector. In order to show, that the consumed resources by the data collector and other parts of ZerOps are within an acceptable range, we have monitored these resources during the experiments described in the previous section.

Table 6.9 summarizes the average CPU utilization for different elements of ZerOps

for the entire duration of the experiment. The first four lines show the resource consumption for individual components of ZerOps: per process and summed up for the entire testbed. The line entitled "Hypervisors" shows the overall CPU utilization measured on the hypervisors, mainly representing the tenant workload. The last three lines summarize the overall resource overhead of ZerOps, and compare it to the entire cloud platform. The results show that ZerOps uses 16.69% of the CPU of the cloud platform's total utilization, and 10.69% of all available CPU resources.

Table 6.9: Average CPU consumption of ZerOps and OpenStack.

Component type	Average CPU consumption per component	Number of components	Total CPU consumption
Data collector	$1.78\% \pm 0.543\%$	12	21.36%
In situ analysis controller	$1.61\% \pm 0.548\%$	1	1.61%
Data preprocessing	$0.47\% \pm 0.167\%$	45	21.15%
Anomaly detection	$1.47\% \pm 1.87\%$	45	84.15%
Hypervisor	$64.05\% \pm 15.08\%$	12	768.60%
Hypervisor total resources	100%	12	1200%
ZerOps combined			128.27%
ZerOps relative to hypervisors			16.69%
ZerOps relative to total resources			10.69%

Table 6.10 shows the average memory consumption of the ZerOps components and the underlying cloud infrastructure. The structure of the table follows that of Table 6.9, but the values are given in MB and GB. The results show that ZerOps utilizes roughly 2% of the overall memory resources and 3% of the memory utilized by the cloud platform. These values indicate that ZerOps is less memory-intensive than it is CPU intensive.

In Situ Data Analysis

One of the goals of the in situ data analysis engine is to reduce the network bandwidth occupied by real-time monitoring data. This section presents measurements in the ZerOps experimental testbed, that quantify these savings. First we calculate how much of the real-time data streams we eliminate by analyzing the data directly at the source, then we show the volume of these data streams.

Table 6.10: Average memory consumption of ZerOps and OpenStack.

Component type	Average memory consumption per component	Number of components	Total memory consumption
Data collector	82.96 MB \pm 16.70 MB	12	995.52 MB
In situ analysis controller	61.72 MB \pm 0.02 MB	1	61.72 MB
Data preprocessing	53.92 MB \pm 1.08 MB	45	2426.40 MB
Anomaly detection	67.59 MB \pm 7.29 MB	45	811.08 MB
Hypervisor	9.5 GB \pm 1.9 GB	12	113.5 GB
Hypervisor total resources	16.8 GB	12	201.6 GB
ZerOps combined			4294.72 MB
ZerOps relative to hypervisors			3.78%
ZerOps relative to total resources			2.13%

The ZerOps data collector records data with a frequency of two Hertz. Typical monitoring systems that record resource utilization data for long-term storage store the data at much lower frequencies. Such monitoring systems aggregate the high-frequency data, which greatly reduces its size, but also sacrifices a lot of information. Assuming a data storage frequency of 5 minutes, 600 high-frequency samples are aggregated into one sample that is actually sent over the network. This reduces the volume of the data stream by 99.83%.

Table 6.11 shows the measured respective data stream volumes produced by the Bitflow Data Collector. The resulting numbers are rather small in the context modern gigabit networks. The bandwidth savings produced by the in situ data analysis engine become significant in bigger data centers and cloud platforms of thousands of hypervisors and tens of thousands of virtual machines. Additional data sources further increase the data volume, and with it the bandwidth saved by in situ analysis. For small deployments, such as the ZerOps experimental testbed, the in situ data analysis offers the benefit of a quicker data analysis results.

Table 6.11: Bandwidth occupied by real-time monitoring data.

Component type	Data stream
Hypervisor	3.29 kB/s
VM	1.32 kB/s
Testbed total	98,88 kB/s

Chapter 7

Conclusion

This thesis presents an architectural and algorithmic approach to a self-healing cloud platform. The first step is to identify what data sources can be used to monitor the behavior of critical components, and how the topology of the system can automatically be inferred. These data streams are then processed in a scalable and efficient way by an in situ data analysis engine. The data analysis pipeline includes steps for anomaly detection, anomaly classification, and the selection of remediation workflows. Finally, our system closes the loop by automatically executing the optimal remediation workflow for each encountered anomaly situation.

The contributions of this thesis are threefold. First, we designed an autonomic, self-healing extension to traditional cloud computing platforms. The resulting zero-touch operations architecture is a blueprint for different cloud systems. Our architecture covers all aspects, from component monitoring and topology inference, to data analysis and automatic remediation. Second, we presented the design of an in situ data analysis engine that fulfills the requirements of our zero-touch operations architecture. Our engine focuses on flexibility and extensibility, while also limiting the resources reserved for data analysis in order to not interfere with the actual cloud workload. And finally, we applied our self-healing cloud architecture to the use cases of a public IaaS cloud platform. We implemented an experimental testbed based on our self-healing cloud platform ZerOps, and performed experimental measurements for different parts of the platform. In a qualitative and quantitative evaluation, we showed that ZerOps fulfills the requirements that we initially defined for a self-healing cloud system.

The zero-touch operations architecture follows a *data-driven* approach, and as any data-driven system, it has certain limitations. The most important precondition

for detecting and analyzing anomalies in monitoring data is that the anomalous behavior of the system actually manifests in observable metrics. If an anomaly does not visibly manifest itself, or if the data analysis does not have access to the according metrics, no detection and automatic remediation is possible. Furthermore, it is inherently impossible to guarantee that a data-driven automation system always makes the same decisions as a human would in the same situation. A small percentage of wrong decisions is to be expected, at least with the currently available data analysis techniques. The managed application has to be able to tolerate that. The overall success of the zero-touch operations architecture is entirely dependent on the utilized data analysis techniques and available remediation workflows.

