FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Data-Driven Automatic Deployment in Edge Computing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Sebastian Meixner, BSc
Matrikelnummer 1126467

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dr.-Ing. Stefan Schulte
Mitwirkung: Dr. Daniel Schall

Wien, 30. April 2018

Sebastian Meixner | Stefan Schulte

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Data-Driven Automatic Deployment in Edge Computing

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Sebastian Meixner, BSc
Registration Number 1126467

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dr.-Ing. Stefan Schulte
Assistance: Dr. Daniel Schall

Vienna, 30th April, 2018

Sebastian Meixner Stefan Schulte

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Sebastian Meixner, BSc
Vienna

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 30. April 2018

Sebastian Meixner

Acknowledgements

First and foremost I would like to thank Stefan Schulte and Daniel Schall for their constructive, honest, and invaluable feedback I received during the course of writing this thesis. Also, I am very grateful for the guidance they provided me with and the patience they had.
Furthermore, I want to express my gratitude to Fei Li and Konstantinos Plakidas for the insightful discussions I was able to have with them about the content of this thesis.
Lastly, I want to thank my family and friends for their moral support, especially during the final stages of my studies.

Kurzfassung

Mit der steigenden Popularität des Internet of Things sehen wir immer häufiger, dass versucht wird das traditionelle Cloud-Computing mit Ressourcen am Rande des Netzwerks (der Edge) zu verbinden. Dadurch wird es ermöglicht die unterschiedlichen Vor- und Nachteile der beiden Plattformtypen auszunutzen. Allerdings bringt das Zusammenführen der beiden Arten von Plattformen neue Herausforderungen, sowohl für Entwickler als auch für das Betriebspersonal, mit sich, da es immer schwieriger wird festzulegen wie Services, basierend auf ihren nicht funktionalen - und Laufzeitanforderung, verteilt werden sollen, während die verfügbaren Ressourcen auf der Edge optimal ausgenutzt werden.

Händisch zu entscheiden, wo jedes einzelne der Services laufen soll und diese dann händisch zu verteilen, wird zu einer nicht bewältigbaren Aufgabe, im Speziellen wenn es sich um eine große Anzahl an Services handelt, was oft der Fall ist wenn eine Microservice Architektur zum Einsatz kommt. Weiters ist es notwendig, dass, wenn die Services einmal verteilt sind, ihr Laufzeit-Verhalten zu überwachen um eine Verschlechterung der Quality of Service Parameter, sowohl der einzelnen Services, als auch des gesamten Systems, feststellen zu können. Dadurch wird es möglich entsprechend Handlungen zu setzten um die Verletzung von Service Level Agreements zu verhindern. Außerdem können die gesammelten Informationen verwendet werden um eine Optimierung von zukünftigen Verteilungsprozesse zu ermöglichen.

In dieser Arbeit schlagen wir eine ganzheitliche Herangehensweise vor, die sowohl Entwickler und als auch das Betriebspersonal bei der Entwicklung, dem Verteilen und dem Betreiben von Applikationen, die einem Miscroservice Muster folgen, unterstützt. Um dies zu erreichen implementieren wir das Data-Driven Automatic Deployment Framework in einer prototypischen Umsetzung. Dieses erlaubt es Applikationen transparent auf Cloud- und Edge-Infrastruktur zu verteilen. Weiters stellt es einen einheitlichen Überwachungsmechanismus für Services zur Verfügung, welcher einen Event-basierten Mechanismus zur Laufzeit Adaptierung ermöglicht.

Abstract

With the growing popularity of the Internet of Things, we see a trend towards combining traditional cloud computing with resources available at the edge of the network. This way it becomes possible to exploit the complementary characteristics of both types of platforms. However, unifying the two types of platforms poses new challenges to developers and operational staff alike, as it becomes increasingly harder to determine where services should run based on their non-functional- and runtime-requirements, while simultaneously utilizing the resources at hand in an optimal way.

Manually deciding where each individual service should run, and rolling them out becomes unfeasible, especially with a large number of individual services, which tends to be the case in a microservice architecture. Furthermore, once the services are deployed into production, it becomes necessary to monitor their runtime behavior to detect a deterioration of the individual services' quality of service parameters as well as those of the system as a whole. Thereby, it becomes possible to take actions to prevent quality of service and service level agreement violations. Additionally, the collected information can be used to optimize future the deployment plans for the services.

In this work we propose a holistic approach towards supporting developers and operational staff in creating and running applications that employ a microservice architectural pattern. To realize this approach we prototypically implement a Data-Driven Automatic Deployment framework which allows the transparent deployment of services onto cloud and edge hosts alike. Furthermore, it provides a uniform monitoring mechanism for the services, which enables an event-based mechanism for runtime adaptation.

Contents

Kurzfassung **ix**

Abstract **xi**

1 Introduction **1**
1.1 Motivating Example 4
1.2 Contributions 7
1.3 Organization 8

2 Background **9**
2.1 Fog Computing 9
2.2 Microservice Architectures 10
2.3 DevOps Methodology 11

3 State of the Art **13**
3.1 Fog Computing 13
3.2 Automatic Deployment 16
3.3 Runtime Monitoring and Adaptation 17
3.4 Summary 18

4 The DDAD Framework **21**
4.1 Requirements 21
4.2 Key Design Decisions 26
4.3 Main Components 31
4.4 Static System View 42
4.5 Dynamic System View 42
4.6 Summary 56

5 Evaluation **59**
5.1 Setup and Context 60
5.2 Performance Measurements 62
5.3 Summary 74

6 Discussion & Conclusion **77**

6.1 Comparison to Related Work 77
6.2 Limitations and Future Work 79
6.3 Summary 80

List of Figures **83**

List of Tables **85**

Listings **87**

Acronyms **89**

Bibliography **91**

CHAPTER 1

Introduction

In the Internet of Things (IoT) there are two distinct types of computing platforms. First there are edge platforms that reside at the edge of the network. They consist of low powered devices that have limited resources. In an industry context these devices reside on a plant operator's premises and might include some machines that are part of an assembly line (e.g., welding robots or a milling machine) which offer some of their computational resources to the edge platform operator. They might also be low powered PCs that are located in a factory, micro-servers, or dedicated, low-powered IoT devices (e.g., the Raspberry Pi single board computer[1]). In the more general context, these devices do not even have to be stationary and can also include smart phones, tablets, or other wireless devices that might join or leave particular networks in a hardly predictable manner [17, 37, 39].

Edge platforms lend themselves very well to achieving narrow time constraints, since most devices on the edge are generally in spatial proximity to each other, which reduces the distance and therefore time data needs to travel to be processed. However, the fact that things like storage and computational resource are limited at the edge, reduces the set of applications that are feasible to run there [6].

Secondly, in contrast to edge platforms, there are cloud platforms, which provide their users with virtually unlimited resources which means that a wide variety of application can run on it, as long as (near-)real-time or privacy guarantees are not things sought after [39]. The inability to achieve (near-)real-time constraints, stems from the fact that data that is used by cloud services, needs to be transferred over the internet into the cloud. Then, the data is used by the services, and lasty the result is transmitted back, again over the internet. This sending of data incurs an overhead that is unacceptable for low latency applications [6]. Also, a possible lack of privacy is introduced to cloud services,

[1] https://www.raspberrypi.org/

because the data leaves the users' premises and cloud providers could theoretically do what they please with the data they receive. There are several levels of abstraction cloud platforms might offer. According to Liu et al. [28] there are Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

IaaS abstracts the underlying infrastructure, providing their users access to Virutal Machines (VMs). Users can rent these VMs that usually come in different sizes (w.r.t. the resources they offer) and bigger machines cost more. The billing usually happens based on the time for which a machine was rented. An example for an IaaS cloud would be Amazon's Elastic Compute Cloud (EC2)[2].

A PaaS cloud offers its users an abstract platform to which she can deploy services without having to worry about setting up the environment of the application. Billing usually is based on the consumed resources like storage or bandwidth. An example for such a PaaS cloud would be a CloudFoundry[3] installation, on a multitude of EC2 instances.

Lastly, a SaaS cloud offers the software itself to their users. This means that the users can utilize the software without having to deploy it or take care of it in any other way. An example would be a web service, that the cloud provider offers to its users, which is hosted on a CloudFoundry installation, which in turn is distributed among a multitude of EC2 instances.

Another difference between cloud and edge platforms are the costs they introduce for using their resources. When utilizing resources at the edge, the only costs incurred are for power and possibly cooling of the edge devices. Generally edge devices are already in place on the users' premises, so there are no upfront hardware costs to the edge platform. Cloud platforms however do create costs for the users, based on how much of which resource is used, since they generally operate on a pay-as-you-go basis. The fact that resources at the edge are basically free while cloud resources mean additional costs for the users, imply that it is favorable to use edge resources whenever possible, to minimize the costs. However, edge platforms are generally less reliable than cloud platform, since devices may arbitrarily join, and more importantly, leave the network [13].

The abovementioned difference in pricing is only true, if the users employ devices they actually own. In a general scenario it is possible that users might offer their computational power at the edge to other interested parties. However, the question how billing could be realized in this scenario, and how Service Level Agreements (SLAs) and agreed upon Quality of Service (QoS) parameters could be enforced remains an open one [37]. Another question that needs to be answered is how to provide incentives for users that offer their resources in an edge computing context, aside from monetary ones [13].

[2]https://aws.amazon.com/ec2/
[3]https://www.cloudfoundry.org/

In general it is deemed desirable to exploit the complementary characteristics of cloud and edge platforms and to use the platform which is best suited for a service's needs [6]. This can be achieve by having some services run on the edge while other applications run in the cloud. For example, a service that detects outliers in sensor readings, which can be done with limited resources and often needs to happen in a timely fashion, should run on the edge. Contrary to that, big data analysis would not be feasible there because of the sheer amount of computational power that is needed. Furthermore, such services generally have liberal time constraints that are in the minutes if not even hours, which makes the cloud the ideal platform to deploy them to.

The problem developers and operational staff are facing, is that it might not be immediately clear what the best deployment location for an application might be. The legal, or even optimal, location for deploying a service can depend on several things, like the resource the service needs to function properly, the software requirements (e.g., a certain operation system, or the runtime of a programming langauge) that the hosting device needs to meet. Other things that might limit the legal deployment locations could be privacy concerns attached to the data that the service produces. Keeping all these requirements in mind becomes especially difficult when a microservice architecture [15] is employed, where each application is comprised of a multitude of services, along with a DevOps methodology [5], where services might be deployed multiple times a day. Manually deciding where each service should run, which also includes selecting the appropriate device at the edge, becomes a tedious and error-prone task, which needs to be automated to free the developers and operational staff of this burden.

It is important to keep in mind, that the optimal deployment location of the individual services may change over time. This stems from the fact that it does not only depends on the static configuration of the services' Non-Functional-Requirements (NFRs), resource and software requirements, as well the hosts' resource and software offerings, but also on the runtime behavior of the services. Unexpected or changing runtime behavior can have several reasons, like a wrongly assessed resource consumption, or unexpected behavior of third-party services. This problem also becomes even more apparent when using microservices and a DevOps methodology, because manually deciding the exact deployment location would mean that the person in charge would constantly needs to adapt the deployment configuration.

Lastly, it is desirable to adapt which services are available to others during runtime, by dynamically activating and deactivating them to dynamically redistribute the workload across the remaining services. This way, individual edge devices can be kept from becoming overloaded and failing to respond. This process needs to happen based on the current state of the system (i.e., the current workload distribution, especially over the edge hosts) and aims to prohibit the interference of user services with the edge devices' primary tasks (i.e., the actual task the device was intended to achieve). The second point becomes especially important when combining cloud and edge computing in an industrial context. Here the edge devices' have primary task that are often important for the safety

of staff or for the proper functioning of an assembly line. Thus, it is prohibited that the user-defined services interfere with these tasks in any way. Although it is possible for user services to also be mission-critical, throughout the work it is assumed that none of them are and they are not relevant to safety. This means that they can be interrupted or migrated at any point in time, without having to take precautions to ensure extremely high availability and possibly (near-)real-time constraints.

One possible solution, for combining the computational power of the cloud, with the low latency possible at the edge is so called *fog computing* [6]. Although there is no clear definition of what the term fog computing actually refers to [37], there is a clear consensus, that fog computing involves the cloud, as well as the edge [1, 6]. This can be achieved in different ways. However, we argue that the most fitting definition of fog computing is given by Vaquero and Rodero-Merino [37] which describe it as a scenario, in which a large set of devices forms a network that provides users with the possibility to deploy applications onto them. The heterogeneous nature of the devices is abstracted and they offer a sandboxed environment for the execution of applications [37]. Furthermore, they are enabled to communicate with each other, which facilitates the usage of the services deployed onto them [37].

1.1 Motivating Example

Vaquero and Rodero-Merino also mention that the owners of the devices that participate in fog computing should be compensated for offering (parts of) their devices [37]. As an example for such a compensation one could imagine that a certain amount of resources of a device is rented in exchange for a fee the user, similar to Amazon's EC2 offerings. However, another incentive for participating in fog computing can be to save money, by using less resources in the cloud, as long as the current workload allows execution on the edge devices, which generally do not have high capacities [17]. Thus, it is easily imaginable that users who already have a multitude of devices in place, which do not need their full computational power at all time, might want to use such an approach to cut their costs for cloud resources. However, the devices in place might occasionally need (almost) all of their power, which means that relying solely on the edge platform would either result in applications being stopped to free the resources they use, or in devices being unable to access the resources they need. Both of these scenarios are generally undesirable. Therefore, a combination of both cloud and edge computing should be used.

A use case that fulfills the above mentioned criteria and demonstrates the benefits of bringing together cloud and edge computing can be found in an industrial context. Here, plant owners already have a multitude of different devices at their disposal (e.g., welding robots, milling machines, industrial PCs, ...) which might not use all of their resources, all the time. Furthermore, they have assets (e.g., an electrical drive) whose condition (e.g., the voltage they draw, their motor temperature, the vibration they cause) is continuously monitored by a multitude of sensors. It is then possible to draw inferences from the

readings of these sensors about the current health of the machine. This can be done by employing machine learning techniques to facilitate predictive maintenance. When using them, historical data needs to be collected. Then, this data needs to be classified into clusters that represent states of an asset (e.g., a high motor temperature in combination with a low ambient temperature might indicate a problem in the asset's cooling), which afterwards need to be labeled. From the data and the labeled clusters, a model is trained that can be used to classify new data and predict its membership of a certain cluster. This classification of incoming, new data is also referred to as *scoring*.

To obtain a model, which can later be scored, the locally collected data is transferred to the cloud via some connectivity service (we assume that the users have a PaaS cloud at their disposal). Machine learning techniques are then used to cluster the data. Afterwards the user has to assign a label to each cluster, which correspond to a state of the machine. When the clustering and labeling are completed, the actual training of the model takes place employing specialized machine learning techniques, like a random forests [27]. This model is then able to classify new data according to the previously obtained classification.

The resulting model is transformed into different formats that are understood by different machine learning engines, and the resulting files are stored in a model registry. This way, the trained model is also made available for other interested parties in the model registry. Normally, the user would now need to decide whether they want to score their data in the cloud or at the edge. This means, there would either be a service deployed in the cloud that takes all scoring requests or one scoring service for each edge device, where users can score locally. The problem stemming from the first option, is that it demands an unnecessarily large amount of cloud resources, which results in increased costs for the users. The second option does not take into account that edge devices might not be dedicated solely to scoring the model, but might have other tasks that have a higher priority.

We explicitly want to use the available resource at the edge in combination with the cloud. To achieve this, we define two services. One scoring service that must be deployed onto an edge device at the users' premises, and one service that can score models in the cloud and acts as a "fallback" for the local scoring service. This needs to be done since most edge devices only execute user services as secondary tasks, which must not interfere with their primary tasks. An example for such a primary task would be executing the defined step in a production process of a welding robot. Since interference with the primary task is not allowed, the devices could decide to interrupt the execution of the scoring, should the workload of the primary task call for it. Thus, we would not be able to guarantee the availability of the scoring service by only using edge-deployment. Since we deploy one scoring service to the cloud, we get the benefit that it is inherently scalable, which allows to adapt the usage of cloud resources depending on the workload on edge devices. Furthermore, users can easily share their trained models with each other, across their own premises, or with other interested parties (e.g., machine builder).

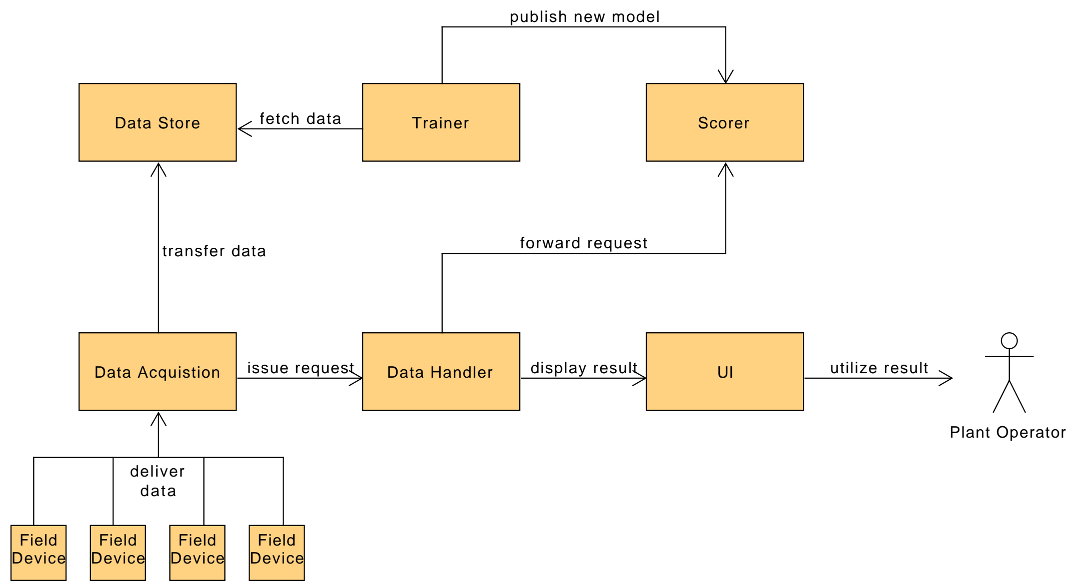

Figure 1.1: Logical View of the Motivating Example

Figure 1.1 shows the basic components of the motivating example. There is a multitude of field devices that deliver their data to the Data Acquisition component, which forwards it to some Data Store and/or to a Data Handler. The Data Handler uses the Scorer to classify the incoming data and displays the result via some kind of User Interface (UI). This UI is then used by the plant operator to utilize the obtained scoring results to plan maintenance accordingly. The Scorer receives the trained models from the Trainer, which in turn gets the data from the Data Store. A problem that arises, is that the Scorer might either run in the cloud, locally, or both. Thus we need a mechanism to know where it runs and which actual service instance should be invoked to optimally fulfill the system's NFRs.