Despite the holistic design of our architecture, we imagine several future research directions to extend its autonomic self-healing capabilities. An interesting extension would be *meta self-healing*: enabling the autonomic system to recursively monitor, analyze, and heal itself. A unification of the different autonomic managers in today's cloud systems would further increase their capabilities. This means, for example, combining the self-healing platform with IDSs for self-protection. In order to integrate with larger cloud infrastructures, the in situ data analysis engine has to optimize its scheduling algorithm for complex network topologies. On the algorithmic side, an interesting extension would be anomaly *prediction* for even faster reaction times. The level of automation could be increased even beyond the current level by implementing automatic *composition* of remediation workflows. Instead of selecting from a list of predefined alternatives, the system could assemble the entire workflow from a large selection of fine-grained steps. This could lead to remediation strategies that were previously unthought of by human administrators.

Even without the named suggestions, our architecture is capable to autonomically detect and resolve anomalies in a cloud infrastructure, and to continuously improve its accuracy by building a knowledge base. We believe that autonomic, data-driven closed-loop systems will further expand, and eventually become the default approach for operating large computer infrastructures.

Bibliography

- [1] 3GPP. *IP Multimedia Subsystem (IMS) Stage 2*. 2019. URL: <https://www.3gpp.org/DynaReport/23228.htm> (visited on 05/03/2020).
- [2] Sherif Abdelwahab, Bechir Hamdaoui, Mohsen Guizani, and Taieb Znati. “Network function virtualization in 5G”. In: *IEEE Communications Magazine*. Vol. 54. 4. IEEE. 2016, pp. 84–91.
- [3] Giuseppe Aceto, Alessio Botta, Walter De Donato, and Antonio Pescapè. “Cloud monitoring: A survey”. In: *Computer Networks*. 9. Elsevier. 2013, pp. 2093–2115.
- [4] Alexander Acker, Florian Schmidt, Anton Gulenko, and Odej Kao. “Online Density Grid Pattern Analysis to Classify Anomalies in Cloud and NFV Systems”. In: *International Conference on Cloud Computing Technology and Science (Cloud-Com)*. IEEE. 2018, pp. 290–295.
- [5] Partnership for Advanced Computing in Europe (PRACE) and Norbert Meyer. *Data Centre Infrastructure Monitoring*. 2016. URL: <http://www.prace-ri.eu/IMG/pdf/WP226.pdf> (visited on 09/18/2019).
- [6] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. “The stratosphere platform for big data analytics”. In: *The VLDB Journal—The International Journal on Very Large Data Bases*. Vol. 23. 6. 2014, pp. 939–964.
- [7] NGMN Alliance. *5G white paper*. 2015. URL: https://www.ngmn.org/wp-content/uploads/NGMN_5G_White_Paper_V1_0_01.pdf (visited on 09/18/2019).
- [8] Amazon Web Services, Inc. *Amazon CloudWatch Developer Guide*. URL: <http://awsdocs.s3.amazonaws.com/AmazonCloudWatch/latest/acw-dg.pdf> (visited on 10/10/2019).
- [9] Martin Andenmatten. “AIOps—Artificial Intelligence für IT-Operations”. In: *HMD Praxis der Wirtschaftsinformatik*. Vol. 56. 2. Springer. 2019, pp. 332–344.

- [10] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. “Adaptive online scheduling in storm”. In: *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM. 2013, pp. 207–218.
- [11] Danilo Ardagna, Gabriele Giunta, Nunzio Ingraffia, Raffaella Mirandola, and Barbara Pernici. “QoS-driven web services selection in autonomic grid environments”. In: *OTM Confederated International Conferences, "On the Move to Meaningful Internet Systems"*. Springer. 2006, pp. 1273–1289.
- [12] Paul Armer. *Social Implications of the Computer Utility*. Rand Corporation, 1967.
- [13] AT&T. *AT&T Domain 2.0 Vision White Paper*. 2013. URL: https://www.att.com/Common/about_us/pdf/AT%20Domain%202.0%20Vision%20White%20Paper.pdf (visited on 10/06/2015).
- [14] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. “Basic concepts and taxonomy of dependable and secure computing”. In: *IEEE transactions on dependable and secure computing*. IEEE. 2004, pp. 11–33.
- [15] Anju Bala and Inderveer Chana. “Fault tolerance - challenges, techniques and implementation in cloud computing”. In: *International Journal of Computer Science Issues*. Vol. 9. 1. IJSCI. 2012.
- [16] Alejandro Baldominos, Esperanza Albacete, Yago Saez, and Pedro Isasi. “A scalable machine learning online service for big data real-time analysis”. In: *IEEE Symposium on Computational Intelligence in Big Data (CIBD)*. IEEE. 2014, pp. 1–8.
- [17] Wolfgang Barth. *Nagios: System and network monitoring*. No Starch Press, 2008.
- [18] Alexander Behm, Vinayak R Borkar, Michael J Carey, Raman Grover, Chen Li, Nicola Onose, Rares Vernica, Alin Deutsch, Yannis Papakonstantinou, and Vassilis J Tsotras. “Asterix: towards a scalable, semistructured data platform for evolving-world models”. In: *Distributed and Parallel Databases*. Vol. 29. 3. Springer. 2011, pp. 185–216.
- [19] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. “Pearson correlation coefficient”. In: *Noise reduction in speech processing*. Springer. 2009, pp. 1–4.
- [20] Gordon S Blair, Geoff Coulson, Lynne Blair, Hector Duran-Limon, Paul Grace, Rui Moreira, and Nikos Parlavantzas. “Reflection, self-awareness and self-healing in OpenORB”. In: *Proceedings of the first workshop on Self-healing systems*. ACM. 2002, pp. 9–14.

- [21] Matthias Bolte, Michael Sievers, Georg Birkenheuer, Oliver Niehörster, and André Brinkmann. “Non-intrusive virtualization management using libvirt”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2010, pp. 574–579.
- [22] Nicolas Bonvin, Thanasis G Papaioannou, and Karl Aberer. “A self-organized, fault-tolerant and scalable replication scheme for cloud storage”. In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, pp. 205–216.
- [23] Kevin D Bowers, Ari Juels, and Alina Oprea. “HAIL: A high-availability and integrity layer for cloud storage”. In: *Proceedings of the 16th ACM conference on Computer and communications security*. ACM. 2009, pp. 187–198.
- [24] David Breitgand, Ealan Henis, and Onn Shehory. “Automated and adaptive threshold setting: Enabling technology for autonomy and self-management”. In: *Second International Conference on Autonomic Computing (ICAC)*. IEEE. 2005, pp. 204–215.
- [25] Eric A Brewer. “Lessons from giant-scale services”. In: *IEEE Internet computing*. Vol. 5. 4. IEEE. 2001, pp. 46–55.
- [26] Mark Burgess. “Computer Immunology”. In: *LISA*. Vol. 98. 1998, pp. 283–298.
- [27] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. “Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility”. In: *Future Generation computer systems*. Vol. 25. 6. Elsevier. 2009, pp. 599–616.
- [28] Gonzalo Camarillo and Miguel-Angel Garcia-Martin. *The 3G IP multimedia subsystem (IMS): merging the Internet and the cellular worlds*. John Wiley & Sons, 2007.
- [29] George Candea and Armando Fox. “Designing for high availability and measurability”. In: *Proceedings of the 1st Workshop on Evaluating and Architecting System Dependability*. 2001.
- [30] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Stefano Iannucci, Francesco Lo Presti, and Raffaella Mirandola. “Moses: A framework for qos driven runtime adaptation of service-oriented systems”. In: *IEEE Transactions on Software Engineering*. Vol. 38. 5. IEEE. 2011, pp. 1138–1159.
- [31] Giuseppe Carella, Marius Corici, Paolo Crosta, Paolo Comi, Thomas Michael Bohnert, Andreea Ancuta Corici, Dragos Vingarzan, and Thomas Magedanz. “Cloudified IP multimedia subsystem (IMS) for network function virtualization (NFV)-based architectures”. In: *IEEE Symposium on Computers and Communications (ISCC)*. IEEE. 2014, pp. 1–6.

- [32] Miguel Castro, Barbara Liskov, et al. “Practical Byzantine fault tolerance”. In: *OSDI*. 1999, pp. 173–186.
- [33] Zenon Chaczko, Venkatesh Mahadevan, Shahrzad Aslanzadeh, and Christopher Mcdermid. “Availability and load balancing in cloud computing”. In: *International Conference on Computer and Software Modeling, Singapore*. Vol. 14. 2011.
- [34] Simon Chan, Thomas Stone, Kit Pang Szeto, and Ka Hou Chan. “PredictionIO: a distributed machine learning server for practical software development”. In: *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. ACM. 2013, pp. 2493–2496.
- [35] Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley Schmerl, and Peter Steenkiste. “An architecture for coordinating multiple self-management systems”. In: *Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE. 2004, pp. 243–252.
- [36] Cheng-Tao Chu, Sang K Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Kunle Olukotun, and Andrew Y Ng. “Map-reduce for machine learning on multicore”. In: *Advances in neural information processing systems*. 2007, pp. 281–288.
- [37] Bo-Yu Chu, Chia-Hua Ho, Cheng-Hao Tsai, Chieh-Yen Lin, and Chih-Jen Lin. “Warm start for parameter selection of linear classifiers”. In: *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2015, pp. 149–158.
- [38] Stuart Clayman, Alex Galis, Clovis Chapman, Giovanni Toffetti, Luis Roderomero, Luis Miguel Vaquero, Kenneth Nagin, and Benny Rochwerger. “Monitoring service clouds in the future internet.” In: *Future Internet Assembly*. Valencia, Spain. 2010, pp. 115–126.
- [39] Igor Costa, Jean Araujo, Jamilson Dantas, Eliomar Campos, Francisco Airtton Silva, and Paulo Maciel. “Availability Evaluation and Sensitivity Analysis of a Mobile Backend-as-a-service Platform”. In: *Quality and Reliability Engineering International*. Vol. 32. 7. Wiley Online Library. 2016, pp. 2191–2205.
- [40] Domenico Cotroneo, Roberto Natella, and Stefano Rosiello. “A fault correlation approach to detect performance anomalies in virtual network function chains”. In: *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2017, pp. 90–100.
- [41] Carlo Curino, Evan PC Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Sam Madden, Hari Balakrishnan, and Nickolai Zeldovich. “Relational Cloud: A Database-as-a-Service for the Cloud”. In: *5th Biennial Conference on Innovative Data Systems Research, CIDR*. 2011, pp. 9–12.

- [42] Yuan-Shun Dai, Bo Yang, Jack Dongarra, and Gewei Zhang. “Cloud service reliability: Modeling and analysis”. In: *Pacific Rim International Symposium on Dependable Computing*. Citeseer. 2009, pp. 1–17.
- [43] Yingnong Dang, Qingwei Lin, and Peng Huang. “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.
- [44] Shirlei Aparecida De Chaves, Rafael Brundo Uriarte, and Carlos Becker Westphall. “Toward an architecture for monitoring private clouds”. In: *IEEE Communications Magazine*. Vol. 49. 12. IEEE. 2011, pp. 130–137.
- [45] Daniel Joseph Dean, Hiep Nguyen, and Xiaohui Gu. “Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems”. In: *Proceedings of the 9th international conference on Autonomic computing*. ACM. 2012, pp. 191–200.
- [46] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM*. Vol. 51. 1. ACM. 2008, pp. 107–113.
- [47] Juan Deng, Hongxin Hu, Hongda Li, Zhizhong Pan, Kuang-Ching Wang, Gail-Joon Ahn, Jun Bi, and Younghye Park. “VNGuard: An NFV/SDN combination framework for provisioning and managing virtual firewalls”. In: *Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. IEEE. 2015, pp. 107–114.
- [48] Brian Dougherty, Jules White, and Douglas C Schmidt. “Model-driven auto-scaling of green cloud computing infrastructure”. In: *Future Generation Computer Systems*. Vol. 28. 2. Elsevier. 2012, pp. 371–378.
- [49] Vincent C Emeakaroha, Tiago C Ferreto, Marco AS Netto, Ivona Brandic, and Cesar AF De Rose. “Casvid: Application level monitoring for sla violation detection in clouds”. In: *36th Annual Computer Software and Applications Conference*. IEEE. 2012, pp. 499–508.
- [50] Vincenzo Eramo, Emanuele Miucci, Mostafa Ammar, and Francesco Giacinto Lavacca. “An approach for service function chain routing and virtual function network instance migration in network function virtualization architectures”. In: *IEEE/ACM Transactions on Networking*. Vol. 25. 4. IEEE. 2017, pp. 2008–2025.
- [51] Mehmet Ersue. “ETSI NFV management and orchestration - An overview”. In: *Proc. of 88th IETF meeting*. 2013.