This use case aims to demonstrate the capabilities and benefits our proposed solution has. It shows the need for a mechanism to deploy services onto edge devices. However, the exact nature of those device is not known upfront, so a generic method for deploying the services, which does not rely on any device specific aspects is needed. This way, the user is enabled to utilize the resources at the edge, to minimize the usage of cloud resources. Apart from that, the use case shows that there is a need for a mechanism that detects devices that cannot handle the workload incurred upon them and instruct services to not use services located on those devices anymore.

The presented scenario helps us to determine important additional requirements that need to be fulfilled to properly use resources available at the edge in combination with cloud resources. First, there is the need for a method to decide where individual services are allowed to be deployed based on their NFRs which are defined by the user beforehand. This needs to be done, since some data might not be allowed to leave the users' premises because they have some privacy concerns attached or because they would reveal business information that needs to be protected. It is also imaginable that users have a small private cloud at their disposal, which would be able to handle these kinds of request, but using this data in a public cloud would not be allowed. After deciding which platform is able to host the service, there is the need to determine where exactly on this platform the service is deployed, should it be an IaaS cloud. This decision has to happen based on the resources a service needs to properly fulfill its task, along with the software it demands to function and the NFRs it has to adhere to.

After the services have been deployed, they have to be monitored to detect undesired behavior like bottlenecks, that might impact their proper functioning of certain devices or potentially the whole system. The results of the monitoring needs to be aggregated, preprocessed, and visualized for the users. Thereby enabling them to observe the system and learn from its behavior. This could, for example, result in an adaption of the services NFRs or resource needs. Since monitoring a complex system comprised of a multitude of services in a meaningful way is a tedious task the user should be able to define rules upon which the system needs to react to anticipated events (e.g., an edge device becoming overloaded) and handle some kind of adaptation automatically. These rules are evaluated based on the metrics that are collected by the system. Lastly, the result of these action need to be propagated throughout the system to allow the devices and services to act appropriately and adapt to the newly obtained information.

1.2 Contributions

The goal of our work is to provide a holistic framework to the user that allows the detailed definition of services, hosts, and platforms, along with their capabilities as well as their runtime- and non-functional-requirements. It aims to facilitate transparent, automated deployment to edge devices, and to provide a mechanism for runtime monitoring and adaptation of deployed services. Our main contributions can be summarized as follows:

(1) Defining and implementing a transparent method for cloud-edge deployment. Thereby allowing users to utilize resources at the edge of the network and combine them with the power of virtually unlimited resources in the cloud

(2) The implementation of a process to determine the optimal deployment location of individual services, taking into account their NFRs, as well as resource and software needs

(3) The implementation of a monitoring process that incurs little overhead and enables (4)

(4) Providing a mechanism for runtime adaptation of IoT applications, which is achieved by employing monitoring techniques, Complex Event Processing (CEP), and registry-aware service clients

1.3 Organization

This thesis will be organized as follows

Section 2 covers the basics of fog computing, microservice architectures, and the DevOps methodology.

Section 3 presents current research in the field of edge computing as well as automatic deployment, and runtime monitoring and adaptation, since these fields are at the core of our framework

Section 4 then introduces the Data-Driven Automatic Deployment (DDAD) framework itself. The section defines the main requirements of the framework we identified and presents the key design decisions we made during its realization. Furthermore, it gives an architectural overview and presents the framework's main components in greater detail. It also offers an in depth discussion of the framework's implementation.

Section 5 will display the experimental results of the benefits of the proposed framework on the basis of the application used as a motivating example. To evaluate the validity of our approach, the presented use case is implemented and our framework is used to manage the services' lifecycle, from deployment planning, over the execution of the planned deployment strategy, to monitoring the individual service and adapting them based on the observed runtime behavior.

Section 6 will reflect upon our work. The section discusses shortcomings of the framework and possible future work to be done based upon it. It will also conclude our work and summarize our findings.

CHAPTER 2

Background

2.1 Fog Computing

The ultimate goal of *fog computing*, or *edge computing* is to overcome issues that are inherent to the traditional cloud computing paradigm [1, 6]. These issues are first and foremost the latency that is introduced when using cloud services, and privacy concerns that are associated with data that is utilized by some applications (e.g., patient data used by health-care applications) [20].

Another important problem that researchers want to solve stems from the fact that mobile devices (e.g., smartphones or tablets) are very constraint in their resources compared to what users want to achieve with them. Thus, it is deemed desirable to extend their capabilities by allowing them to use cloud resources seamlessly and transparently for the user [23, 33].

A key goal of fog computing is to utilize resources at the edge of the network which helps to achieve low latency for critical parts of an application, while using the cloud if possible or necessary. As an example for such an application Bonomi et al. present a *Smart Traffic Light System* [6], where each intersection is equipped with a smart traffic light and there is communication across intersections. In this use case the system has several responsibilities, which all have different NFRs and demand different (amounts of) resource. The authors identify key requirements for their use case, that can be extended to a general fog application. These include a middleware platform that orchestrates the individual software components, as well as a well-defined and uniform mechanism for participating devices to communicate with each other.

Additionally to enabling the communication of the individual devices, this middleware platform has to facilitate the interplay of the edge with the cloud [6]. This need for interaction stems from the fact that the system collects data which can then be analyzed

to improve the system itself. Based on the sheer amount of data that is collected and needs to be analyzed it would not be feasible to do this computation at the edge. Thus, cloud services, which can achieve these tasks, have to be made available to the services residing on the edge [6].

Some of the problems that hinder the usage of edge resources to their full potential, stem from the fact that the edge in general consists of a multitude of heterogenous devices that need to be abstracted. Furthermore, devices that reside at the edge are often wireless and mobile, which means that they can unexpectedly leave a network, which is an issue that has to be dealt with [13].

There are several different reasons and approaches on how to bring cloud and edge computing together in a meaningful way and thereby combine their complementary benefits and drawbacks. Recurring use cases for fog computing include time critical applications. The need for low latency applications varies from application to application, but can be generally summarized as either stemming from the fact that too high response times would interfere with the user experience [39], or have a critical impact on the system state [6].

2.2 Microservice Architectures

As modern software systems are getting more complex and distributed, traditional and monolithic applications are no longer a viable option of software development. Thus Fowler [15] presents microservices. They are an architectural pattern realizing an improved version of the Service Oriented Architecture (SOA) style [18]. Is used by well known companies like Netflix to cope the growing complexity of their systems [18]. Micoservices typically make use of some core services (e.g., storage, messaging ...) which are provided by the platform they run on. The web services offered by Amazon, can be seen as such core services which enable the developers to build upon them to create more complex software systems [18]. Apart from complexity, scalability and resilience become major issues when designing, implementing, and operating highly distributed systems as companies like Amazon and Netflix do. The main idea behind a microservice-based architecture is that each deployable service is a software component that has exactly one well-defined task [34]. Other services do not need to know how it works internally as long as it behaves as expected and does its defined task (i.e., each service acts a black box to other services) and to be available to other services each service has to have a well-defined interface through which it can be invoked [16]. Typically these interface are realized by Representational State Transfer (REST) endpoints [34]. Another important factor when developing microservices is that each service has its own data storage to which only instances of the service itsef have direct access. This implies that no service can access another one's data directly [34], which in turn leads to better encapsulation.

By giving each service well-defined responsibilities and capabilities, it enables developers and operational staff to easily test services in isolation, by mocking or simulating the

services they depend on. Furthermore, by exposing only a well-defined interface without relinquishing anything about its internal workings (e.g., which implementation language was used, which services are used in the background ...) different implementations of services become easily interchangeable [3]. This does not only decouple the individual services from each other, but also their build and deployment process [3]. This brings the added benefit that new versions of an application can be rolled out by gradually replacing its services one after another. When doing so, it is easy to identify if the new version of a service exhibits any defects, and remove it from production.

In an optimal case microservices are stateless, which means that they can be easily migrated between hosts in a IaaS cloud to consolidate multiple services onto a single VM On the other hand this enables operational staff to easily spawn multiple instances of a service and put them behind a load balancer that simply exposes the same interface as the service itself. This way, the application can be scaled out easily without the user ever noticing that they do not interact with the original service but with a load balancer [34].

However, there are several drawbacks when using micoservices. Savchenko et. al [34] argue that using this architectural pattern does not remove any complexity from the applications, it only relocates it to the infrastructure. Furthermore, the communication between the individual services also introduces additional complexity and accessing the data of different services is only possible through the exposed interfaces. This also implies that tasks that would have been trivial in a monolithic application, like joining data, can become tedious tasks that need to be dealt with. On the subject of data handling, it is important to note that applications which based on a micorservice architecture seldom only rely on traditional relational databases, but often also use some kind of noSQL data storage.

According to the CAP theorem, one has to choose two of the three properties, but cannot have all of them at the same time. These properties are Consistency, Availability, and the tolerance for network Partitioning [7]. Thus it is common for appliactions realizing a microservice architecture to use noSQL datastores that provide BASE (Basically Available, Soft state, Eventual consistency [31]), instead of ACID (Atomicity, Consistency, Isolation, Durability) guarantees [15]. The main reason for not using ACID (Atomicity, Consistency, Isolation, Durability) data-stores, is that availability is often more important than strict consistency. Keeping data consistent across multiple, distributed stores would induce the need for prohibitively expensive transaction mechanisms, like the Two-Phase Commit Protocol.

2.3 DevOps Methodology

When designing an application based on a microservice architecture, deploying and operating all services properly can be a challanging task, especially when the abidance by some quality rules is also a goal. These problems introduce the need for a new set of practices. These practices have to ensure that services adhere to the highest possible

measure of quality, while still enabling to deliver changes to production in a timely fashion. This requirements perfectly capture the essence of the DevOps methodology, as described by Bass et al. [5].

This methodology is a set of practices that aims to bring developers (Dev) and operational staff (Ops) closer together, to build software of higher quality [5]. These practices include making developers responsible for handling possible failures of the application, with the goal of reducing the time until a new version of the failing application can be rolled out, or the old version can be redeployed. Furthermore, Bass et al. [5] argue that operational staff needs to play a key role when defining requirements for applications, so they can, for example, raise their concerns about the usability of log messages.

Another key aspect of DevOps, which comes from its advocacy of Continuous Delivery (CD) [22] to ensure quick and repeatable deploys of services, is the need for the automation of the deployment process [5]. CD can be described as an extention of Continuous Integration (CI) [14]. CI's goal is to automate the process of obtaining a tested artifact from changed source code. To do this, it advocates automated testing (whose result should be visible for everyone involved), dedicated integration servers and commiting changes to a Source Code Management (SCM) as often as possible. As an extension of CI, CD aims to keep the time it needs for a change in the code to make it into production (also called the *"cycle time"* [22]) as small as possible. Automating this process makes it much faster and more reliable than it could be achieved manually. Furthermore, it makes the process repeatable and removes the possibility of human errors, that can easily occur during such a cumbersome task [5]. However special care has to be taken of the code that is used for this automation, as it should be developed with the same rigor as the actual application code.

By having such an automated build process, together with many small, decoupled, microservices, it is possible to employ techniques like a *Canary Release* [5]. A Canary Release happens when a new service is moved from staging to production, but only made available for a selected set of users. This way possible software defects can be detected without affecting the whole user base. Should the service hold up, all user requests are gradually routed to the new version of the service, and the old one is removed, once it is no longer needed [5].

Balalaie et al. [3] argue that employing a microservice architecture facilitates using a DevOps methodology. They present a use case where a monolithic application was migrated to a microservice architecture. Since the resulting services were small, and easily manageable by small teams, it was possible to do what Bass et al. [5] describe as *"breaking down silos"*. This means that developers, operational staff, and members of quality assurance, work together in a single team, which generally leads to better cooperation between them. This way the quality of the software can be improved, by taking the concern of all involved parties into account [5].

CHAPTER 3

State of the Art

3.1 Fog Computing

Currently, there is extensive research going on which aims to close the gap between the edge and the cloud. To achieve this, researchers often describe a *Middleware Orchestration Layer*, which is commonly refered to as a *Fog Layer* [6], because it resides between the edge and the cloud. In many works, it is explicitly pointed out that the goal is not to replace cloud computing with edge computing, but rather to complement its shortcomings and to extend its capabilities to the edge of the network [1, 6, 37]. This way it becomes possible to provide resources that are in spatial proximity to where they are needed, which might be of interest for several broad fields of applications.

As an example for such a use case, Bonomi et al. [6] sketch out the scenario of an autonomous wind farm. In this application of fog computing, embedded devices at the edge collect real-time data from the turbines and react accordingly, for example by changing the tilt of the turbine blades. Not reacting to a change in condition in real-time might damage the turbines, thus fast response times are key. This part of the application alone, would still be sufficiently well-handled by traditional real-time systems [6]. However, users want to make use of the data that the sensors collected and use it for big data analysis. This analysis would not be possible at the edge alone because of the sheer amount of data that needs to be analyzed. Thus, the data is transferred to the cloud where it can be processed at a later time. The collected data can then be used to tweak the algorithms that decide how to react to which conditions.

In their survey Hu et al. [20] describe, what they identified to be, the architectural foundation of fog computing. Namely, a hierarchical three-layer architecture, where each individual layer has vastly different characteristics. At the very bottom of the hierarchy there is the *Terminal Layer* which consists of end-devices or *Smart Objects* that are

comprised of e.g., temperature sensor, card readers, or actuators. As examples for such end-devices the authors mention mobile-phones and smart-cars [20]. The devices collect information about the current state of a physical device and forward it to the next layer in the hierarchy, namely the *Fog Layer.* The connection between the Terminal and the Fog Layer is primarily realized via technologies such as 3G, WiFi, or Bluetooth [20]. The devices in the Fog Layer are generally low-powered, but have enough resources to accomplish simple tasks, such as caching or aggregation and anonymization of data produced by the devices in the layer beneath [20]. Furthermore, they have a connection to the layer above them, which is the cloud. This connection is generally realized via the IP protocol. The ultimate goal is to optimally use the complementary benefits of the cloud and the fog layer. Hu et al. [20] identify computation offloading as one of the means to achieve this. They summarize several approaches towards this task.

Hong et al. [19] present a programming model for applications that use fog computing as a combination of an application model and an Application Programming Interface (API). As a basis for their model, they assume that data-centric applications are split up in a hierarchical way. The parts of the application, which the authors call *Mobile Fog Processes*, are then distributed across edge devices exposing a defined API. These devices are called *Fog Nodes*, which are basically micro-datacenters. Apart from being deployed to the edge, parts of the application can also be deployed to the cloud. Each of the Mobile Fog processes then executes its defined task, which could be reading sensor values and pushing them to the next level of the hierarchy, preprocessing data received from down the hierarchy and forwarding it up, or doing big data analysis. The defined API includes for example the querying of a nodes metadata (e.g., available sensors or actuator, its location ...) or forwarding of a message to a child node. One problem that we see with this approach is that the structure of the application always has to be hierarchically and that this structure implicitly defines the deployment locations of the individual Mobile Fog Processes.

Skarlat et al. [35] provide a more formal approach to optimize resource allocation in the fog, allowing to distribute applications among Fog Nodes. The goal of the optimization is to decrease the latency and cost. To achieve this the authors divide the available infrastructure in an hierarchical way. Although the authors provide a method how one can optimize the workload distribution on the edge, they do not answer the question how one could easily and automatically deploy applications to the edge. Furthermore, they do not propose a solution how the interplay between cloud and edge can be managed.

Another approach to harnessing the power of the cloud on devices with constrained resources is offloading expensive computations like image processing from edge devices to the cloud [4, 8, 10]. The MAUI framework [10] lets users annotate methods to indicate that their execution can be moved to the cloud. This feature however is only implemented for *.NET*[1] applications. One hurdle that has to be overcome stems from the fact that

[1] https://dotnetfoundation.org/

the edge devices' CPUs might have a architecture or a different instruction sets than traditional server machines [10], which prohibits direct execution of the compiled .NET core on both devices. To cope with this issue, Cuervo et al. [10] use the Common Intermediate Language (CIL) to enable execution on servers and edge devices alike. The authors enable migration of annotated methods to a MAUI server that can execute expensive (w.r.t. resource consumption) tasks in the cloud. The problem still remains, that users need to manually annotate remoteable methods (i.e., methods whose execution can be moved to the cloud) and that the solution only works for .NET applications. The problem with hard coding the set of methods, is that their ability to being executed remotely might vary over time. Thus, it would be more desirable to have a mechanism that lets users declare what requirements certain methods have and let an automated system decide if a method should be remotely executed or not.

To mitigate the problem that users have to explicitly annotate methods manually, Chen et al. [9] propose a static code analyzer that scans the static control flow graph, to determine which parts of an application can be offloaded. These are parts that do not use any device specific features like I/O-operations. The authors limit their work to applications that run on the Android operating system. To enable the offloading, they additionally needed to modify the Dalvik VM (the Java VM used on Android smartphones). The problems we see with both approaches stem from the fact that the employed techniques are highly specific to the used technology and prohibits the usage of the technology in a general software environment.

Hung et al. [23] propose a framework that aims to overcome the limited resources of mobile devices, like computational power, storage, and battery lifespan. They achieve this by emulating the user's phone in the cloud and migrating the state of the application to this virtual phone. Once migrated, the actual computation of the application is done in the cloud, while user inputs are simply relayed to the cloud version of the application.

Yigitoglu et al. [40] describe a fog computing framework, that not only aims to provide access to resources at the edge of the network, but also to answer the question how one can optimally distribute services onto available IoT infrastructure. In the context of their framework they refere to the IoT devices as *Nodes* onto which an *Orchestration Client* is deployed. This client corresponds to what Bonomi et al. [6] refer to as a Foglet (i.e., a background service that manages the IoT device) [40]. Furthermore, they derive a simple description model for the resource needs and NFRs of a service, which helps by determining valid target devices for services. Additionally, the authors in [40] put a CI workflow at the core of their framework. Thereby, they aim to automate as much of the deployment process as possible. To enable the individual services to communicate with each other an MQTT broker is used [40]. The individual services are packaged in containers, which enables a high degree of flexibility with regards to the edge device onto which the service is deployed. Furthermore, they advocate a microservice architecture pattern for the applications that are realized using their framework [40]. This architectural style, in combination with a way of declaring the resource needs and

NFRs of the individual services, removes the need to explicitly split the application into functional blocks or define which parts of the application can be run in the cloud and which locally, as done in [19] and [10] respectively.

3.2 Automatic Deployment

Apart from the possibilites and issues fog computing brings, there is also the question, how one can transparentyl deploy services onto available infrastructure. A possible solution is presented by van der Burg and Dolstra [36]. They present *Disnix*, which allows users to declare the existing services, the available infrastructure, and a mapping how the system should distribute the services onto the infrastructure. To enable the declaration of these facts they propose a custom Domain Specific Language (DSL) that allows the integration of common build tools. However, the users have to define the mapping of services to hosts by hand. In order to make hosts available to the system, the user needs to run a setup script and start a daemon which is then responsible for receiving instructions regarding the deployment of new services. The system provides desirable features like a declarative way of specifying the available infrastructure and services, as well as transactional rollouts, where either all services are started or none of them are. However, the user is forced to manually define the mapping of services to hosts, which is something we explicitly want to automate in our framework.