- [52] Kaniz Fatema, Vincent C Emeakaroha, Philip D Healy, John P Morrison, and Theo Lynn. “A survey of Cloud monitoring tools: Taxonomy, capabilities and objectives”. In: *Journal of Parallel and Distributed Computing*. Vol. 74. 10. Elsevier. 2014, pp. 2918–2933.
- [53] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. “Auto-sklearn: Efficient and Robust Automated Machine Learning”. In: *Automated Machine Learning*. Springer. 2019, pp. 113–134.
- [54] Stephanie Forrest and Catherine Beauchemin. “Computer immunology”. In: *Immunological reviews*. Vol. 216. 1. Wiley Online Library. 2007, pp. 176–197.
- [55] Stephanie Forrest, Steven A Hofmeyr, and Anil Somayaji. “Computer immunology”. In: *Communications of the ACM*. Vol. 40. 10. Citeseer. 1997, pp. 88–96.
- [56] Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andrew Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, and Ion Stoica. “Above the clouds: A berkeley view of cloud computing”. In: *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*. Vol. 28. 13. 2009.
- [57] Moshe Gabel, Assaf Schuster, Ran-Gilad Bachrach, and Nikolaj Bjørner. “Latent fault detection in large scale services”. In: *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2012, pp. 1–12.
- [58] Amir Gandomi and Murtaza Haider. “Beyond the hype: Big data concepts, methods, and analytics”. In: *International Journal of Information Management*. Vol. 35. 2015, pp. 137–144.
- [59] Alan G Ganek and Thomas A Corbi. “The dawning of the autonomic computing era”. In: *IBM systems Journal*. Vol. 42. 1. IBM. 2003, pp. 5–18.
- [60] Sachin Garg, Aad Van Moorsel, Kalyanaraman Vaidyanathan, and Kishor S Trivedi. “A methodology for detection and estimation of software aging”. In: *Ninth International Symposium on Software Reliability Engineering*. IEEE. 1998, pp. 283–292.
- [61] David Garlan, S-W Cheng, A-C Huang, Bradley Schmerl, and Peter Steenkiste. “Rainbow: Architecture-based self-adaptation with reusable infrastructure”. In: *Computer*. Vol. 37. 10. IEEE. 2004, pp. 46–54.
- [62] Gartner, Inc. *AI Ops Platforms*. 2017. URL: <https://blogs.gartner.com/andrew-lerner/2017/08/09/aiops-platforms/> (visited on 04/10/2019).
- [63] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. “Understanding network failures in data centers: measurement, analysis, and implications”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 41. 4. ACM. 2011, pp. 350–361.

- [64] Michael Glaß, Martin Lukasiewicz, Felix Reimann, Christian Haubelt, and Jürgen Teich. “Symbolic reliability analysis of self-healing networked embedded systems”. In: *International Conference on Computer Safety, Reliability, and Security*. Springer. 2008, pp. 139–152.
- [65] Chon-Huat Goh, Yung-Chin Alex Tung, and Chun-Hung Cheng. “A revised weighted sum decision model for robot selection”. In: *Computers & Industrial Engineering*. Vol. 30. 2. Elsevier. 1996, pp. 193–199.
- [66] Javier Gonz, Antonio Munoz, Antonio Mana, et al. “Multi-layer monitoring for cloud computing”. In: *13th International Symposium on High-Assurance Systems Engineering*. IEEE. 2011, pp. 291–298.
- [67] Michael Grottke, Rivalino Matias, and Kishor S Trivedi. “The fundamentals of software aging”. In: *International Conference on Software Reliability Engineering Workshops (ISSRE Wksp)*. IEEE. 2008, pp. 1–6.
- [68] Anton Gulenko, Florian Schmidt, Alexander Acker, Marcel Wallschläger, Odej Kao, and Feng Liu. “Detecting Anomalous Behavior of Black-Box Services Modeled with Distance-Based Online Clustering”. In: *International Conference on Cloud Computing (CLOUD)*. IEEE. 2018, pp. 912–915.
- [69] Anton Gulenko, Marcel Wallschläger, and Odej Kao. “A Practical Implementation of In-Band Network Telemetry in Open vSwitch”. In: *International Conference on Cloud Networking (CloudNet)*. IEEE. 2018.
- [70] Anton Gulenko, Marcel Wallschläger, Florian Schmidt, Odej Kao, and Feng Liu. “A System Architecture for Real-Time Anomaly Detection in Large-Scale NFV Systems”. In: *Procedia Computer Science*. Vol. 94. Elsevier. 2016, pp. 491–496.
- [71] Anton Gulenko, Marcel Wallschläger, Florian Schmidt, Odej Kao, and Feng Liu. “Evaluating Machine Learning Algorithms for Anomaly Detection in Clouds”. In: *International Conference on Big Data*. IEEE. 2016, pp. 2716–2721.
- [72] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. “Why does the cloud stop computing? Lessons from hundreds of service outages”. In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM. 2016, pp. 1–16.
- [73] Hakan Hacigumus, Bala Iyer, and Sharad Mehrotra. “Providing database as a service”. In: *Proceedings 18th International Conference on Data Engineering*. IEEE. 2002, pp. 29–38.
- [74] Hamed Haddadi, Miguel Rio, Gianluca Iannaccone, Andrew Moore, and Richard Mortier. “Network topologies: inference, modeling, and generation”. In: *IEEE Communications Surveys & Tutorials*. Vol. 10. 2. IEEE. 2008, pp. 48–69.