Another approach is described by Matougui and Leriche [29], where they present a constraint-based deployment architecture. In their work, they use a custom DSL (similar to van der Burg and Dolstra [36]), but here the language is only used to declare constraints and attach them to services and not to declare services, infrastructure, and the deployment plan. Possible deployment locations are discovered in the network by a dedicated service, and the administrators of the hosts have to give the deployment system appropriate access right which allows for deployment of services and the installation of software. Additionally, the system includes a hierarchically organized agent system that supervises the deployment process. Should a failure occur, it takes care of propagating this failure and ensures that the deployment process is rolled back. To decide where each service should be deployed to, Matougui and Leriche translate the declarations of the users (which is written in their custom DSL) to a Constraint Satisfaction Problem (CSP), which is used as the input to a specialized program (a solver) that is optimized to compute a solution that satisfies all constraints in an efficient way. If the solver cannot satisfy all constraints, the user is alerted that the model resulting from their declaration and the discovered infrastructure is unsatisfiable. The problem of this approach lies in the expressiveness of the DSL, which is used to define the services and their attached constraints. Furthermore, it lacks the capability to define NFRs for services, that need to be honored when deploying them. We also see the need, to deduce certain constraints (e.g., the set of valid deployment platforms for services) based on the information present. This is not done in [29] since the authors do not consider different deployment platform with varying characteristics.

Gabrielli et al. [16] employ techniques similar to [29] by also using a DSL to specify the requirements of the services and use Zephyrus (a CSP-based planning tool [11]) to determine the optimal deployment configuration of services. However, the DSL they propose is more powerful than the one presented by Matougui and Leriche [29] and they also aim to optimize the resource consumption of the deployment plan. As a basis of their work they assume a microservice architecture and the deployment to an IaaS cloud such as Amazon EC2. In their work the authors also bring up the problem of changing runtime behavior of services which poses the need for a replanning of the deployment, which happens based on the already available deployment plan that is improved by the system to meet the changed requirements. Although, the taken approach enables the users to define a rich set of requirements for their services, Gabrielli et al. [16] do not take into account the services' NFRs that might constrain their possible deployment locations. We try to improve this problem by automatically deriving certain constraints from the users' service definition.

Yigitoglu et al. [40] use a data-centric approach to guide the search for a possible deployment location. The authors start by placing a service onto the node that is closest to the source of the data that it consumes (e.g., a surveillance camera in the case of a service that realizes facial recognition). If this node is unable to handle the service, e.g. because it does not have enough computing power, the next closest is tried. Once the service has been placed, the next service, which consumes data from the previously deployed service, is deployed.

3.3 Runtime Monitoring and Adaptation

Once the users' services are deployed, it is important to monitor their runtime behavior and take according actions to improve the resource usage of the overall system. To prevent over- or under-provisioning of cloud resources and to reduce the amount of SLA violations, several approaches are presented in the literature.

To monitor cloud applications and transform the raw metrics obtained from the applications and hosts, Emeakaroha et al. [12] use a monitoring agent in place at each host that collects infrastructure metrics and delivers them to an aggregator. This aggregator collects the metrics from a variety of hosts and maps them to high level SLAs, according to predefined mapping rules. The computed values are then used to predict possible SLA violations and proactively take measures to prevent them. The problem we see with this approach is that it does not take into account application-specific metrics that might provide an insight into the applications' runtime behavior, helping developers and operational staff to detect possible defects in their applications.

Zabolotnyi et al. [41] present JCloudScale, an event-based framework for scaling Java applications in an IaaS cloud transparently, by abstracting the underlying (virtualized) infrastructure. The framework operates on user-defined scaling policies, which are defined by extending a certain abstract class provided by the framework. To execute the users'

applications, a special JCloudScale server component has to be in place, which receives *Cloud Objects*, which represent a runnable user application. The decision when to apply a certain scaling strategy is declared by defining CEP rules, which are evaluated and actions are taken accordingly.

Huber et al. [21] present an approach for dynamic runtime adaptation of software systems based on QoS aspects of services. They propose a technique that draws a clear line between the logic of the system itself and the implementation of the runtime adaptation. This way the adaption mechanism becomes generic and reusable and is no longer bound to an individual software system. To achieve this goal, the authors devised a meta model for runtime adaptation which consists of *Strategies*, *Tactics*, and *Actions* [21]. Strategies are high level description of what needs to be done to achieve certain objectives. An example for such an objective would be to minimize the costs for a service provider. To realize this, a strategy is equipped with one or more tactics, in this case one tactic could be to switch to utilize cheaper resources or to remove unused resources. A tactic can be realized in multiple ways (e.g., utilizing cheaper resources can either mean switching to another cloud provider, or using resources with looser SLA guarantees). To define the concrete realization of a tactic an action is used. An action contains the steps that are needed to be taken in order to achieve the desired goal. The execution of strategies can either be triggered by an event (e.g., overall costs exceeding a certain threshold) or by a scheduled timer, so that they are executed repeatedly. To determine which tactic should be applied when using a certain strategy the system calculates the tactics are ranked by their presumed impact. The highest ranking tactic is then applied. To execute it, tactics are made up from actions, which are the actual steps taken to realize it. For example to migrate a service, first the new version needs to be started somewhere, its start-up must be announced, and the old service needs to be shutdown.

Apart from increasing or decreasing the available resources in the cloud, Chen et al. [8] propose a workflow-based approach towards runtime adaptation. The authors use a probabilistic approach based on the workflow of composite services (i.e., services that are made up of a number of sub-services). They aim to proactively reroute requests to different services, should the system determine that an SLA violation is likely to occur. This enables the system to use the available resources in an optimized fashion (w.r.t. the defined SLAs), rather than provisioning new ones. The framework uses the locally available information about the system state, along with an automatically constructed model based on the workflow between the individual components. Based on the available information, it chooses the optimal path of execution to prohibit or minimize SLA violations and QoS deterioration.

3.4 Summary

Although there are several works about fog and edge computing which identify key requirements of an orchestration middleware layer that enables harnessing the computational

resources at the edge of network and combining them with the power of the cloud [6], most of them only present a sound theoretical foundation for solving the problem. These key requirements include an abstraction of the underlying edge devices to allow uniform access to their resources and potentially offloading resource intensive computation to the cloud, enabling users to have services that are able to meet narrow time constraints but can also handle large amounts of data at the same time. Furthermore, there is a lot of work done in the field of optimizing resource usage, both in cloud and edge environments. The goals are manifold, and might include the reduction of round-trip-times for requests, the optimal compliance with SLAs, the minimization of costs for users, or the extension of an edge device's battery lifespan. Most of the presented approaches include a monitoring infrastructure that allows the systems to reason about which actions it should take to meet the desired goals. An open question in these works however, remains how the deployment location of the individual services is determined.

To answer these questions, researcher have determined several different approaches in the field of automatic deployment [2, 29]. The main goal here is to free users of the burden of manually rolling out their services. This becomes especially important when using microservices as an architectural pattern, because the number of services that need to be deployed tends to grow rapidly with the complexity of the application. There are efforts to automatically determine where the applications should be deployed to by defining constraints that the target hosts have to fulfill. However, once applications are deployed, it is desirable to adapt their behavior at runtime, which the presented works do not consider. Although Gabrielli et al. [16] consider this problem, their proposed solution still needs manual involvement of operation staff to adapt the services' requirements and trigger a redeployment of services.

In the field of runtime adaptation, there are several approaches how one can react to changing runtime behavior of software systems. Huber et al. [21] argue that the separation of the adaption mechanism and strategies from the system itself is a key factor for creating reusable adaptation mechanism, while Chen et al. [8] use a probabilistic, workflow-based model to make decisions about which service instance to invoke, based on locally available information.

Because automatic deployment and runtime adaptation are mostly treated as separate concerns, we see the need to bring together automatic deployment techniques with runtime adaption mechanism. We identified a fog computing scenario, especially in an industrial environment, as a viable context for our framework. In our opinion, it is desirable not only to roll out services in an automated fashion, but to aid the user in maintaining a healthy system state. This can be achieved by allowing them to define the NFRs their services have to meet, thereby limiting the possibilities of target hosts. Furthermore, users should be able to define rules that constitute a trend towards an undesirable system state and which action the system should take in order to continue to function properly. This means that the services are in place, they have to be monitored, and the result of the monitoring has to be analyzed to decide if actions need to be taken.

CHAPTER 4

The DDAD Framework

In this chapter, we present the main outcome of our work. We discuss the key design decision that were made when creating the Data-Driven Automatic Deployment (DDAD) framework. Furthermore, we introduce its main components along with an architectural overview. Together with the architectural overview we give an insight into how the individual components communicate with each other and which information they exchange. We also discuss how the framework can be integrated into a DevOps workflow to enable continuous delivery when using a microservice pattern and edge computing.

4.1 Requirements

In this section, we discuss the key requirements of our framework. We examine the use case presented in Section 1.1 to determine what functionality the framework has to provide to its users.

4.1.1 Abstraction of Heterogeneous Edge Devices

In the literature, the abstraction of heterogeneous edge devices is a well-discussed problem [6, 19]. The problem one faces when trying to integrate edge with cloud computing is that the underlying devices at the edge are in general rather heterogeneous and differ in nature [17, 20, 38], and that the users should need to know as little as possible about their specific properties. Generally this is solved by forcing all devices that are able to run user services to expose a uniform interface, which facilitates the interaction with these devices. To facilitate this, authors advocate a small service (w.r.t. resource consumption) residing on each device, which exposes this interface [6, 38]. This service we will refer to as the *Device Manager* (Section 4.3.1). However, in the literature it is also sometimes called a *Foglet* [6]. The extent of this interface varies from paper to paper [6, 19, 40], but we identify a small set of capabilities, which the interface has to

expose at a bare minimum. These capabilities are: deploying and starting user services; pausing and undeploying user services; as well as being able to receive information about the system's state.

We choose a minimal set of capabilities for a number of reasons. First, we wanted to expose computational resources, which needed to be made available to users somehow. This can be achieved by allowing them to deploy services onto the edge devices. Once a service is running on a device, it should be able to contact other services, for which the Device Manager needs to obtain information about the system's state (i.e., which services are available and where). When a service is overloading an edge device, or the framework found a better deployment location for it, it needs to be either paused or completely undeployed respectively. Secondly, we wanted to only expose computatoinal resources, in contrast to Hong et al. [19] and Bonomi et al. [6], where also sensing and manipulating abilities of edge devices are exposed. This stems from the fact, that the envisioned context for our framework is within a production site, where sensing and manipulation is done by purpose-built devices. Another reason, for choosing such a minimal API was that the edge devices would proactively register themselves and announce their available resources. Lastly, since we are continuously gathering information about the devices resource consumption which allows us derive the available resources we are eliminating the need to making it queryable, which is done in [19].

To keep the interface small and simple, we choose two commands that the Device Manager can receive from the framework and in response to which it needs to act accordingly, similar to [38]. These commands are the *Deploy Command* and the *Service Update Command* (see Section 4.3.1 for examples of how they look like). The former contains a list of services that are expected to run on the device. With this information it is possible to check which services are already running and determine which need to be started and which need to be shut down. We choose to disallow multiple service instances of the same service to run on the same edge device, because this does not bring any benefits in terms of scalability or resilience. Scalability is not improved since the services are still limited by the available resources. Neither is resilience, because if the device fails, both services fail. However, it should be noted that it would be sensible to deploy multiple instances of a service to a single host, if the service has long blocking operations, which do not incur significant load onto the CPU. Because multiple instances of a single service on one edge device do not yield a significant benefit, we deem it desirable to distribute the instances of the same service across multiple hosts.

The Service Update Command contains information about concrete instances of certain service types and at which endpoints they are reachable. When the Device Manager receives this information, it needs to forward it to all services running on the edge device it manages. This implies that the individual, user-defined services also need to realize a well-defined interface to receive this information. Once the Device Manager has forwarded the information about the concrete service instance, the individual services might need to change the service instance they are currently invoking.

An important type of information that needs to be obtained from the edge devices is what capabilities and resources they offer [19]. In a general edge computing scenario, these capabilities could include sensor or actuators attached to the devices. Hong et al. [19] propose an extensive API for edge devices which allows obtaining detailed information about their capabilities and resources. They also decide to organize the edge in a hierarchical way, which introduces the need for a communication mechanism based on the present hierarchy. In contrast to that, we obtain the information about which resources are available at the devices by letting the Device Manager announce them upon its start-up. We choose this approach, because we assume that the overall resources available on a device do not change drastically over time. To determine which resources a device has to offer, we continuously collect runtime data of the device and its services, thereby enabling the computation of the utilizable resources.

Furthermore, once the Device Manager has received received information about the services that are expected to run on the device it manages, it needs to be able to obtain the executables of the services. Together with these executables, it needs to know how the services are started and what their dependencies to other services are. Additionally, it is desirable to have a mechanism that gives services the opportunity to shut down gracefully within a certain period of time. Should they not be able to do so they need to be shut down forcefully by the managing entity.

To facilitate the runtime adaptation of services that have been deployed to edge devices (Section 4.1.4), we need to be able to activate and deactivate certain service instances dynamically at runtime. To achieve this, we first need to supply the Device Manager with a list of instances that services running on the device they manage can invoke; as well as a list of services that are expected to run on the respective device.

4.1.2 Finding Deployment Strategies in Cloud-Edge Scenarios

Once the edge devices are made available to the user, the question remains, how to plan the distribution of a set of services onto these devices. This needs to be done, while adhering to all specified NFRs, providing the required soft- and hardware, and utilizing edge resources in an optimized way. Forcing the users to manually decide where each service should run is not a viable option, since with a growing number of devices and services this task becomes cumbersome and error-prone [36].

Furthermore, the system should be able to derive certain constraints concerning the deployment location of services, based on the services' definitions. This way, the users can define the services without needing to concern themselves with defining basic constraints manually. Examples for such constraints would be the platform to which services can be deployed based on their need for (near-)real-time communication, privacy concerns associated with the data they produce or use, which can be fulfilled by edge devices [1, 6, 20]. Another example for such constraints would be the need for handling large amounts of data or providing enormous computational resources, which can only be achieved in

the cloud [6]. Additionally, when employing new platforms, and replacing or altering the properties of existing ones, the allowed deployment locations might change.

Without a method to automatically derive feasible platforms for the services' deployment, users would have to keep the manually defined constraints up to date, when altering defined platforms, which again, is an error-prone and cumbersome task.

As we argue later in Section 4.2.5, we integrate the DDAD framework into a DevOps workflow which is liekly to result in very frequent deployments [5]. This gives the framework the possibility to determine a new and updated strategy for each of these deployment events. Therefore, it is even more desirable to automate this process, to further optimize the service distribution across the available devices. This becomes even more apparent when we take into account that we apply runtime monitoring (Section 4.1.3) to observe runtime behavior which might differ from the assumed runtime behavior of a service, possibly leading to a correction in the hardware requirements of a service. Such a correction, as well as newly added edge devices, can lead to different and better deployment scenarios.

4.1.3 Runtime Monitoring of IoT Applications

Since monitoring is a key part of the DDAD framework, users need to be able to gather different metrics independently of the service and the device it is running on. The need for monitoring the services in place, stems from the fact that the collected metrics provide insight into the application's actual runtime behavior, which may very well differ from the anticipated one. Furthermore, it enables the prediction of imminent QoS or SLA violations, as well as the possibility to detect a movement of the system towards an undesirable state.

There are two distinct kinds of metrics that need to be collected to enable the users to analyze their system's runtime behavior, because each of them allows insight into different aspects of the runtime behavior and can influence different actions.

First, there are *device metrics*, that are independent of any service running on an edge device. They give an overview of the device's state and to how much of its capacity it is working. This is an important aspect when deciding how much more services can be deployed to a device or if some of the services running on it need to be paused or undeployed. These metrics might include the current, total CPU load of the device, the overall amount of memory used, or the temperature of the device's CPU. Since these metrics are not associated with any particular service, they need to be collected by a process running in the background of every device.

Secondly, there are metrics that are service specific. They give insight into how an individual service behaves, which can be used for example to decide which service on an overloaded device should be shut down. Another use of these metrics is the adjustment of the service's resource needs which can yield better strategies for future deployments,

since the data on which the planning is than based more accurately reflects reality. These service metrics can be put into two distinct categories, *generic service metrics* and *application-specific service metrics*. The first category would be the CPU load of an individual service or how it long it is running uninterrupted which can be obtained by the Device Manager. Examples for application-specific service metrics include the length of queue of work items, the execution time of certain methods, or the average number of requests during a defined interval. These metrics need to be collected by the services themselves, since no other service can obtain this kind of information.

To make use of the collected metrics, there is the need to aggregate them. The metrics then need to be stored for further analysis. To facilitate this analysis, the user needs to have a possibility to visualize the collected metrics in a meaningful and easily understandable way.

4.1.4 Runtime Adaptation of IoT Applications

Once all services have been deployed according to the calculated deployment plan and with the monitoring mechanism in place, the next step is to make use of the collected metrics by analyzing them and adapting the system accordingly. This needs to be done, because services might exhibit runtime behavior that differs from the expected one. Thus, when a movement towards an undesirable system state is detected, it is highly likely that users wants that some action is taken. This needs to happen automatically, since users cannot and do not want to observe the runtime behavior of their system permanently. One of the main concerns is, that the devices become overloaded by the workload introduced by the user services, which might cause an interference with the devices' primary tasks.

The first step in detecting the movement towards such an undesirable state (i.e., a state where certain user-defined SLAs and/or QoS parameters are violated), is to enable the users to define which event, or chain of events, indicates this movement. When such a movement is detected, action has to be taken. Thus, users need to be able to define such actions which are executed in response to a certain event or chain of events. The executed actions will generally result in an update of the system state. This way, user-configured runtime adaptation is facilitated.

Furthermore, it would be desirable to automatically detect immanent SLA violations in an effort to even further automate and optimize the workflow of the framework. Nevertheless, we see this automatic detection as out of scope for our work. There exists research that aims to answer this question [24, 26]. Therein, the authors train a prediction model with machine learning techniques. This model is then applied to the current runtime data and can predict immanent SLA violations in composite systems (i.e., systems comprised of multiple services).

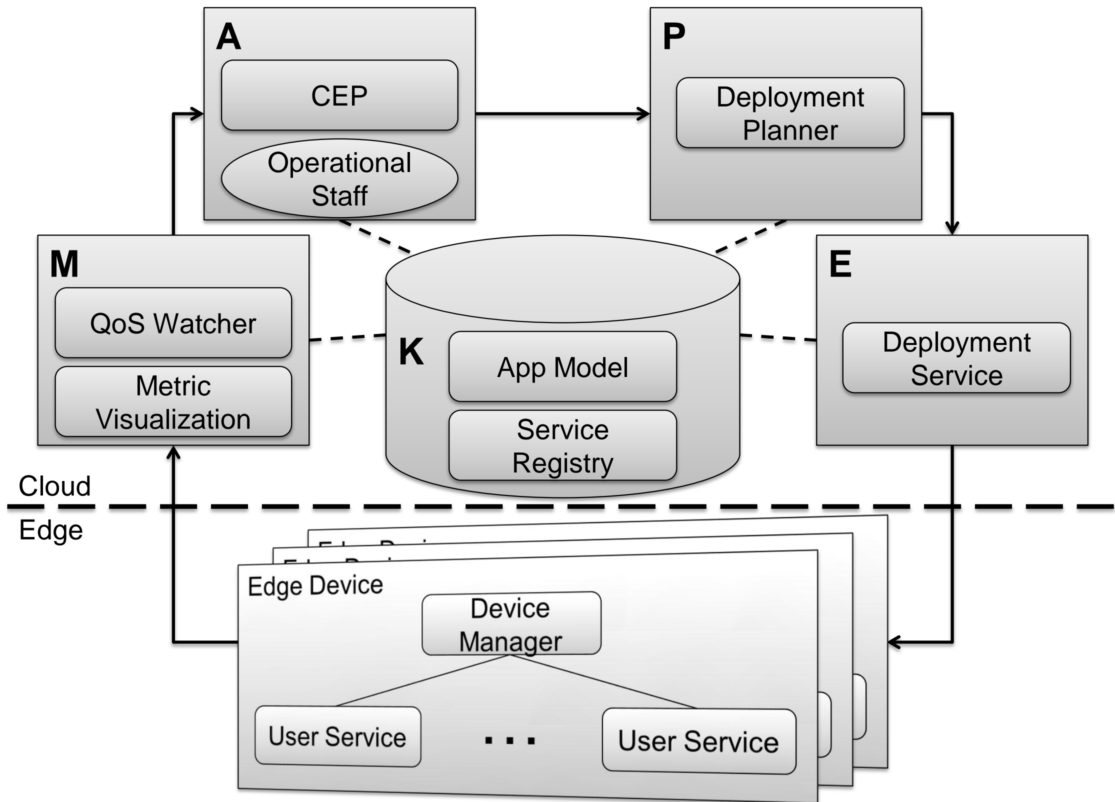

Figure 4.1: Architecture of the Framework

4.2 Key Design Decisions

4.2.1 Realization of a MAPE-K Cycle

Figure 4.1 shows the basic architecture of our system and how each of the components fits into the MAPE-K (Monitor-Analyze-Plan-Execute on a shared Knowledge Base) cycle [25]. This cycle is is a fundamental model of autonomic computing. The idea behind it is that a system components that realize the four stages and access a shared knowledge base to do so. Thereby, the system is enabled to manage itself [25].

Since the DDAD framework realizes self-configuration, by only deploying services in a way that fulfills all NFRs and satisfies all software dependencies and hardware needs of the services, taking into account the capabilities of the edge devices. Furthermore, the DDAD framework strives for self-optimization, as the Deployment Planner tries to optimally use the available resource and an undesirable system state might trigger a redeployment or an adaptation which might change the set of available services at individual edge devices. Therefore, the framework is an optimal candidate to realize a MAPE-K cycle.

In the DDAD framework, the operational staff and/or the CEP component realize the *Analyze* part of the cycle, as they observe the current state of the system and take actions accordingly. They obtain the needed information from the *Monitor* component, which is comprised of the QoS Watcher and some kind of Metric Visualization. The *Planning* part of the MAPE-K cycle is realized by the Deployment Planner, which queries the available knowledge and determines a legal and optimized service distribution among the available devices. Lastly, the Deployment Service realizes the *Execute* part, as it performs the actual deployment of the individual services. The shared *Knowledge Base* is realized by the Service Registry and the App Model. The latter is a service that holds information about the current state of the system and uses a knowledge graph to store this information (e.g., which platform has which characteristics and which hosts reside there) and is discussed in greater detail in Section 4.3.2.

4.2.2 Push-Based, Autonomous Monitoring

To realize the need for continuous monitoring of the individual edge devices and the services running on them we decide that we need a dedicated service running in the background at each device which collects these metrics. Since we already need such a service to abstract the underlying edge devices enabling the users to run services on them, we extend its functionality, to also take care of monitoring the devices and their services.

Another possibility would be to create a new service which takes over the duty of collecting system information about the edge device, as done in [38]. However, we want to be able to not only collect device metrics which are independent of any concrete service, but also service metrics (see Section 4.1.3). These tasks are realized by the Device Manager (Section 4.3.1). We choose to combine both duties, to remove the need for communication between the service that handles the services' lifecycle and the one that is in charge of monitoring the services.

Thus, the Device Manager is not only in charge of handling the services' lifecycles on the edge device, but also of collecting device metrics and generic service metrics. It collects a predefined set of metrics, buffers them locally, and pushes them to the cloud. In the concrete case of the prototypical implementation of the DDAD framework this predefined set consists of the following metrics: the device's CPU load and memory consumption, for the category of device metrics. As well as each service's CPU load for the category of generic service metrics. We choose such a minimal set for the prototypical implementation of the framework, because this small set of metrics, in combination with application-specific service metrics, already enables us to realize a basic adaptation scenario as shown later in Section 5.

Apart from actively collecting these metrics, the Device Manager also receives application-specific service metrics from the running services. It also buffers them and sends them to the cloud. By not sending every measurement individually, we aim to reduce the overhead that is needed to transfer the data.

We choose a push based approach for our monitoring mechanism, because this way the Device Manager does not need to care about which services are providing which application-specific service metrics and in which interval it needs to poll them. This reduces the complexity and the range of duties of the Device Manager. Additionally, the services need to gather application-specific data anyway because the Device Manager has no possibility to obtain them, other than from the services themselves. The storage of the data happens in the cloud because this allows us to scale the services that handle the metrics based on the number of currently running services.

To deliver the metrics, the Device Manager opens a TCP/IP socket to which the services need to push their collected metrics. This happens via a simple JSON-based protocol, to incur minimal overhead. We did not use a message-based protocol like STOMP[1] or MQTT[2] because this would have induced the need to also run a message broker which would introduce too much overhead for simply streaming messages to an aggregator. Having the metric storage and handling located in the cloud removes the need for determining which metric needs to be forwarded to which edge device to handle it. Furthermore, it removes the need for storing the data in a distributed fashion, since it is unlikely that there is an edge device that can hold the metrics of all running devices.

A drawback that comes from using a push-based approach, is that it is not possible to determine if an edge device, or its Device Manager, has stopped to function properly or if there are simply no metrics to deliver. However, this problem could be easily solved by sending heartbeat messages in a predefined interval, to indicate that the Device Manager is still up.

4.2.3 Modeling the Deployment Planning as a Constraint Satisfaction Problem

Since we aim to provide the users with a method to automatically deploy their services in an optimized fashion, there is a need to determine how to distribute a set of services onto a set of devices. This needs to be done while adhering to all specified NFRs, providing the required soft- and hardware, and properly using the resources available at the edge. Manually deciding for all services where they should be deployed to is a cumbersome and error-prone task, that becomes infeasible as the number of services, devices, and NFRs grows [16, 29].

The problem of properly distributing the services onto devices while not exceeding the devices' resources is a variant of the Multidimensional Knapsack Problem, which is known to be NP-hard [32], which means for a general case there is no efficient (w.r.t. time), deterministic algorithm to solve it. Thus, using a naive algorithm to determine the mapping is not an option. Since we also want to be able to impose additional constraints upon or deployment plan, like the need for devices to provide the appropriate software in

[1] https://stomp.github.io/
[2] http://mqtt.org/

a compatible version for the services, we formulate the problem as a CSP. We then use a specialized program (a solver) to obtain its solution, similar to the approach taken in [16, 29].

We see the need to have a parameterizable problem that captures constraints that need to be met for the deployment plan to be regarded as valid. Then, every time a deployment is planned, the needed parameters are created and the solver can be used to solve the concrete instance of the problem. This facilitates updating the problem dynamically, without the need to change the service that invokes the solver and does the translation to and from a representation the solver can understand.

4.2.4 Microservice Architecture Pattern

We decide that all services, except for the Device Manager, should run on a PaaS cloud, based on the assumption that they need more resource than edge devices could provide. Bonomi et al. [6] argue that the goal of utilizing edge resources it not to replace cloud computing, but rather to complement its capabilities. Thus, the appropriate platform for each service has to be chosen, which generally results in services that handle large amounts of data and require extensive computational resources to remain in the cloud. Furthermore, deploying the parts of the framework to the cloud also enables us to easily scale them in and out, based on their resource demand. Furthermore, deploying the services to a PaaS cloud relieves us from the burden of manually providing commodity services like databases and storage, as well as from having to care about the underlying infrastructure and setting up the runtimes environments for the individual services.

Additionally, when running services on the edge they can be interrupted and deactivated if the current workload of the edge device does not allow the execution. This stems from the assumption we made in Section 1, that edge devices only run user services as a secondary task which must not interfere with their primary task, thereby making the edge an inappropriate platform to run our framework on [37]. Also, it is undesirable to have a service that aims to solve a CSP within a reasonable time-frame to run on a low-powered device.

Since different services of our framework might exhibit vastly different resource needs, we determined that the best way to cope with this, is to use a microservice architecture [5]. This way, we can scale all services independently [5]. It also enables us to use the tool best-suited for each task, and decouples the individual components from each other [5]. This facilitates to update services without affecting other ones, as long as an agreed upon interface is preserved [5].

Another reason why we choose a microservice architecture for the DDAD framework, is that the applications that are being deployed by it, should also employ such an architecture. This way, the runtime adaptation mechanism we envision is facilitated, by splitting the services' functionality up, enabling a relatively fine-grained (re)distribution

of services across the devices. Since, the microservice pattern was shown to be suitable for large-scale distributed applications [3], employing it removes the need for partitioning applications based on their capabilities explicitly, as done in [4, 10].

4.2.5 Integration into a DevOps Workflow

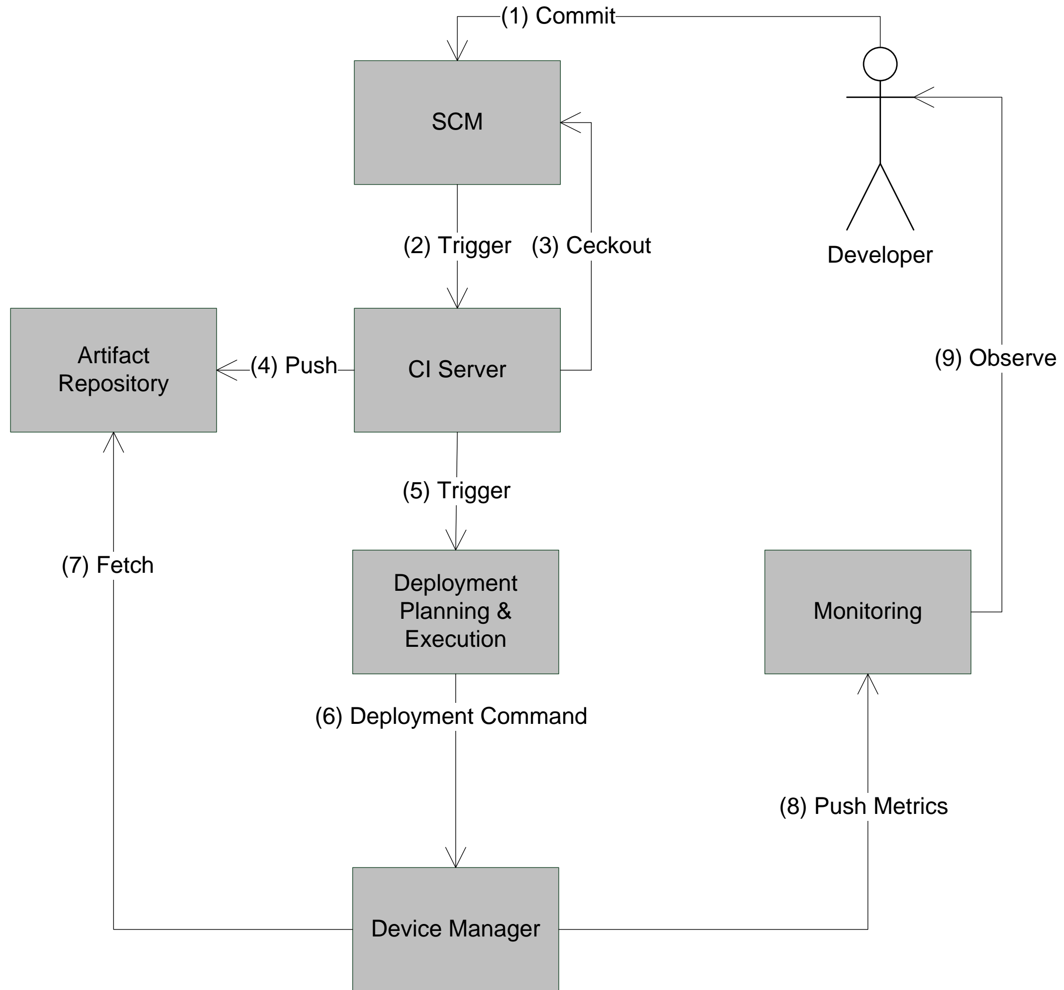

Figure 4.2: Integration of the DDAD Framework into a DevOps Workflow

As we argued in Section 4.2.4, the intended scenario for our framework is the deployment of applications, whose architecture is based on the microservice pattern. Since Bass et al. [5] and Balalaie et al. [3] argue that developing and operating applications that follow this pattern is an exemplary use case for DevOps, we aimed, that our framework is easily integrated into a DevOps workflow. Figure 4.2 shows how this integration can be achieved. When using this methodology, there are eight steps that are continuously repeated [5],

which are: *Planning*, *Coding*, *Building*, *Testing*, *Releasing*, *Deploying*, *Operating*, and *Monitoring*.

First, there is the *Planning Step*, during which DevOps engineers plan their next steps, and how to implement the services or how to incorporate desired changes. Then, during the *Coding Step* developers write the production code. They then commit their changes into some SCM like Git[3], which is modeled by (1) in Figure 4.2. This action triggers the next step in the DevOps cycle, namely the *Building Step*.

In general the commit triggers a CI server like Jenkins[4] (2) which checks out the newly committed code (3) and starts a new build. This build is followed by a *Testing Step*, which might include unit-, integration-, and UI-tests. Should the new code pass all tests, the resulting artifacts are transferred to an Artifact Repository (4), thereby realizing the *Releasing Step*. Up until now, the system followed a generic DevOps workflow [5], but now our framework comes into the pictures. After the artifacts have been transferred to some repository the CI server calls the Deployment Service (5). Thereafter, the planning of the deployment proceeds as described in Section 4.5.2 starts. When a deployment plan is determined, the system sends the appropriate messages to all affected devices (6). All Device Managers the receive a message then fetch the artifacts from the Artifact Repository (7), which corresponds to the *Deploying Step*. Next, they start the corresponding services (this realizes the *Operating Step* step), execute the *Monitoring Step* (i.e., begins monitoring the services), and continuously pushe metrics to the Monitoring component (8). The information gathered from monitoring the system is then used in the *Planning Step*, to decide what needs to be done, thereby closing the circle.

4.3 Main Components

4.3.1 Device Manager

Based on the need of abstracting heterogeneous edge devices and the need for distributed monitoring of IoT applications, we propose a service that corresponds to what Bonomi et al. [6] call a *Foglet* and to which we refer as the *Device Manager* (see Section 4.1.1). Its key requirements are having a small resource footprint (w.r.t. memory consumption and CPU load), being able to run on a wide variety of devices, and enabling communication through a uniform interface.

Although, different authors describe different APIs which an edge device should expose via its manager [6, 19, 38], we determine that at a minimum the interface must provide the possibility to deploy and undeploy services, as well as to inform the device about changes in the system state (see Section 4.1.1). How these changes in system state are handled, is then left to the services themselves. The Device Manager only has the duty of delivering the information to the services.

[3] https://git-scm.com/

[4] https://jenkins.io/

We provide two different ways for external services to contact the Device Manager (i.e., services that want to send any of the commands defined in Section 4.1.1). One way of contacting the Device Manager is by invoking REST endpoints that correspond to the specific commands and which can be used if the Device Manager is directly reachable for a service that needs to communicate with it. This approach is chosen because libraries and frameworks that help exposing an HTTP based interface are available for basically every programming language, and using a generic communication interface hides the actual implementation of the Device Manager from the components communicating with it, making it easily interchangeable [5].

The other way of communication is via a message queue, which can be used if inbound traffic to the Device Manager is not possible. This scenario might for example happen, when the edge device is located within a production plant, where incoming traffic might be blocked by a firewall due to security considerations. Using a message queue however, implies that the queue has to be reachable for all services that want to communicate over it, thus outbound traffic for the Device Manager must be allowed, otherwise it cannot check for the arrival of new messages. Both the REST endpoints and the messaging protocol use the same message structure so that commands can be handled without considering the way they arrived.

Listing 4.1: Structure of a Deploy Command Message

```
{
  "instances": [
    {
      "id": "string",
      "serviceName": "string",
      "artifactUrl": "string"
    }
  ]
}
```

Listing 4.2: Structure of a Service Update Command Message

```
{
  "serviceName" : "string",
  "available" : [
    {
      "id": "string",
      "protocol" : "HTTP|MQ",
      "endpoint" : "string"
    }
  ]
}
```

Listing 4.1 and 4.2 show the structure of a message containing a Deploy Command or a Service Update Command, respectively. In the context of these commands the *serviceName* parameter specifies the type of service (e.g., *scoring-service*) and *id* specifies the concrete implementation of a service should more than one implementation of a service exist. However, all implementations have to adhere to the same interface, which needs to be guaranteed so the registry-aware service client can transparently invoke any of the implementations. The *artifactUrl* property specifies where the Device Manager can find the artifact for a certain service which the manager needs to download, unpack, and execute in order to start a service instance. The *protocol* and *endpoint* property specify how one service can reach a concrete instance of another service which is already running (e.g., HTTP, message queues, websockets, . . .).

To actually start a service, the Device Manager needs to know how it can obtain the service's executable. This is done by providing an Unique Resource Locator (URL) to an artifact repository, which contains an archived version of the executable. How to start a service can depend on multiple factors, like the programming language it was written in. Furthermore, the Device Manager needs to know which other services the started service depends on, so it can propagate changes in the system state to the right services. To transfer this information, each service needs to be packaged with a *metadata file*. This file closely resembles the *manifest file* used to deploy services to CloudFoundry[5]. An example of such a file can be seen in 4.3

Listing 4.3: Example For a Metadata File

```
service:
  id: service-id
  endpoint: http://some-endpoint.com/
  protocol: [HTTP|SQS]
  services:
    - other-service-name-1
    - other-serivce-name-2
```

Because there is a need to announce the capabilities of an edge device, the Device Manager analyses the available resources of the system it runs on, once it was started. When this analysis is done, it pushes the obtained information to the cloud, where it is stored by the appropriate service. This also means, that when a new device is added to the set of available devices, its information is automatically published. Thus, there is no need for device discovery at the edge.