- [75] Hakan Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. “Executing SQL over encrypted data in the database-service-provider model”. In: *ACM SIGMOD international conference on Management of data*. ACM. 2002, pp. 216–227.
- [76] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. “Network function virtualization: Challenges and opportunities for innovations”. In: *IEEE Communications Magazine*. Vol. 53. 2. IEEE. 2015, pp. 90–97.
- [77] Florian Helff, Le Gruenwald, and Laurent d’Orazio. “Weighted Sum Model for Multi-Objective Query Optimization for Mobile-Cloud Database Environments.” In: *EDBT/ICDT Workshops*. 2016.
- [78] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. “MINIX 3: A highly reliable, self-repairing operating system”. In: *ACM SIGOPS Operating Systems Review*. Vol. 40. 3. ACM. 2006, pp. 80–89.
- [79] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation*. Vol. 9. 8. MIT Press. 1997, pp. 1735–1780.
- [80] Christina N Hoefer and Georgios Karagiannis. “Taxonomy of cloud computing services”. In: *Globecom Workshops*. IEEE. 2010, pp. 1345–1350.
- [81] Paul Horn. “Autonomic computing: IBM’s Perspective on the State of Information Technology”. In: IBM. 2001.
- [82] Markus C Huebscher and Julie A McCann. “A survey of autonomic computing – degrees, models, and applications”. In: *ACM Computing Surveys (CSUR)*. Vol. 40. 3. ACM. 2008.
- [83] IBM. “An architectural blueprint for autonomic computing”. In: *IBM White Paper*. Vol. 31. 2006. Citeseer. 2006, pp. 1–6.
- [84] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. “Directed diffusion: A scalable and robust communication paradigm for sensor networks”. In: *Proceedings of the 6th annual international conference on Mobile computing and networking*. ACM. 2000, pp. 56–67.
- [85] Rebecca Isaacs and Paul Barham. “Performance analysis in loosely-coupled distributed systems”. In: *7th CaberNet Radicals Workshop*. Citeseer. 2002.
- [86] Xing Jin, W-P Ken Yiu, S-H Gary Chan, and Yajun Wang. “Network topology inference based on end-to-end measurements”. In: *IEEE Journal on Selected Areas in Communications*. Vol. 24. 12. IEEE. 2006, pp. 2182–2195.
- [87] Gail Kaiser, Phil Gross, Gaurav Kc, Janak Parekh, and Giuseppe Valetto. “An approach to autonomizing legacy systems”. In: *Workshop on Self-Healing, Adaptive and Self-MANaged Systems (SHAMAN)*. 2002.

- [88] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. “Pegasus: A petascale graph mining system implementation and observations”. In: *IEEE international conference on data mining*. IEEE. 2009, pp. 229–238.
- [89] Jeffrey O Kephart and David M Chess. “The vision of autonomic computing”. In: *Computer*. IEEE. 2003, pp. 41–50.
- [90] Jeffrey O Kephart and Rajarshi Das. “Achieving self-management via utility functions”. In: *IEEE Internet Computing*. Vol. 11. 1. IEEE. 2007, pp. 40–48.
- [91] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H Campbell. “The case for reflective middleware”. In: *Communications of the ACM*. Vol. 45. 6. ACM. 2002, pp. 33–38.
- [92] Lars Kotthoff, Chris Thornton, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. “Auto-WEKA: Automatic Model Selection and Hyperparameter Optimization in WEKA”. In: *Automated Machine Learning*. Springer. 2019, pp. 81–95.
- [93] Young-Jou Lai, Ting-Yun Liu, and Ching-Lai Hwang. “Topsis for MODM”. In: *European journal of operational research*. Vol. 76. 3. Elsevier. 1994, pp. 486–500.
- [94] Kin Lane. “Overview of the backend as a service (BaaS) space”. In: *API Evangelist*. 2013.
- [95] Jean-Claude Laprie. “Dependable computing and fault-tolerance”. In: *Digest of Papers FTCS-15*. 1985, pp. 2–11.
- [96] Thomas Ledoux. “OpenCorba: A reflective open broker”. In: *International Conference on Metalevel Architectures and Reflection*. Springer. 1999, pp. 197–214.
- [97] Anna Levin, Shelly Garion, Elliot K Kolodner, Dean H Lorenz, Katherine Barabash, Mike Kugler, and Niall McShane. “AIOps for a Cloud Object Storage Service”. In: *International Congress on Big Data (BigDataCongress)*. IEEE. 2019, pp. 165–169.
- [98] Moises Levy and Jason O Hallstrom. “A reliable non-invasive approach to data center monitoring and management”. In: *Advances in Science, Technology and Engineering Systems Journal*. Vol. 2. 3. 2017, pp. 1577–1584.
- [99] Moises Levy and Daniel Raviv. “An overview of data center metrics and a novel approach for a new family of metrics”. In: *Advances in Science, Technology and Engineering Systems Journal*. Vol. 3. 2. 2018, pp. 238–251.
- [100] Lei Li, Kalyanaraman Vaidyanathan, and Kishor S Trivedi. “An approach for estimation of software aging in a web server”. In: *Proceedings International Symposium on Empirical Software Engineering*. IEEE. 2002, pp. 91–100.

- [101] Harold C Lim, Shivnath Babu, Jeffrey S Chase, and Sujay S Parekh. “Automated control in cloud computing: challenges and opportunities”. In: *Proceedings of the 1st workshop on Automated control for datacenters and clouds*. ACM. 2009, pp. 13–18.
- [102] Tong Liu, Hertong Song, et al. “Availability prediction and modeling of high mobility OSCAR cluster”. In: *International Conference on Cluster Computing*. IEEE. 2003, pp. 380–386.
- [103] Tania Llorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. “A review of auto-scaling techniques for elastic applications in cloud environments”. In: *Journal of grid computing*. Vol. 12. 4. Springer. 2014, pp. 559–592.
- [104] Manolis IA Lourakis et al. “A brief description of the Levenberg-Marquardt algorithm implemented by levmar”. In: *Foundation of Research and Technology*. Vol. 4. 1. 2005, pp. 1–6.
- [105] Samuel Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. “TAG: A tiny aggregation service for ad-hoc sensor networks”. In: *ACM SIGOPS Operating Systems Review*. Vol. 36. ACM. 2002, pp. 131–146.
- [106] Saeed Haddadi Makhsoos, Anton Gulenko, Odej Kao, and Feng Liu. “High Available Deployment of Cloud-based Virtualized Network Functions”. In: *International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2016, pp. 468–475.
- [107] Ming Mao and Marty Humphrey. “Auto-scaling to minimize cost and meet application deadlines in cloud workflows”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2011, pp. 1–12.
- [108] Ming Mao, Jie Li, and Marty Humphrey. “Cloud auto-scaling with deadline and budget constraints”. In: *11th IEEE/ACM International Conference on Grid Computing*. IEEE. 2010, pp. 41–48.
- [109] Lena Mashayekhy, Mahyar Movahed Nejad, Daniel Grosu, Quan Zhang, and Weisong Shi. “Energy-aware scheduling of mapreduce jobs for big data applications”. In: *IEEE transactions on Parallel and distributed systems*. Vol. 26. 10. IEEE. 2015, pp. 2720–2733.
- [110] Adnan Masood and Adnan Hashmi. “AIOps: Predictive Analytics & Machine Learning in Operations”. In: *Cognitive Computing Recipes*. Springer. 2019, pp. 359–382.
- [111] Matthew L Massie, Brent N Chun, and David E Culler. “The ganglia distributed monitoring system: design, implementation, and experience”. In: *Parallel Computing*. Vol. 30. 7. Elsevier. 2004, pp. 817–840.

- [112] Rivalino Matias and JF Paulo Filho. “An experimental study on software aging and rejuvenation in web servers”. In: *30th Annual International Computer Software and Applications Conference (COMPSAC)*. Vol. 1. IEEE. 2006, pp. 189–196.
- [113] Michael Maurer, Ivan Breskovic, Vincent C Emeakaroha, and Ivona Brandic. “Revealing the MAPE loop for the autonomic management of cloud infrastructures”. In: *symposium on computers and communications (ISCC)*. IEEE. 2011, pp. 147–152.
- [114] Leopoldo AF Mauricio, Marcelo G Rubinstein, and Otto CMB Duarte. “Proposing and evaluating the performance of a firewall implemented as a virtualized network function”. In: *7th International Conference on the Network of the Future (NOF)*. IEEE. 2016, pp. 1–3.
- [115] Andrew McAfee, Erik Brynjolfsson, Thomas H Davenport, DJ Patil, and Dominic Barton. “Big data: the management revolution”. In: *Harvard business review*. Vol. 90. 10. 2012, pp. 60–68.
- [116] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. “BRITE: An approach to universal topology generation”. In: *Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2001, pp. 346–353.
- [117] Hong Mei, Gang Huang, and Wei-Tek Tsai. “Towards self-healing systems via dependable architecture and reflective middleware”. In: *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*. IEEE. 2005, pp. 337–344.
- [118] Peter Mell, Tim Grance, et al. *The NIST definition of cloud computing*. National Institute of Standards and Technology, 2011.
- [119] Paul Menage, Paul Jackson, and Christoph Lameter. *CGROUPS*. URL: <https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt> (visited on 04/08/2019).
- [120] Daniel Menasce, Hassan Gomaa, Joao Sousa, et al. “Sassy: A framework for self-architecting service-oriented systems”. In: *IEEE software*. Vol. 28. 6. IEEE. 2011, pp. 78–85.
- [121] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. “Network function virtualization: State-of-the-art and research challenges”. In: *IEEE Communications Surveys & Tutorials*. Vol. 18. 1. IEEE. 2015, pp. 236–262.

- [122] Daniele Miorandi, Iacopo Carreras, Eitan Altman, Lidia Yamamoto, and Imrich Chlamtac. “Bio-inspired approaches for autonomic pervasive computing systems”. In: *Workshop on Bio-Inspired Design of Networks*. Springer. 2007, pp. 217–228.
- [123] Daniele Miorandi, David Lowe, and Lidia Yamamoto. “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.
- [124] Soumendra Mohanty and Sachin Vyas. “IT Operations and AI”. In: *How to Compete in the Age of Artificial Intelligence*. Springer. 2018, pp. 173–187.
- [125] Jesús Montes, Alberto Sánchez, Bunjamin Memishi, Maria S Pérez, and Gabriel Antoniu. “GMonE: A complete approach to cloud monitoring”. In: *Future Generation Computer Systems*. Vol. 29. 8. Elsevier. 2013, pp. 2026–2040.
- [126] Fabrizio Montesi and Janine Weber. “Circuit breakers, discovery, and API gateways in microservices”. In: *CoRR*. Vol. 1609.05830. 2016.
- [127] Moogsoft. *Understanding the Machine Learning in AIOps*. 2018. URL: <https://www.moogsoft.com/resources/aiops/white-paper/understanding-machine-learning> (visited on 10/10/2019).
- [128] Justin Moore, Jeff Chase, Keith Farkas, and Partha Ranganathan. “A sense of place: Toward a location-aware information plane for data centers”. In: *Hewlett Packard Technical Report*. 2004.
- [129] Justin Moore, Jeff Chase, Keith Farkas, and Parthasarathy Ranganathan. “Data center workload monitoring, analysis, and emulation”. In: *Eighth Workshop on Computer Architecture Evaluation using Commercial Workloads*. 2005, pp. 1–8.
- [130] Jorge J Moré. “The Levenberg-Marquardt algorithm: implementation and theory”. In: *Numerical analysis*. Springer. 1978, pp. 105–116.
- [131] John Mugler, Thomas Naughton, Stephen L Scott, Brian Barret, Andrew Lumsdaine, Jeffrey M Squyres, Benoît des Ligneris, Francis Giraldeau, and Chokchai Leangsuksun. “Oscar clusters”. In: *Linux Symposium*. Citeseer. 2003.
- [132] Arun Babu Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L Scott. “Proactive fault tolerance for HPC with Xen virtualization”. In: *Proceedings of the 21st annual international conference on Supercomputing*. ACM. 2007, pp. 23–32.
- [133] Hiep Nguyen, Zhiming Shen, Yongmin Tan, and Xiaohui Gu. “FChain: Toward black-box online fault localization for cloud systems”. In: *33rd International Conference on Distributed Computing Systems*. IEEE. 2013, pp. 21–30.
- [134] David L Olson. “Comparison of weights in TOPSIS models”. In: *Mathematical and Computer Modelling*. Vol. 40. 7-8. Elsevier. 2004, pp. 721–727.