Figure 4.3 shows the components that we use to realize the desired functionality of the Device Manager. The *Command Handler* receives instructions from the cloud to deploy and undeploy services, as well as updates of the available services. It forwards this information to the *Lifecycle Manager* which is in charge of starting and stopping

[5]https://www.cloudfoundry.org/

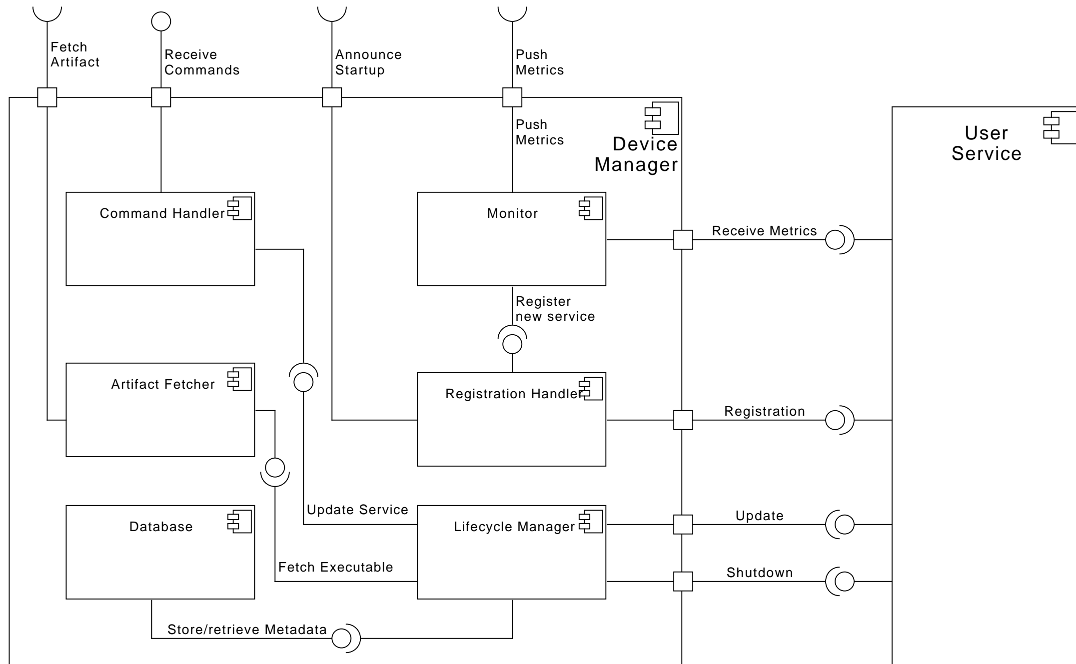

Figure 4.3: Component Diagram of the Device Manager

services, as well as forwarding updates of available services to the individual user services. Furthermore, it stores metadata about all running services in an in memory database. This database contains information about the type of service, which other services they use, and at which endpoint they are reachable locally. As discussed above, the Device Manager needs to obtain the executable of the service, which is achieved by the *Artifact Fetcher*. This component contacts a cloud storage and downloads the respective artifact.

Once the service is started, it needs to register itself at the Device Manager. The *Registration Handler* receives information about the services' startup and instructs the *Monitor* to start gathering metrics about the services. The Monitor does this by opening a TCP/IP socket on a random port which is unique to each service. The services then start start pushing their application-specific service metrics over this socket. Furthermore, the Monitor starts monitoring the resource consumption of the services' processes. It buffers this data and pushes it to the cloud, where it is used for further analysis of the runtime behavior of the individual services.

4.3.2 App Model & Service Registry

In order to properly capture the different types of services and their requirements, we use a data model that lets users define components, which in turn can either be self-contained, part of other components, or have other components as their components. An example

would be a software library, which can be comprised of a multitude of other libraries and can be used by a service to realize a certain functionality. In the model, we differentiate between a static and a dynamic view. The static view is based on the definition of services, platforms, and hosts. These definitions not only include their basic properties like name and endpoint for hosts, or the name and the library dependencies for services. The static view also includes the declaration of the NFRs the services need, and the ones that the platforms provide.

Listing 4.4: Static Definition of a Service in the App Model

```
{
  "id": "9fbd8721-4372-473e-a799-6373074dae49",
  "instanceOf": "SampleService",
  "serviceName": "ExampleService",
  "artifactUrl": "http://somestorage.io/example-service.zip",
  "resources": {
    "CPU": 600,
    "RAM": 600,
    "BANDWIDTH": 20
  },
  "usageParameters": [
    "ElasticScalability"
  ],
  "services": [],
  "software": [
    {
      "software": {
        "name": "Python",
        "version": 3.5,
        "type": "Language"
      },
      "relation": "AT_LEAST"
    },
    {
      "software": {
        "name": "cherrypy",
        "version": 3.8,
        "type": "Framework"
      },
      "relation": "EXACTLY"
    }
  ]
}
```

Listing 4.4 shows the static definition of an exemplary service. The *instanceOf* property defines the type of the service, which enables multiple implementations of service co-existing and being transparently invoked, based on the current needs of the calling services. To uniquely identify the service the *id* or the more human-readable *serviceName*

property can be used. To declare where the Device Manager can obtain the artifact which contains the service's executables the *artifactUrl* property needs to be defined by the user. The *resources* and *usageParameters* are used to decide onto which host on which platform the service can be deployed. To declare dependencies to other services the *services* list can be used, for which the user supplies the type of the service on which the services that is being defined depends. Lastly, the user needs to define which software the service needs to function. This is achieved by establishing a relation between an existing instance of a software with a certain version (e.g., the Python programming language in version 3.5) and the type of relation (i.e., is it the minimum, maximum, or exact version of the software that the service needs).

Listing 4.5: Example Definition of an Edge Device

```
{
  "id": "7d67fe63-3ef0-4142-b384-0c5b115ed0b7",
  "hostname": "edge-device"
  "queueId": "79e9e34bc9cdf6deab9437b441341e26",
  "endpoint": null,
  "resources": {
    "CPU": 2400.0,
    "RAM": 1073332224,
    "BANDWIDTH": 100
  },
  "software": [
    {
      "name": "cherrypy",
      "version": 3.8,
      "type": "Framework"
    },
    {
      "name": "psutil",
      "version": 5.2,
      "type": "Library"
    },
    {
      "name": "python",
      "version": 3.5,
      "type": "Language"
    },
  ],
  "platform": {
    "name": "Edge",
  }
}
```

Listing 4.5 shows the static definition of an edge device. When defining a host the user also needs to supply a unique name for the host, and the system will assign a unique id to it. To enable communication with the host either the *queueId*, the *endpoint*, or both

properties have to be defined. The next property, namely the *resources* property, does not have to be supplied by the user, since the device manager will obtain information about the resources available at the edge device upon startup and inform the App Model about it. However, the automatic detection of installed software and software packages is not part of the Device Manager's functionality, since the process of reliably detecting installed software and also installed libraries for available programming language runtimes is considered out of scope for this work. Thus, the user has to manually insert relations between hosts and software packages, similar to the definition of services. However, since the notion of having a maximum or minimum version installed is not very sensible in this context all relations specify the exact version of the software package.

Listing 4.6: Example Definition of an Edge Device

```
{
  "id": "964d3399-1ed5-4839-8f23-0a70d65c4338",
  "serviceId": "9fbd8721-4372-473e-a799-6373074dae49",
  "runsAt": "7d67fe63-3ef0-4142-b384-0c5b115ed0b7",
  "endpoint": null,
  "queueId" : "4265408116eb95a16bb6afd864dddf7e",
  "start": 702835610,
  "end" : null
}
```

In contrast, the dynamic view establishes relationships between the hosts and the services and contains information that is only valid for a certain service instance running on a certain host. Listing 4.6 shows this dynamic view of a concrete deployment of the service defined in Listing 4.4 to the host defined in Listing 4.5. The additional information provided in the dynamic view also includes the *endpoint* or *queueId* which both expose the service's functionality. Lastly the start and end of this particular deployment as a Unix timestamp, which are created by the system, are stored.

Because hosts, services, requirements, and properties, along with their relationships, form the basic building blocks of our system, we employ a graph-based model to capture all these facts. More concretely, these relationships describe onto which host a service is deployed at a certain point in time. As well as of which (if any) sub-services a service is comprised of. Additionally, the model allows to defined how services and host relate to certain resources or requirements which describes what they offer or need respectively. By using a graph-based model, we are able to naturally and efficiently query the system state and extract the needed information (e.g., which service can be deployed to which platform).

An added benefit of this representation, is that users can declare NFRs which their services need to achieve, and based on these definitions the system automatically detects which platforms are suited for deploying the individual services to. However, interacting with the graph-based model directly might be cumbersome for most users. Thus the

App Model offers a JSON-based interface that translates the users' definitions to suit our model which can be seen in the Listings above.

4.3.3 Deployment Planner

As we have argued in Section 4.2.3, we employ a CSP solver, to obtain a valid and optimized mapping of services to hosts, which constitutes a deployment plan. The metadata of the services, for example how much of which resources they need and which software versions they are compatible with, is retrieved from the App Model. Additionally, the Deployment Planner fetches the list of valid deployment locations of the individual services that need to be deployed. Furthermore, it obtains the current mapping of services to hosts, which is also taken into account when determining a deployment plan.

Once the Deployment Planner has obtained the above-mentioned information, it translates the data into a format that the MiniZinc [30] solver can handle, and invokes the solver. When it has finished, the Deployment Planner either returns the resulting plan to the Deployment Service, or it informs the user that there exists no plan that satisfies all given constraints.

Because the basic problem formulation for the CSP never changes, we want the problem to be parameterizable, as described in Section 4.2.3. Thus, we use the MiniZinc language [30], which allows the definition of problems that contain variables whose value is not known beforehand, and supply the actual values in a data file. Since MiniZinc is not a solver itself, but only a frontend, that allows a high-level specification of the CSPs, the models need to be translated from MiniZinc's high-level modeling language into a low-level language that the solver backend understands.

The need to compile the model before it can be used by the solver (which can be more time-consuming than the actual solving process, as we determine later in Section 5.2.1) and the desire to derive certain constraints from the NFRs given by the user, lead to the decision to use the App Model to precompute certain constraints. More specifically, we use the App Model to determine the legal deployment platforms for the individual services. This way, the number of possible solutions to the problem (i.e., the number of valid deployment plans) is reduced. Furthermore, this information can be concisely represented, removing the need for supplying the NFRs as parameters to the CSP, which in turn reduces its compile time.

Due to the inherent mismatch of data representation between a CSP model in MiniZinc and our App Model, there is the need to translate data from the App Model's representation into a set of integers, (possibly nested) arrays, and sets. This data can be supplied to the solver, which returns an array of integers, that has to be parsed back and the corresponding host to service mapping has to be recreated from it, see Section 4.5 for an example of how the input data for the CSP solver might look like.

4.3.4 QoS Watcher and CEP Engine

To properly utilize the metrics collected by the Device Manager, we provide a component that receives and analyses them, and executes actions in accordance to certain user-defined rules. This component is made up of two distinct services, namely the *QoS Watcher* and the *CEP Engine*. The QoS Watcher's duties are receiving metrics from the Device Managers, storing them in a timeseries database, and forwarding them to the CEP Engine.

Our decision to have the QoS Watcher run in the cloud is based on the fact that having it run at the edge would mean that the data needs to be distributed across multiple devices (because no single device has enough storage to persist all metrics collected in the system). Furthermore, the process of obtaining stored metrics becomes more complicated and would call for a specialized data retrieval service. By storing the data in the cloud, we get the added benefit that we can replicate it easily to anticipate the failure of a database. This can be done easily, since cloud providers generally offer the possibility to replicate data.

We also decide to deploying the CEP Engine in the cloud, rather than at the edge. This decision is made, because a deployment on the edge would have introduced the need for properly sharding the metrics' database, since it is unlikely that a single device can handle all metrics produced by all devices. By having all metrics go through a single node, we gain the ability to easily analyze the stream of incoming data to detect certain events we might be interested in. Furthermore, by running the service in the cloud we have the additional benefit of it being easily scalable based on the current amount of services running at the edge. However, this way the service becomes a single point of failure, as far as the processing of metrics is concerned. But, although the collection of metrics is an important part of the DDAD framework, it is not a mission-critical component, that has to always be available. In general, if it fails, it is sufficient to start the service again without the system's core functionality being severely impacted.

To enable users to define what constitutes a movement towards an undesirable system state, we allow them to define rules based on the CEP Engine's DSL to detect certain events or chains of events. These rules need to be associated with certain actions that need to be taken. Since the intended context of our system is using it embedded into a system that follows a microservice architecture, we decided that these actions can be represented as service invocations. This way, users are enabled to either use services already in place, or to call one of the framework's services. This definition of actions also eases the introduction of new, specialized services that are able to adapt the system, based on the supplied information. Exemplary callbacks would result in sending an E-Mail or a push notification to the person in charge, the (de-)activation of services on a certain edge device, or the triggering of a new deployment. We also provide a set of blueprints for rules that capture scenarios, which we deem of interest for a broader audience of users. These include the workload of a device rising steadily and breaching a

user-defined threshold for a user-defined period of time. Another example would be the round-trip time for a request exceeding a certain threshold for a predefined time interval.

The execution of a callback associated with an event or a chain of events, generally results in an update of the system's state. The information about the change in state needs to be propagated to all devices and cloud services that might be interested in it. In case of edge devices, the information is pushed to the Device Manager, which then forwards the new information to the services that run on the device it manages. How the services handle this information is up to them and needs to be implemented by the user. The preferred way of doing this, is by using a registry-aware service client. This means, that the client library that is used to access a service, knows that there exists a service registry that holds information about how to reach the individual instances. Having such a client implies that it is possible to choose the optimal service instance to forward the request to, based on the information it has locally available. An example for such a decision would be to choose the service that has the lowest latency, or using a round-robin mechanism to evenly distribute the load across available service instances.

Listing 4.7: Example for Rule Definition With Callback to the App Model

```
{
  "statement": "<CEP-Query>",
  "callbackUrl":
    "http://appmodel.ddad.io/hosts/{host-id}/services/disable",
  "callbackMethod": "PUT",
  "message": "CPU load exceeded acceptable level",
  "arguments": {
    "isCritical": "true",
    "origin": "QoS Watcher"
  }
}
```

Listing 4.8: Example For an Actual Rule That Triggers a Callback

```
SELECT window.maximum FROM
  DefaultWindow(
    process_name='system',
    device='device-id',
    metric_name='cpu_load'
  ) AS window
  WHERE window.average >= 0.75
  OR
  (window.average > 0.7 AND window.maximum > 0.90)
```

Listing 4.7 shows the definition of a callback to be executed when a rule is triggered by the incoming metrics. The definition contains the CEP query which is evaluated by the CEP Engine against the incoming data and which can be see in Listing 4.8. Furthermore, it contains the callback URL which in this case points to the App Model and instructs

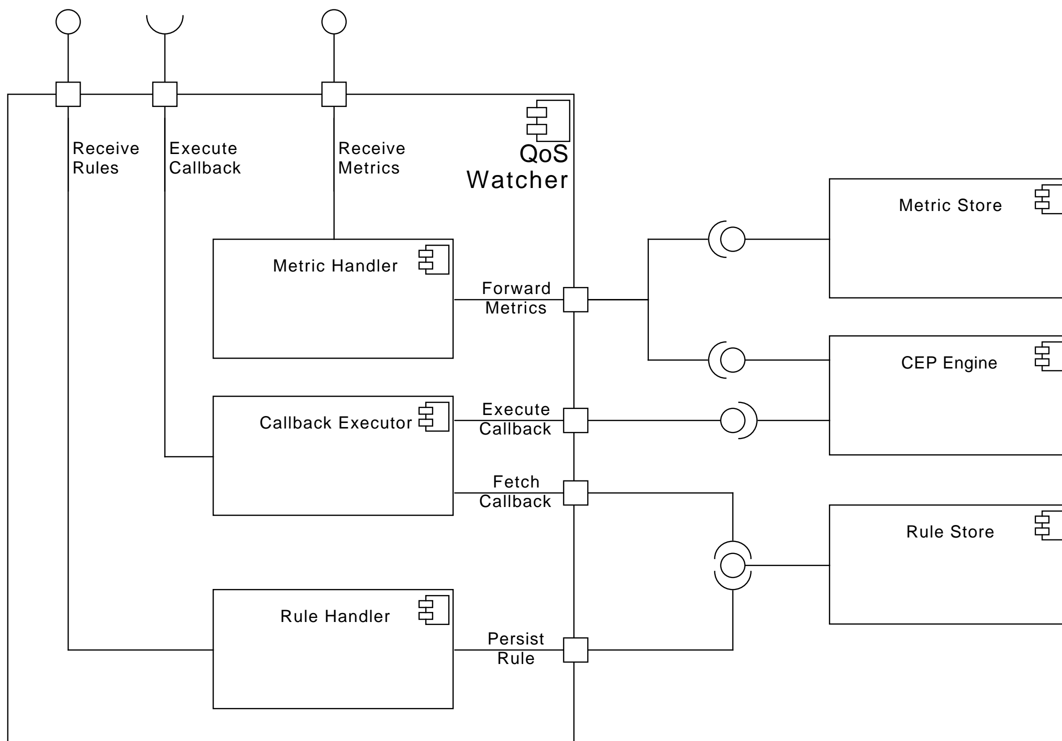

Figure 4.4: Component Diagram of the QoS Watcher

it to deactivate a the services running on this host. In addition to defining the URL which should be invoked, the method which should be used is also defined, since this is an HTTP callback. The user can also specify a message, which is more interesting for the case where the callback triggers for example a push notification to an operator. The same holds true for the arguments, which are also transmitted with the event message and can contain additional information about the event. Listing 4.8 shows an example query which emits an event when the average CPU load of the device with id *device-id* rises above 75% during the default time window, or when its maximum is above 90% and the average above 70%. This is one of the rules we used in Section 5.2.2 during the evaluation of the runtime adaptation mechanism.

Figure 4.4 shows the main components of the *QoS Watcher*. It also displays the *CEP Engine*, which is a simple service, exposing a custom interface to abstract the underlying CEP system used for analyzing the metrics as they arrive. The QoS Watcher itself is comprised of the *Metric Handler* which receives metrics from the Device Manager. It then forwards the metrics to a persistent storage where they are kept for future analysis by developers or operational staff. Furthermore, it forwards them to the CEP Engine, which evaluates user defined rules on the received data. The rules are defined via the

Rule Handler which receives input from the user. The user needs to define a specific rule that should be evaluated on the stream of incoming metrics. Along with this rule a callback (in the form of an endpoint) is specified. Once the CEP Engine detects that a rule matches against the incoming data, the QoS Watcher, more specifically the *Callback Executor* receives information about which rule matched against the incoming data, along with metadata about the events that triggered the rules, and the events themselves. It then fetches the corresponding callback from the *Rule Store* and executes it. Since the users are not only able to specify the endpoint which corresponds to a callback but also the protocol (in the case of HTTP also the method that should be used) the user can choose from a variety of communication mechanisms to distribute the information about an event.