- [135] Randal S Olson and Jason H Moore. “TPOT: A tree-based pipeline optimization tool for automating machine learning”. In: *Automated Machine Learning*. Springer. 2019, pp. 151–160.
- [136] Rihards Olups. *Zabbix 1.8 network monitoring*. Packt Publishing Ltd, 2010.
- [137] Raja Parasuraman, Thomas B Sheridan, and Christopher D Wickens. “A model for types and levels of human interaction with automation”. In: *IEEE Transactions on systems, man, and cybernetics – Part A: Systems and Humans*. Vol. 30. 3. IEEE. 2000, pp. 286–297.
- [138] Douglas F Parkhill. *Challenge of the computer utility*. Addison-Wesley, 1966.
- [139] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, et al. *Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies*. Tech. rep. UCB//CSD-02-1175, UC Berkeley Computer Science, 2002.
- [140] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. “Sensing as a service model for smart cities supported by internet of things”. In: *Transactions on emerging telecommunications technologies*. Vol. 25. 1. Wiley Online Library. 2014, pp. 81–93.
- [141] Matthias Poloczek, Jialei Wang, and Peter I Frazier. “Warm starting Bayesian optimization”. In: *Winter Simulation Conference (WSC)*. IEEE. 2016, pp. 770–781.
- [142] Rahul Potharaju and Navendu Jain. “When the network crumbles: An empirical study of cloud network failures and their impact on services”. In: *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM. 2013.
- [143] Harald Psailer and Schahram Dustdar. “A survey on self-healing systems: approaches and systems”. In: *Computing*. Vol. 91. 1. Springer. 2011, pp. 43–73.
- [144] Massimiliano Rak, Salvatore Venticinque, Gorka Echevarria, Gorka Esnal, et al. “Cloud application monitoring: The mosaic approach”. In: *Third International Conference on Cloud Computing Technology and Science*. IEEE. 2011, pp. 758–763.
- [145] Brian Randell. “System structure for software fault tolerance”. In: *IEEE transactions on software engineering*. IEEE. 1975, pp. 220–232.
- [146] Mauro Ribeiro, Katarina Grolinger, and Miriam AM Capretz. “Mlaas: Machine learning as a service”. In: *14th International Conference on Machine Learning and Applications (ICMLA)*. IEEE. 2015, pp. 896–902.

- [147] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. “A taxonomy and survey of cloud computing systems”. In: *International Joint Conference on INC, IMS and IDC*. IEEE. 2009, pp. 44–51.
- [148] Gabi Dreo Rodosek, Kurt Geihs, Hartmut Schmeck, and Burkhard Stiller. “Self-Healing Systems: Foundations and Challenges”. In: *Self-Healing and Self-Adaptive Systems*. Citeseer. 2009.
- [149] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. “Efficient autoscaling in the cloud using predictive models for workload forecasting”. In: *4th International Conference on Cloud Computing*. IEEE. 2011, pp. 500–507.
- [150] Felix Salfner, Maren Lenk, and Mirosław Malek. “A survey of online failure prediction methods”. In: *ACM Computing Surveys (CSUR)*. Vol. 42. 3. ACM. 2010.
- [151] Florian Schmidt, Anton Gulenko, Marcel Wallschläger, Alexander Acker, Vincent Hennig, Feng Liu, and Odej Kao. “IFTM-Unsupervised Anomaly Detection for Virtualized Network Function Services”. In: *International Conference on Web Services (ICWS)*. IEEE. 2018, pp. 187–194.
- [152] Florian Schmidt, Florian Suri-Payer, Anton Gulenko, Marcel Wallschläger, Alexander Acker, and Odej Kao. “Unsupervised Anomaly Event Detection for Cloud Monitoring Using Online Arima”. In: *International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE/ACM. 2018, pp. 71–76.
- [153] Florian Schmidt, Florian Suri-Payer, Anton Gulenko, Marcel Wallschläger, Alexander Acker, and Odej Kao. “Unsupervised Anomaly Event Detection for VNF Service Monitoring Using Multivariate Online Arima”. In: *International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2018, pp. 278–283.
- [154] Chris Schneider, Adam Barker, and Simon Dobson. “A survey of self-healing systems frameworks”. In: *Software: Practice and Experience*. Vol. 45. 10. Wiley Online Library. 2015, pp. 1375–1398.
- [155] Christoph Schuler, Roger Weber, Heiko Schuldt, and H-J Schek. “Scalable peer-to-peer process management-the OSIRIS approach”. In: *IEEE International Conference on Web Services*. IEEE. 2004, pp. 26–34.
- [156] Mina Sedaghat, Eddie Wadbro, John Wilkes, Sara De Luna, Oleg Seleznev, and Erik Elmroth. “Diehard: reliable scheduling to survive correlated failures in cloud data centers”. In: *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE. 2016, pp. 52–59.
- [157] Jin Shao and Qianxiang Wang. “A performance guarantee approach for cloud applications based on monitoring”. In: *35th Annual Computer Software and Applications Conference Workshops*. IEEE. 2011, pp. 25–30.