4.3.5 Timeseries Store And Metric Visualization

Although the Timerseries Store and the Metric Visualization are important parts of the proposed framework, because it enables users to manually monitor their applications in real-time, we decided to use off-the-shelf products for realizing these components. We use these components, because these are commodity services that serve a general purpose that does not need to be specifically tailored for the use in our framework. For the Metric Visualization we used Grafana[6], which is written in Go[7] and can be compiled to native Linux code. For the Timeseries Store we used KairosDB[8], which is written in Java [9].

4.4 Static System View

Figure 4.5 shows the individual components of the DDAD framework and summarizes the components in the previous section. It also showcases how the components interact with each other and how users can interact with the system. They can use the exposed interface of the App Model to define their services. Furthermore, they are able to define rules at the QoS Watcher, which are forwarded to the CEP Engine. The CEP Engine then evaluates those rules on all incoming metrics. Should a rule apply to the received data, a user-defined callback, that is defined along with the rules and also stored at the QoS Watcher, is executed. To achieve this, the QoS Watcher receives information about which rule matches and executes the associated callback.

4.5 Dynamic System View

Figure 4.6 shows the information flow for the process of planning and executing the deployment, as well as monitoring and adapting a service. The DDAD framework's tasks, can be described as *Deployment Planning*, *Deployment Execution*, *Monitoring*, and

[6] https://www.grafana.com/
[7] https://www.golang.org/
[8] https://kairosdb.github.io/
[9] https://www.oracle.com/java/

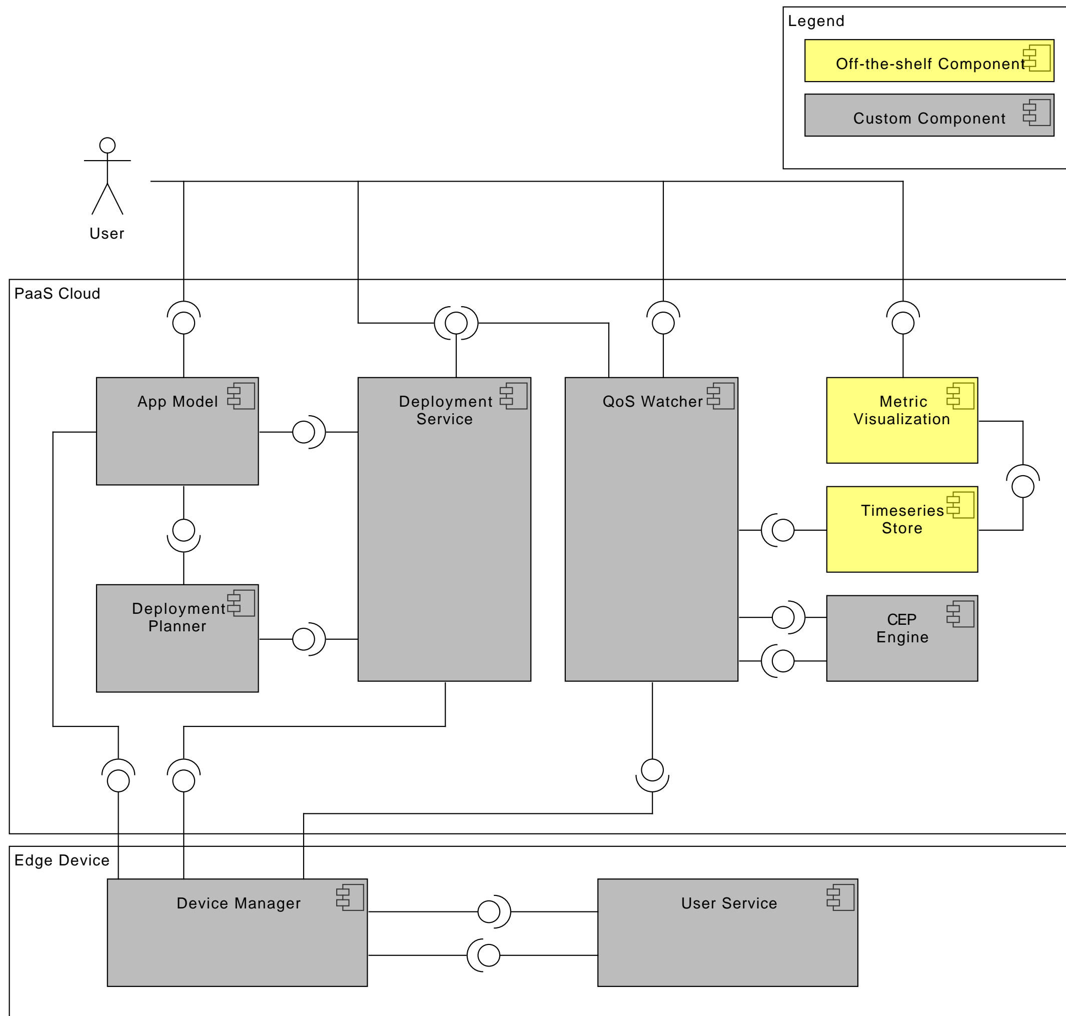

Figure 4.5: Structure of the DDAD Framework

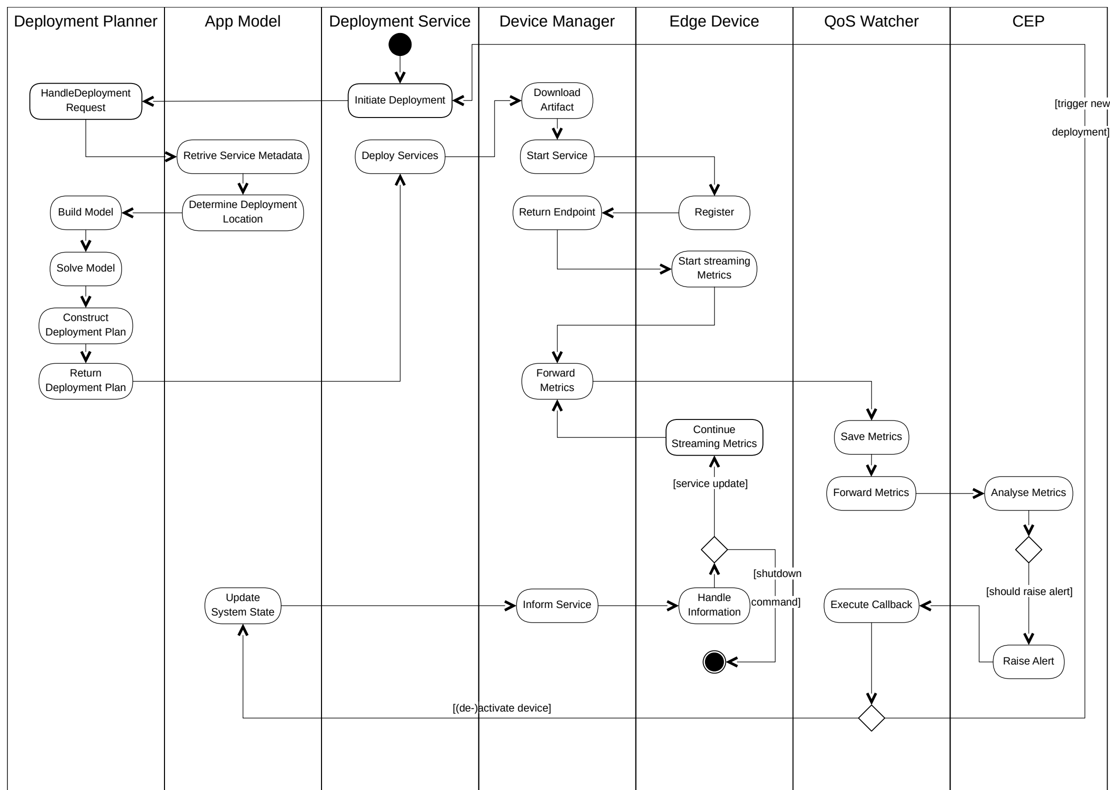

Figure 4.6: Process of Deploying, Monitoring, and Adapting a Service

Adaptation. The following sections will cover each of these individual steps in greater detail.

Deployment Planning

Figure 4.7 shows the process of planning a new deployment in detail. It starts when the Deployment Service receives a request to deploy services. The request contains a list of services that should be deployed, along with the number of instances that should be created. The initiation of a new deployment can either happen manually by the user, automatically by some service (like the QoS Watcher that executes a callback in response to an alert raised by the CEP Engine), or semi-automatically (e.g., as the deploy step in the DevOps cycle). The Deployment Service processes the request and contacts the Deployment Planner, which constructs an optimized placement of services onto hosts. To achieve this, it first needs to gather the needed information, which is obtained from the App Model. This information includes the allowed deployment locations for each service, as well as its resource and software needs. Furthermore, information about the available hosts in the system is returned.

The App Model receives a list of services as input and computes to which platforms

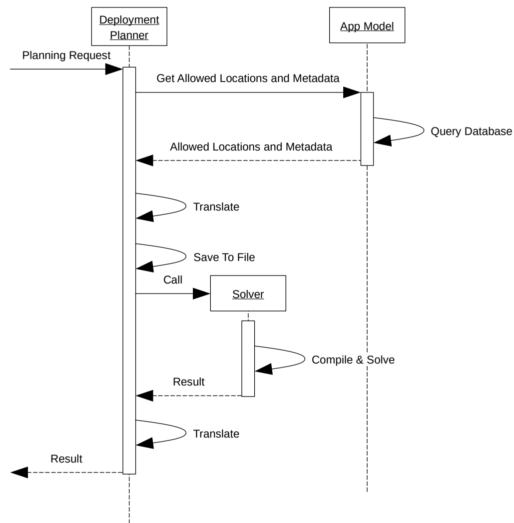

Figure 4.7: Sequence Diagram for the Planning Process

each of them can be deployed to. The computation is done based on the user-defined NFRs of the services as well as the ones of the platforms the users have at their disposal. The information about legal deployment location, together with the current state of the system is returned to the Deployment Planner. The state of the system is comprised of the hosts available to the user, together with their resource and software offerings, as well as the information which services are currently deployed to which hosts. The Deployment Planner uses the obtained information to create the input data for the CSP, calls the solver with the created data, and finally awaits the result. Exemplary input data for the CSP can be see in Listing 4.9. The listing shows an infrastructure of four hosts, three platforms, and two services. Section 4.5.2 goes into more detail explaining how the problem of finding optimized deployment locations for all services is modeled.

Listing 4.9: Example Input of the CSP

```
hosts = 4;
resources = 3;
services = 2;
currently_running_at = [0,2];
locations = 3;
PlatformHosts = {4};

software_count = 3;

has_resources =
[|2700000,3750000,  450000
 |2700000,3750000,  450000,
 |1600000,1200000, 1000000,
 |      0,      0,       0|];

needs_resources =
[|600000, 600000,20000,
 |600000,1200000,20000|];

host_location = [1,1,2,3];
allowed_locations = [{2},{1}];

has_software =
[|{10200},{3500},{180},
 |{10200},{3500},{180},
 |{10200},{3500},{180},
 |{10200},{3500},{180}|];

needs_software =
[|{10200},{3500},{180},
 |{10200},{3500},{180}|];
```

Should there exist a mapping from hosts to services that does not violate any of the

constraints (which the diagram in Figure 4.7 assumes), then the Deployment Planner then parses the result and constructs the corresponding mapping. This mapping is returned to the Deployment Service which then acts accordingly (i.e., distributes the necessary messages across the edge devices).

Should the problem be unsatisfiable (i.e., there is no possiblity to distribute all user services across the available hosts in a way that does not violate any of the constraints) the user is informed and has to reconsider the definition of NFRs, or of the services' resource needs. Another option would be to install additional software onto some of the edge devices, thereby potentially extending the set of services that can run on them. A different approach to overcome this problem would be to force the user to make all services standalone executables or packaging all dependencies together with the services' executables [36]. We choose not to enforce this, since it would mean that if multiple services on the same device have a dependency to the same library, it would need to be packaged into all services. For example, each service that realizes some sort of data analytics at the edge and is implemented in Python would need to be shipped with its own instance of an analytic library. This would increase the filesize and thereby the resource demand of the services unnecessarily. Therefore, we aim to use libraries already present on the edge devices.

Another possibility to reduce the risk of not being able to find a service-to-host mapping that satisfies all constraints would be to obtain the missing software (and possibly install it onto the device), as done in [38]. However, this would add additional complexity to the deployment process, because there are several things that would have to be taken into account. These include the fact that different versions of different libraries could not be allowed to be installed onto the same system, or that the installation of a new library might change the available persistent storage of a device, thereby making it impossible to deploy all planned services to the devices. However, supporting an automatic upgrade of libraries and the additional installation of software dependencies would be a desirable functionality, but is out of the scope of this work.

Deployment Execution

After the Deployment Service received the planning result, it analyses the mapping and sends appropriate messages to all Device Managers that are affected by the new deployment (i.e., those who reside on hosts that either have new services deployed to them or those which need to shutdown services). This process is illustrated in Figure 4.8. On each affected device the following steps happen. First, the Device Manager receives a message that contains a list of services that should be running on the device it manages. Then, it compares this list with the services that are actually running. Thereafter, it determines which services need to be shutdown and which need to be started.

To start a service, the Device Manager first downloads the artifact of the service (as specified in the artifact URL also contained in the received message) and then unpacks

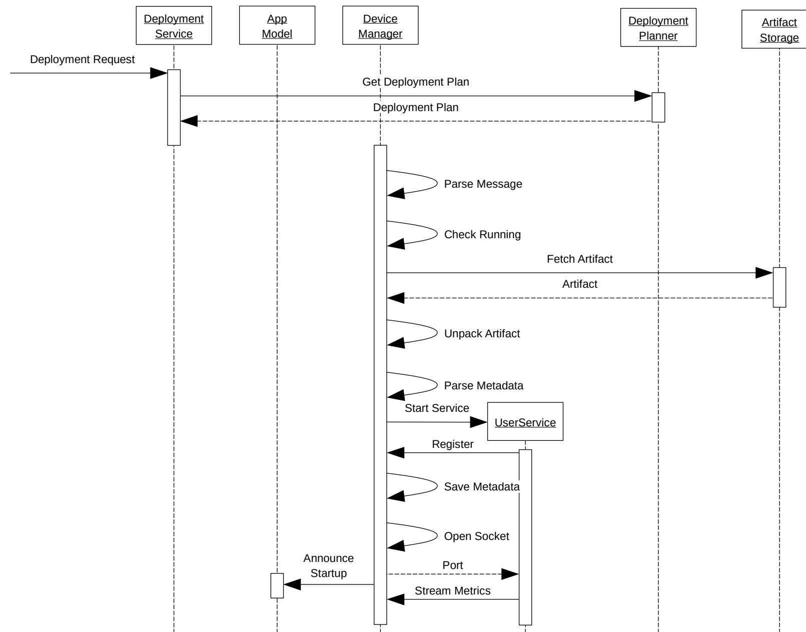

Figure 4.8: Sequence Diagram for the Deployment Process

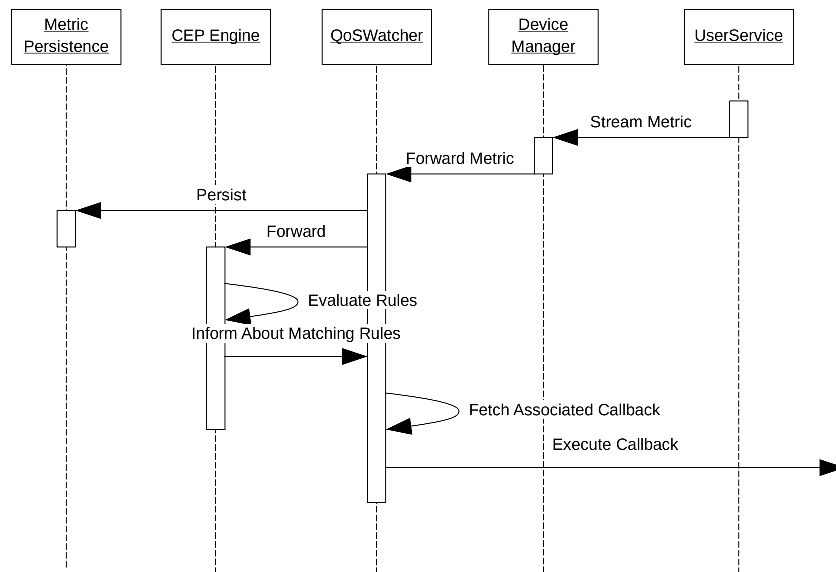

Figure 4.9: Sequence Diagram for the Monitoring Process

it. Besides the executables of the service, the artifact also contains a metadata file. This file contains a list of services which the service depends on, and the command which allows to start it. The Device Manager starts the service by executing the command and stores the metadata in an in-memory database to access it later when updates arrive or when the service needs to be shut down. Along with the metadata, it stores the process id of the service, which is needed to stop it, should it fail to shutdown gracefully. After the service was started, the Device Manager announces the startup and the App Model is updated accordingly. Furthermore, each service must register itself with the Device Manager. Upon registration, the Device Manager assigns a port number to each service and opens a TCP/IP socket so the service can start streaming metrics to it.

Monitoring

Figure 4.9 shows the interaction between the individual components involved in the monitoring process. When the service has registered itself and received a port number, it connects to the socket and starts streaming application-specific service metrics. These metrics might include the time outgoing requests need until the respective response arrives or the size of a queue with working items. These metrics are collected, along with generic service metrics like the relative CPU time the service uses. In addition to metrics that are associated with an individual service, the Device Manager also collects device metrics about the host, like the total CPU usage or the total memory usage. See Section 4.1.3 for a discussion on the different types of metrics and why they are important to collect.