- [158] Michael W Shapiro. "Self-healing in modern operating systems". In: *Queue*. Vol. 2. 9. ACM. 2004, pp. 66–75.
- [159] Onn Shehory. "A self-healing approach to designing and deploying complex, distributed and concurrent software systems". In: *International Workshop on Programming Multi-Agent Systems*. Springer. 2006, pp. 3–13.
- [160] Xiang Sheng, Jian Tang, Xuejie Xiao, and Guoliang Xue. "Sensing as a service: Challenges, solutions and future directions". In: *IEEE Sensors journal*. Vol. 13. 10. IEEE. 2013, pp. 3733–3741.
- [161] Thomas B Sheridan and William L Verplank. *Human and computer control of undersea teleoperators*. Tech. rep. MIT Man-Machine Systems Lab, 1978.
- [162] Martin L Shooman. *Reliability of computer systems and networks: fault tolerance, analysis, and design*. John Wiley & Sons, 2003.
- [163] Anil Somayaji, Steven Hofmeyr, and Stephanie Forrest. "Principles of a computer immune system". In: *NSPW*. Vol. 97. 1997, pp. 75–82.
- [164] Jonathan Steuer. "Defining virtual reality: Dimensions determining telepresence". In: *Journal of communication*. Vol. 42. 4. Wiley Online Library. 1992, pp. 73–93.
- [165] Nenad Stojnić and Heiko Schuldt. "Osiris-sr: A safety ring for self-healing distributed composite service execution". In: *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press. 2012, pp. 21–26.
- [166] Da-Wei Sun, Gui-Ran Chang, Shang Gao, Li-Zhong Jin, and Xing-Wei Wang. "Modeling a dynamic data replication strategy to increase system availability in cloud computing environments". In: *Journal of computer science and technology*. Vol. 27. 2. Springer. 2012, pp. 256–272.
- [167] Gerald Tesauro, David M Chess, William E Walsh, Rajarshi Das, Alla Segal, Ian Whalley, Jeffrey O Kephart, and Steve R White. "A multi-agent systems approach to autonomic computing". In: *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*. IEEE Computer Society. 2004, pp. 464–471.
- [168] Lauritz Thamsen, Ilya Verbitskiy, Florian Schmidt, Thomas Renner, and Odej Kao. "Selecting resources for distributed dataflow systems according to runtime targets". In: *35th International Performance Computing and Communications Conference (IPCCC)*. IEEE. 2016, pp. 1–8.
- [169] Marialena Vagia, Aksel A Transeth, and Sigurd A Fjerdings. "A literature review on the levels of automation during the years. What are the different taxonomies that have been proposed?" In: *Applied ergonomics*. Vol. 53. Elsevier. 2016, pp. 190–202.

- [170] Kalyanaraman Vaidyanathan and Kishor S Trivedi. “Extended classification of software faults based on aging”. In: *International Symposium on Software Reliability Engineering*. Citeseer. 2001.
- [171] Robbert Van Renesse, Kenneth P Birman, and Werner Vogels. “Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining”. In: *ACM transactions on computer systems (TOCS)*. Vol. 21. 2. ACM. 2003, pp. 164–206.
- [172] Steven Van Rossem, Wouter Tavernier, Balazs Sonkoly, Didier Colle, Janos Czentye, Mario Pickavet, and Piet Demeester. “Deploying elastic routing capability in an SDN/NFV-enabled environment”. In: *Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. IEEE. 2015, pp. 22–24.
- [173] Mark Velasquez and Patrick T Hester. “An analysis of multi-criteria decision making methods”. In: *International Journal of Operations Research*. Vol. 10. 2. 2013, pp. 56–66.
- [174] William Voorsluys, James Broberg, Rajkumar Buyya, et al. “Introduction to cloud computing”. In: *Cloud computing — Principles and Paradigms*. Wiley Online Library. 2011, pp. 1–41.
- [175] Marcel Wallschläger, Anton Gulenko, Florian Schmidt, Alexander Acker, and Odej Kao. “Anomaly Detection for Black Box Services in Edge Clouds Using Packet Size Distribution”. In: *International Conference on Cloud Networking (CloudNet)*. IEEE. 2018.
- [176] Marcel Wallschläger, Anton Gulenko, Florian Schmidt, Odej Kao, and Feng Liu. “Automated Anomaly Detection in Virtualized Services Using Deep Packet Inspection”. In: *Procedia Computer Science*. Vol. 110. Elsevier. 2017, pp. 510–515.
- [177] Yingxu Wang. “On autonomous computing and cognitive processes”. In: *Proceedings of the Third IEEE International Conference on Cognitive Informatics*. IEEE. 2004, pp. 3–4.
- [178] Jianping Weng, Jessie Hui Wang, Jiahai Yang, and Yang Yang. “Root cause analysis of anomalies of multitier services in public clouds”. In: *IEEE/ACM Transactions on Networking*. Vol. 26. 4. IEEE. 2018, pp. 1646–1659.
- [179] Xindong Wu, Xingquan Zhu, Gong-Qing Wu, and Wei Ding. “Data mining with big data”. In: *IEEE transactions on knowledge and data engineering*. Vol. 26. 1. IEEE. 2013, pp. 97–107.
- [180] Min Xie, Yuan-Shun Dai, and Kim-Leng Poh. *Computing system reliability: models and analysis*. Springer Science & Business Media, 2004.

- [181] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing”. In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 2–2.
- [182] B Zaluski, B Rajtar, H Habjanic, M Baranek, N Slibar, R Petracic, and T Sukser. “Terastream implementation of all IP new architecture”. In: *International Conference on Information & Communication Technology Electronics & Microelectronics (MIPRO)*. IEEE. 2013, pp. 437–440.
- [183] Arkady Zaslavsky, Charith Perera, and Dimitrios Georgakopoulos. “Sensing as a service and big data”. In: *arXiv preprint arXiv:1301.0159*. 2013.
- [184] Qi Zhang, Lu Cheng, and Raouf Boutaba. “Cloud computing: state-of-the-art and research challenges”. In: *Journal of internet services and applications*. Springer. 2010, pp. 7–18.
- [185] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. “BIRCH: an efficient data clustering method for very large databases”. In: *ACM Sigmod Record*. 1996, pp. 103–114.
- [186] Weizhong Zhao, Huifang Ma, and Qing He. “Parallel k-means clustering based on mapreduce”. In: *IEEE International Conference on Cloud Computing*. Springer. 2009, pp. 674–679.
- [187] Paul Zikopoulos, Chris Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.