All these metrics are buffered by the Device Manager and pushed to the cloud in a

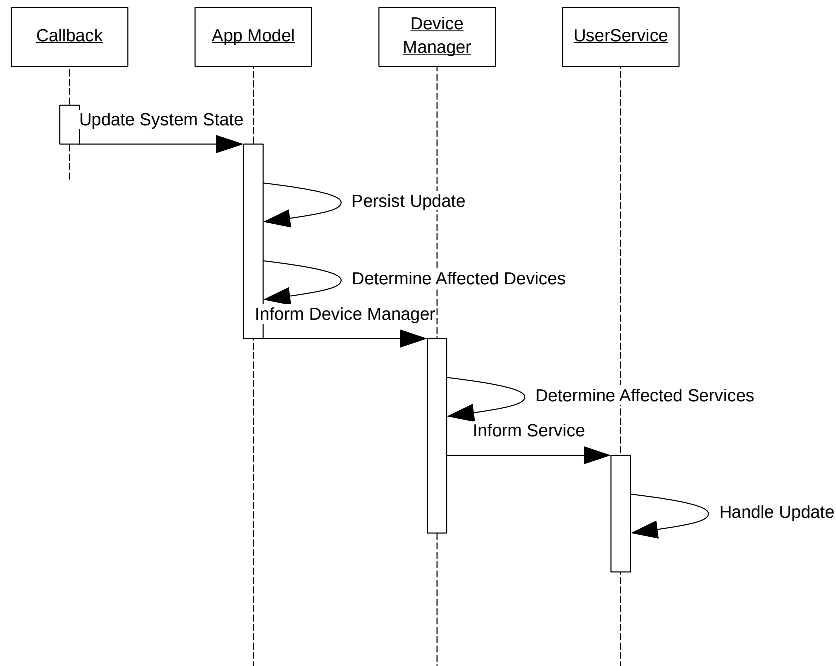

Figure 4.10: Sequence Diagram for the Adaptation Process

predefined interval. In the cloud the QoS Watcher receives these metrics and forwards them to a CEP Engine and persists them for later analysis, an example for such a rule can be seen in Listing 4.8, in Section 4.3.4. The CEP Engine is equipped with user-defined rules to raise alerts when certain conditions are met. For example, a user might want to trigger a push notification to operational staff if the round-trip-time for a request starts to grow and breaches a threshold. Another possible rule would be that an edge host is deactivated when its CPU load exceeds a predefined threshold. The CEP Engine calls the QoS Watcher and supplies it with the information which rule triggered the alert. The QoS Watcher then looks up the callback associated with this rule and executes it. There are two scenarios in which the system is influenced directly; first, the callback can be to the Deployment Service which is instructed to plan and execute a new deployment based on the current state of the system. Should this happen, the system starts the planning, deployment, execution and adaption cycle again. Secondly, the callback could instruct the App Model to deactivate a certain edge device, see Listing 4.7, in Section 4.3.4 for an example of such a callback definition.

Adaptation

The intended result of a user-defined callback after the CEP Engine detected a match for a rule is an update of the system state. In general, this update will either cause the pausing or resuming of a certain service on an edge device or the (de)activation of an edge device. Both actions trigger the runtime adaptation process as shown in Figure 4.10.

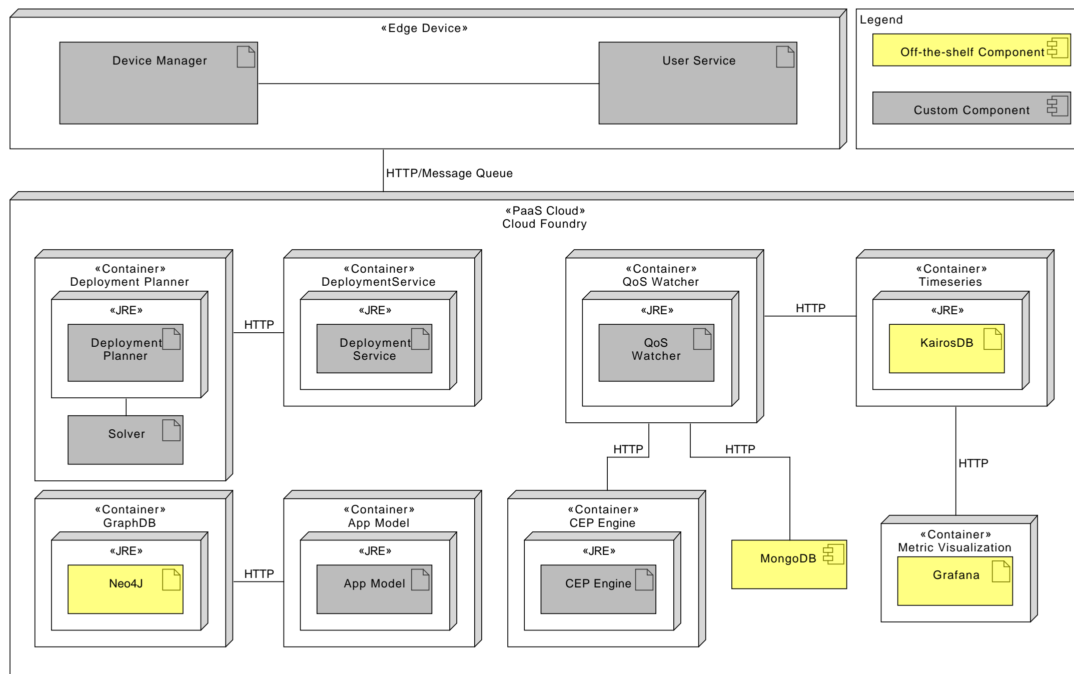

Figure 4.11: Deployment Diagram of the DDAD Framework

The callback (directly or indirectly) updates the system state. This information is handled by the App Model, which determines all devices affected by the change and sends a message to the respective Device Managers. The affected devices are those, where at least one running service uses a service on the edge device that was (de)activated, for the case where an edge device was (de)activated. For the case where only a certain service on a device was stopped, all services that use this service are informed. The message contains a list of alternative service instances that are still available, which are described by the service type they are realizing, the endpoint at which they are reachable, and the protocol that is used to invoke the service.

The Device Manager then looks up which of its running services are interested in which of the received services. It uses the exposed interface of the services to deliver the information. The service receives the information and is now able to adapt to the newly obtained information. For example, the registry-aware service client now knows that it should no longer route requests to a certain service instance.

4.5.1 Deployment Overview

Figure 4.11 shows how the individual components of the DDAD Framework are distributed onto the available infrastructure. As described in Section 4.2.4 we deployed all components except the Device Manager into the cloud. More specifically we used CloudFoundry

which abstracts the underlying infrastructure. In this case, all services deployed to the cloud are packaged in containers that provide the needed runtime environment.

In almost all cases this is the Java Runtime Environment, except for the metric visualization, for which we used Grafana[10], which is written in Go[11] and compiled to native Linux code. Since not all needed commodity services were provided by the platform, we had to package some of them ourselves and deploy them in a dedicated container to make them available to the other components. These include the Neo4J[12] graph database as well as the KairosDB[13] timeseries database.

4.5.2 Deployment Planning

Representing the System State

An important factor when using a combination of cloud and edge is that the users do not have to concern themselves with the question where their services run. Since we wanted to be agnostic with regards to the languages are used to implement the services or the operating system needed to run a service we used a graph-based data model that allows the user to declare which dependencies each service has. These can either be languages, whose runtime needs to be available at the host, operating systems or other software that need to be installed, or language libraries the software depends on, if those dependencies are not already packaged with the executable. The dependencies can also include services that need to be running somewhere else in the system. These services are automatically detected and also deployed if needed.

In Section 4.3.2, we describe the data model that is used to represent defined services, hosts, and platforms.

CSP Model

At its core, the cloud-edge deployment has a planning and an execution phase. In the planning phase, we create the data for a CSP that needs to be solved, in order to fulfill all NFRs and other constraints imposed upon the services. The creation of the data is triggered when the Deployment Planner receives a request for a new deployment plan. This request consists of a list of services along with an instance count, indicating how many instances of each services should be running in total after the deployment process has finished.

The next step for the Deployment Planner is to query the Application Model to obtain a list of legal deployment platforms for each of the services. Furthermore, it obtains a list of currently running services, available hosts, and the specifications of the services to

[10]https://www.grafana.com/
[11]https://www.golang.org/
[12]https://www.neo4j.com/
[13]https://www.kairosdb.github.io/

Table 4.1: Variables Used in the CSP Formulation With Their Intended Meaning

Variable	Intended Meaning
S	Set of service instances
H	Set of hosts
CH	Set of hosts that reside on the cloud
p	A special PaaS-host
P	Set of platforms
R	Set of resource types
T	Set of service types
SW	Set of software
V_{sw}	Set of versions of software $sw \in SW$
t_s	Type of service s
μ_s	Host to which service s is deployed
l_h	Platform on which host h resides
α_s	Set of allowed deployment locations of service instance s
δ_h	Set of services which are deployed to host h
ρ_h^r	Amount of resource r available at host h
ϱ_s^r	Amount of resource r needed by service s
κ_h^r	Costs per unit of resource r at host h
σ_h^{sw}	Set of versions of software sw available at host h
ς_s^{sw}	Set of versions of software sw compatible with service s
φ_s	Current location of service s

deploy. Similar to [29] we then translate the given information, and use a CSP solver to obtain a deployment plan.

There are four basic constraints that need to be fulfilled in order to make a deployment plan valid. These constraints would suffice to obtain a mapping of services to hosts that yields a valid deployment strategy. Informally, they can be formulated as follows

(1) Each service instance needs to be associated with exactly one host.

(2) Each service instance associated with a host must be allowed to be deployed to the location where the host resides.

(3) The resource demand of the service instances deployed to a single host must not exceed the host's available resources.

(4) The host to which an instance of a service is deployed must offer the software it needs to run.

(1) is modeled implicitly by restricting the domain of μ_s to range from 1 to $|H|$. This way, each service is associated with a valid host (to which it will be deployed), and since one service cannot be mapped to multiple hosts, the constraint is automatically fulfilled. μ can be seen as function that maps services to hosts.

To model **(2)** we restricted μ_s to certain hosts such that

$$\forall s \in S : l_{\mu_s} \in \alpha_s$$

where α_s is the set of platforms where the service can be deployed to, which is computed beforehand by the App Model.

(3) is modeled using the set of services which are deployed to a host (δ), the resources a host offers (ρ), and the resources the individual services need (ϱ). δ_h is the set of services deployed to host h, ρ_h^r indicates the resource offering of resource r at host h, ϱ indicates the resource need of resource r for service s. We used the following constraint for all hosts $h \in H \setminus \{p\}$ that are non-PaaS hosts and the services that were deployed to them δ_h:

$$\forall r \in R : (\sum_{s \in \delta_h} \varrho_s^r) \leq \rho_h^r$$

Lastly, we model **(4)** by defining the variables σ, and ς. σ_h^{sw} indicates the version of software sw available at host h, ς_s^{sw} indicates the versions of software sw with which service s is compatible. We enforce the constraint that if a service is compatible with at least one version of a software then the host to which it is deployed needs to provide at least one of these versions. This can be formalized as follows, for all services s which are deployed to a non-PaaS host $h \in H \setminus \{p\}$

$$\forall sw \in SW : \ |\varsigma_s^{sw}| > 0 \rightarrow |\sigma_h^{sw} \ \cap \ \varsigma_s^{sw}| > 0$$

Apart from these four basic constraints, because we want to deploy the services in an optimized way, we also specify an objective function, which is partly comprised of the monetary costs the deployment yields. To calculate these costs, we multiplied the defined costs of each resource with the amount that was used. We decide to model cloud and edge platforms differently with regards to costs, such that resources in the cloud incur costs for the user while resource on the edge are for free, since they are on the user's premise. We choose to disregard possible costs for cooling or additional power consumed by the edge devices.

Another aspect of the deployment we want to optimize is the number of migrations. We choose this because migrating a running service is an expensive and non-trivial task which we want to avoid.

Lastly, it would theoretically be possible to simply deploy all services that might run on both the edge and the cloud to the cloud, which would also yield a valid deployment plan. However, we want to maximize the resource usage at the edge, to take advantage of the available resource. Thus, we define a third part of the objective function, which aims to minimize the amount of unused resources at the edge. To let the user decide how important each factor of the objective function is, we provide the possibility to assign weights to the individual parts of the function.

The three parts of the object function are formalized in the following ways. To calculate the costs we use κ_h^r which corresponds to the costs of resource r at host h and compute

$$\sum_{h \in CH} \sum_{r \in R} \sum_{s \in S} \varsigma_s^r \cdot \kappa_h^r$$

Since we retrieve the full system state when computing a new deployment plan, we also know which services are currently running where. Thus, for each service s, φ_s denotes its previous location, where $\varphi_s \geq 1$ means that the service was previously deployed to a host, and $\varphi_s = 0$ means that it was not. Therefore, we can introduce a penalty for all services whose previous location was not zero, and whose current location does not match their previous location. This gives us the following function

$$\sum_{s \in S'} 1 \qquad \textit{where} \qquad S' = \{s' : s' \in S \land \varphi_{s'} \neq 0 \land \varphi_{s'} \neq \delta_{s'}\}$$

The last part of the objective function aims to minimize the total amount of unused resources at the edge by simply subtracting the amount of used resources from the amount of available resources. This leaves us with the amount of resources we failed to utilize

$$\sum_{h \in H} (\sum_{r \in R} (\rho_h^r - \sum_{s \in \delta_h} \varrho_s^r))$$

After the translation and model creation is done, the CSP solver is invoked. Should it determine for the model to be unsatisfiable with the provided data (i.e., there is no mapping from services to hosts that would satisfy all constraints) the user gets informed about that fact and needs to reconsider the NFRs, resource or software needs of the services, or add new edge devices. However, as mentioned in Section 4.5 there are other possibilities how one could deal with an unsatisfiable problem instance.

If the model was satisfiable (i.e., there is a mapping that satisfies all constraints), the result is parsed by the Deployment Planner and the mapping is transferred to the Deployment Service. This in turn takes the mapping and informs each host that is affected by the deployment plan (i.e., there are services to be deployed to it or undeployed from it). To achieve this task, the Deployment Service sends a message, directly or via the messaging infrastructure in place, to each Device Manager. This message contains the list of services that are expected to run on each device. For each service the Device Manager determines if this service is already running. If so it is discarded, since as we argued in Section 4.5.2, we disallow multiple instances of the same service on the same device. If however, the service is not already running, the Device Manager downloads the archive file specified by the service from some sort of Artifact Registry. The Device Manager then unpacks the files into a specified directory and parses the metadata-file, which contains information how the service is started, and which other services it depends on.

The Device Manager then obtains a list of service instances from the Service Registry that are instances of the services the new service depends on. Afterwards, the new service is started according to the command specified in the metadata-file. All services (that depend on some other services) need to expose a REST endpoint on the local machine, to which the Device Manager pushes updates. This way, each service is aware that there are multiple services and can potentially determine the optimal service to which it should route a request autonomously. Another purpose of the exposed interface is to provide a possibility to shut a service down gracefully by calling the appropriate endpoint.

After the services have been started, they call the Device Manager to obtain a port number to which they will stream application-specific service metrics. Simultaneously, the Device Manager announces the startup of the services at the Service Registry.

Since the Device Manager receives a full list of services that should be running on the device it is responsible for, it also computes the difference of the set of services that are already running and the ones that should be running. This results in the services that need to be shutdown. Upon startup of each service, the Device Manager obtains the process-id for each service it starts and saves it in an in-memory database. Thus it can easily look up the services and simply kill the process with the corresponding id should it fail to stop after receiving the respective instruction from the Device Manager.

4.6 Summary

After identifying the main requirements of a framework that realizes Data Driven Automatic Deployment in an edge computing scenario, we provided an insight into the rationale behind the key design decisions of the framework. To realize the abstraction of heterogeneous edge devices, we use a custom software agent that runs on each individual edge device, similar as described by previous research [6, 20, 38, 40]. We defined the responsibilities of this agent, which are the announcement of the start-up of a service instance, as well as publishing metrics of the services to the cloud. Furthermore, it

receives commands that allow to deploy and undeploy services, as well as to forward information about a change in the system's state to the services it supervises. The push based collection and pre-processing of metrics realizes another key requirement of the framework, namely providing a mechanism for runtime monitoring of IoT applications.

Apart from that, we enable finding optimized deployment strategies for cloud-edge applications based on a knowledge graph. This knowledge graph holds information about which services are located where, what resources services that are about to be deployed need, and what resources and software the edge devices can offer. On the basis of this knowledge, we can determine valid deployment platforms for the services and thereafter we can use a CSP solver to optimally distribute the services across these platforms, similar to [29]. We went into great detail to present how we formalized this problem in a way that the CSP solver can understand it.

Additionally, we presented our key design decisions and described how they influenced the architecture of the system. These were the realization of a MAPE-K cycle, which we choose because there is nearly a one-to-one correspondence of the steps in such a cycle and the processes in our framework. Next, a push-based monitoring solution with a cloud storage was chosen to minimize the amount orchestration and organization that the framework has to take care of, regarding the monitoring processes. This was achieved by shifting the responsibility of delivering metrics to the user-defined services and the Device Manager. We also argued why we chose a microservice architecture, since it gives us much more flexibility than a monolithic system design [5].

Lastly, we gave a high level overview of the system architecture, as well as detailed insight into the individual component that make up the DDAD framework. To optimally utilize the resources at hand we decided to deploy part of the framework to the edge and part of it to the cloud. Furthermore, a dynamic system view was presented. Therein we explained in great detail how the individual components interact with each other and which information they need to exchange. We presented the four main processes in the system, which are the *Deployment Planning*, *Deployment Execution*, *Monitoring*, and *Adaptation*, and gave a conceptual overview of how they are executed.

CHAPTER 5

Evaluation

In this section, we present the findings of the performance study we conducted in order to verify the validity of our approach. We do this by performing runtime experiments with a well-defined application, which realizes the machine learning use case described in Section 1.1. This section aims to answer the following questions

1. How well does the Deployment Planner's problem definition for finding a valid deployment plan perform under a growing number of services?
2. How does the runtime behavior of the application differ when using cloud-only or edge-only computation as compared to employing the DDAD framework for runtime adaptation?
3. How much of the computation in the cloud can be moved to the edge without drastically affecting the adherence to previously defined QoS parameters?
4. How intrusive is the Device Manager with regards to resource consumption?

To answer Question 1, we define a fictitious, yet realistic, deployment scenario with a fixed number of hosts and growing number of services. By measuring the time the CSP solver needs to compile the problem and find a solution, we get a metric on the basis of which we can evaluate how well the problem was modeled. Section 5.2.1 describes the exact setup in greater detail.

As a basis for answering Questions 2, 3, and 4, we implement and deploy the sample application, which realizes the motivating example which is presented in Section 1.1. The setup of the application is described in Section 5.1. Since all three questions are concerned with one or more runtime metrics, we use the DDAD framework's mechanism

to collect these metrics to evaluate them. We ran the experiment a total of three times to achieve more meaningful results.

In addition to collecting and analyzing metrics, we also need to define QoS parameters to which the application must adhere to properly answer Question 3. These parameters along with the reasoning behind why we choose them, is presented in Section 5.1.

5.1 Setup and Context

The condition monitoring scenario which is described in Section 1.1 is chosen as an exemplary use case for our framework. The scenario is an analytics use case, where device owners want to evaluate the current state of an asset, based on a previously trained model. Although this scoring is not a vital or mission-critical task, we aim to keep the round-trip-time of scoring requests under one second. We choose this threshold because data is read in one second intervals and we want to avoid that requests are starting to impound. Since executing user services is generally not the primary task of edge devices, we need to make sure that the user services do not consume too much of the devices' resource. Therefore, we want to keep the overall resource consumption at a reasonable level, which still keeps some buffer to cope with unexpected workload bursts. To achieve this we define a total 75% as the CPU load threshold for the device, meaning that we aim to keep the 15 second average of the overall CPU load of the device below this value. We choose the 15 second average, so we do not react prematurely to singular spikes in the workload. Furthermore, we assume that the device owner wants to harness the available resources at the edge to reduce the workload in the cloud, which means finding a trade-off between offloading computation to the cloud and to adhering to the defined QoS parameters [39].

To simulate an edge, device we use a t2.micro instance on Amazon EC2[1] which has a single 3.3 GHz CPU core and 1 GB of RAM. This properly emulates a dedicated IoT Device that collects data from multiple sources since it lies roughly in the middle between the most powerful and the most constraint Raspberry Pi[2], a popular single-board-computer, which is often cited as an example for a typical edge device [38, 40]. Onto this device we install our Device Manager and a data acquisition software. Besides these two services there is also a *Connectivity Service* in place, that forwards the data which was obtained by the data acquisition service to the cloud. With this data, a model is trained in the cloud with appropriate methods and uploaded to a registry from which the local scoring service can fetch it. The DDAD framework runs on a CloudFoundry installation on top of the SAP Cloud Platform[3].

Figure 5.1 shows the desired setup when using a dynamically adaptable service client to choose which concrete service instance should be invoked. It shows a multitude of field

[1] https://aws.amazon.com/ec2/
[2] https://www.raspberrypi.org/
[3] https://cloudplatform.sap.com/

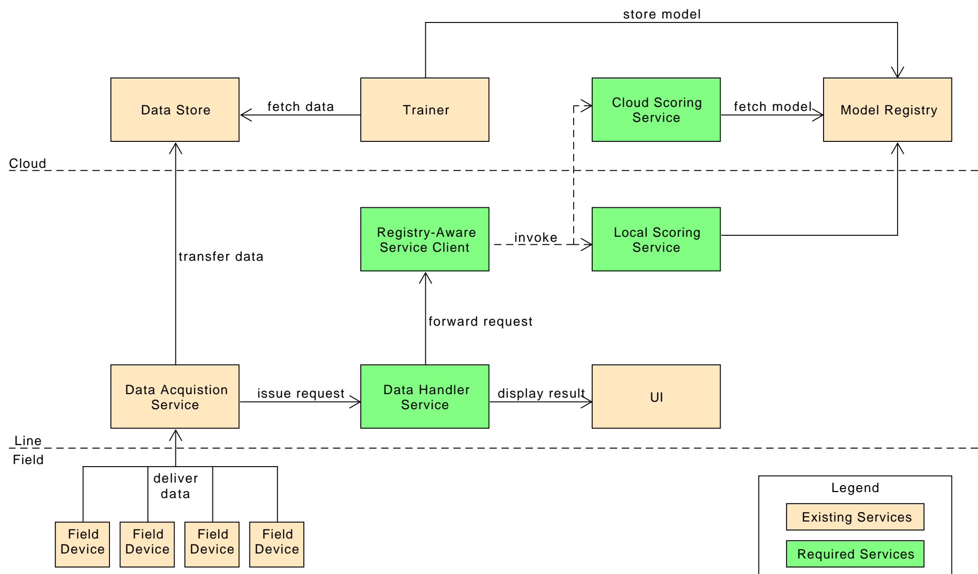

Figure 5.1: Logical View of the Motivating Example

devices (e.g., sensors that measure different properties of an electric drive) that collect data. The data is obtained by a Data Acquisition Service, which in turn forwards it to some Data Store Service, as well as to a Data Handler Service. This handler wants to score the incoming data and calls the Registry-Aware Service Client, which dynamically decides which concrete instance of the Scoring Service (either locally or in the cloud) should be invoked. The decision is based on the information the Registry-Aware Service Client has available locally, which could be as simple as a list of available services, or detailed information about them, like their average response time.

Upon startup the Data Handler Service subscribes itself to the Data Acquisition Service which then publishes sensor readings in a one second interval. The Data Handler comes with a Registry-Aware Service Client to which it delegates the scoring requests. The handler itself depends on a service which can score data (i.e., an abstract Scoring Service, which is potentially realized by multiple concrete implementations). However, since the scoring process should be transparent for the user, it does not depend on any particular instance.

There are two different instances of the Scoring Service in place. One of them is the Local Scoring Service, which is implement in C# and utilizes bindings for the R programming language[4]. Appropriate R libraries are used for the actual scoring of the model. The

[4]https://rdotnet.codeplex.com/

other one is the Cloud Scoring Service, which uses OpenScoring[5] to evaluate the incoming data. By default the local instance of the Scoring Service is used to utilize resources at the edge. This continues until the CEP Engine (which applies user-defined alerting rules) instructs the system to deactivate the local instance of the scoring service and forces the Registry-Aware Service Client to forward the data to the cloud for evaluation.

5.2 Performance Measurements

5.2.1 Deployment Planning

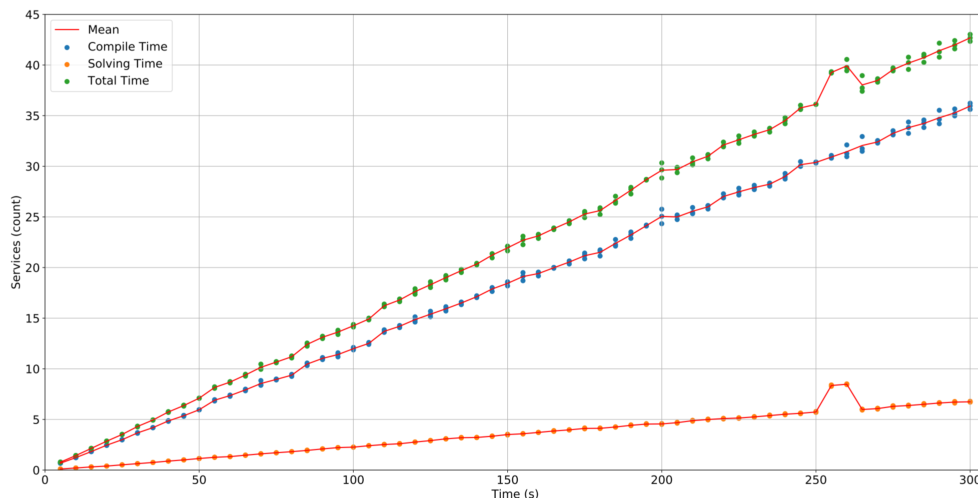

Figure 5.2: Compile and Solving Time for the CSP

As a first step, we examine how fast our CSP model can compute an initial deployment plan. Since the defined test application only consists of a small number of services whose requirements are easily determined, we decide to randomly generate a multitude of fictional services and hosts to test the model based on this data. Thus, we define four types of edge hosts, each with different software and resource offerings. Additionally, we define five services with different software and resource needs. These come in three flavors, (i) those that must only run at the edge, (ii) those that must only run in the cloud, (iii) and those for which the NFRs do not restrict the deployment platform. Since it is likely that resource-intensive applications are deployed to the cloud, and those at the edge have generally lower resource needs [6], we model these assumptions by defining cloud-only services with high resource needs, edge-only services with low resource needs, and those where the target platform is unrestricted with moderate resource needs.

Figure 5.2 displays the result of the performance measurement when using 60 hosts and increasing the number of services to deploy. All of the hosts reside at the edge.

[5]https://openscoring.io/

Additionally there is a pseudo-host, without resource and software constraints, that models a PaaS cloud. We start with 5 services and increased the number to 300 in steps of 5. For each iteration, a quarter of the services is only allowed to be deployed to the edge, a quarter is only allowed to be deployed to the cloud, and the remaining half can be deployed to the cloud and the edge as well.

The limiting factor of the solving process is the need to compile the model with the given data, as Figure 5.2 shows. This takes up to roughly 45 seconds for 300 services, while the time the solving process itself needs also grows steadily but much more slowly than the time needed to compile. Although the model (especially the compiling) does not hold up under a large amount of hosts and/or services, it works reasonably well for the intended context, namely users deploying services to devices they have available in their plant.

5.2.2 Runtime Adaption

We realize that a major key performance indicator which needs to be measured when evaluating the feasibility of our approach, is how the runtime adaptation mechanism of the DDAD framework can influence the adherence to defined QoS parameters and the amount of computation that can be moved from the cloud the edge (which generally results in a reduction of costs). To examine this, we use the setup described in Section 5.1. The performance measurements are gathered by using the Device Manager, which is part of the DDAD framework, to collect the metrics and push them to the cloud.

We distinguish three scenarios which need to be compared:

1. Doing all computational work at the edge, thereby saving the most costs and keeping the round-trip-time of requests extremely low but risking the overloading of an edge device.
2. Moving all computation away from the edge and evaluating the collected data in the cloud, in which case the CPU load of the edge devices is drastically reduced. However, this will incur longer round-trip-times as well as additional costs, since the usage of cloud resources generally incurs monetary of costs.
3. Applying runtime adaptation based on predefined rules, with the aim of finding a proper trade-off between the CPU load of the edge device, the duration of evaluating the gathered data, and the costs which have to be paid when utilizing cloud resources.

For the performance measurements, we classify the state of 45 assets per second, since we determine this to be a reasonable number of assets that are handled by one edge device. Since we do not want to exceed the available bandwidth or incur too much load onto the device for simply transferring metrics to the cloud, we choose to compute the average of all requests, returning during one second. This way we do not create (and

more importantly do not have to transmit) 45 datapoints per second from a low-powered device, but can reduce this number to one.

As mentioned above, we want to achieve a trade-off between the three driving factors in this scenario, the CPU load of the edge device, the time it takes for scoring requests to be handled, and the costs induced by using cloud resources. Since we do not want to react to singular spikes in measurements, the rules also take an sliding average into consideration when determining to switch from/to local scoring. Thus we derive the following rules for the QoS Watcher to match the incoming data against and the corresponding actions to execute in response to these rules:

1. Deactivate the computation on the edge device when the 15 second average of the CPU load is above 70% and the maximum during this period is above 90%.
2. Reactivate the computation on the edge device when one of the following occurs
   a) The 15 second average of the time it takes to handle a scoring request exceeds 600 ms.
   b) The average handling of a scoring request takes longer than 550 ms and the maximum during the 15 second sliding window is greater than 900 ms.

To establish a baseline of our performance measurements we let the system handle all computation locally. Later we move the whole computation to the cloud. After the baseline is established, we put the above mentioned rules for runtime adaptation in place and start the services while continuously monitoring them. As a callback, we define a call to the App Model which either activates or deactivates the scoring service on the edge device as a result. Listing 4.7 in Section 4.3.4 shows how the definition of this rule looks like. The (de)activation is propagated to the Device Manager, which then has to forward this information, to enable the Registry-Aware Service Client to act accordingly.

Since it is highly unlikely that our service(s) will be the only one running on an edge device we also execute a script that continuously uses 35% of the device's CPU. This can be seen as the device's primary task, which cannot be stopped and with which the scoring service must not interfere.

System CPU Load

Figure 5.3 displays the system's CPU load over the course of ten minutes of scoring 45 request per seconds when only using the simulated edge device. Since only the local service is used to score the model the load mostly stays between 60 and 80%. However, there are a lot of times, when the CPU Load exceeds the threshold of 75%.

In Figure 5.4, the edge device's overall CPU load when scoring all data in the cloud can be seen. As expected, the total load is substantially lower because the scoring itself is

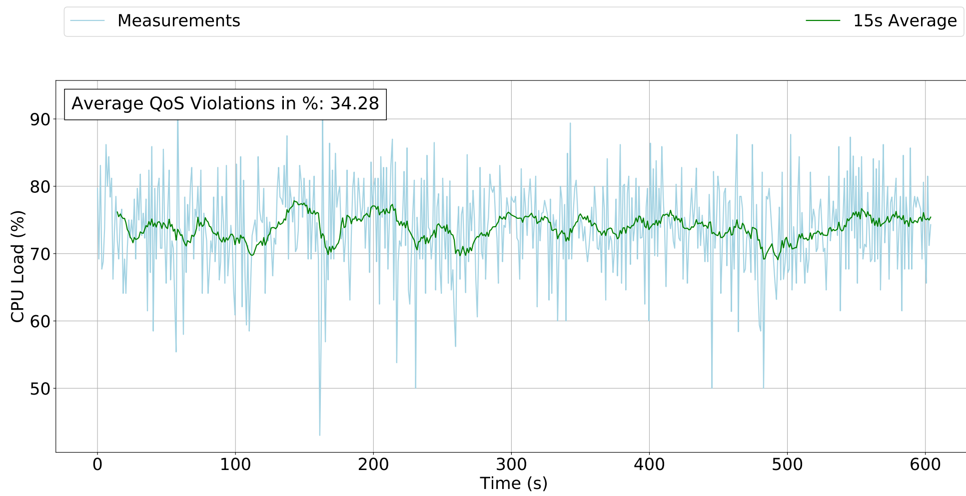

Figure 5.3: CPU Load of the Edge Device When All Computation Is Done on the Edge Device

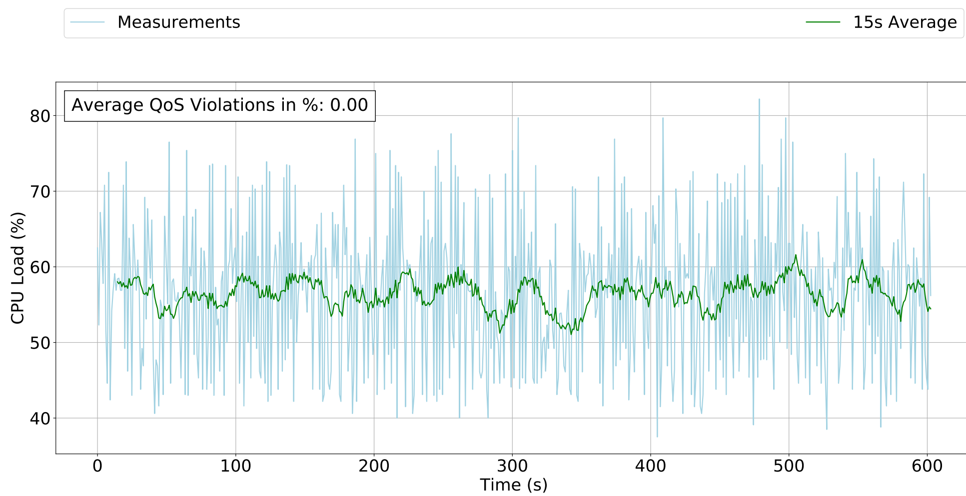

Figure 5.4: CPU Load of the Edge Device When Completely Moving the Computation to the Cloud

offloaded, and the service only has to wrap the data, transfer it to the cloud, and receive the result. The results also show that the CPU load fluctuates much more heavily than when only scoring locally.

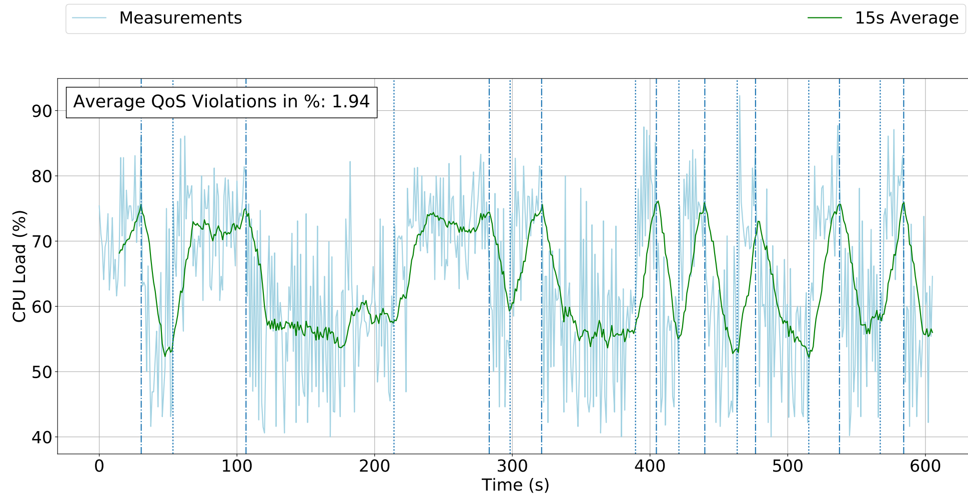

Figure 5.5: CPU Load of the Edge Device When Using Runtime Adaptation to Switch Between Local and Cloud Scoring

Figure 5.5 showcases the CPU load of the edge device when using the implemented runtime adaption by applying the rules described above. The goal being that the devices does not become overloaded while still not leaving too much computational resources at the edge unused. Also, Figure 5.5 shows that the average load oscillates between 55 and 75% and that once the device is deactivated the load drops rapidly. However, it also rises quickly after the device is activated again.

User Service CPU Load

Since the scoring service is the only service running on the edge device it comes as no surprise that the CPU load for the scoring service in Figure 5.6 looks very similar to the overall CPU load in Figure 5.3 only offset by roughly 45 percentage points. We can see that the load fluctuates between a maximum of 40% and a minimum of 0%. The minimum of 0% can be explained by the fact that the measurements were taken in a one second interval, so the process had no CPU time for this particular second.

Figure 5.7 shows the CPU load of the Scoring Service when using only cloud scoring. The load distribution over time is very similar to the one in Figure 5.4, again only offset by roughly 45 percent points. Again, the load does not fluctuate as much as compared to scoring locally, as showcased in Figure 5.6.

In Figure 5.8, the CPU load of the Scoring Service when using runtime adaptation is displayed. Again, the collected data looks very similar to the overall CPU load of the

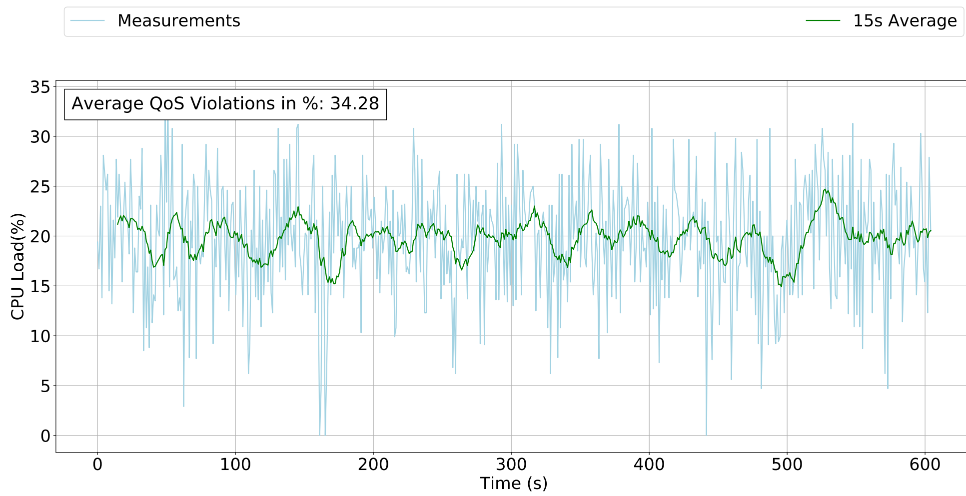

Figure 5.6: CPU Load Induced by the User Services When Using Only Local Scoring

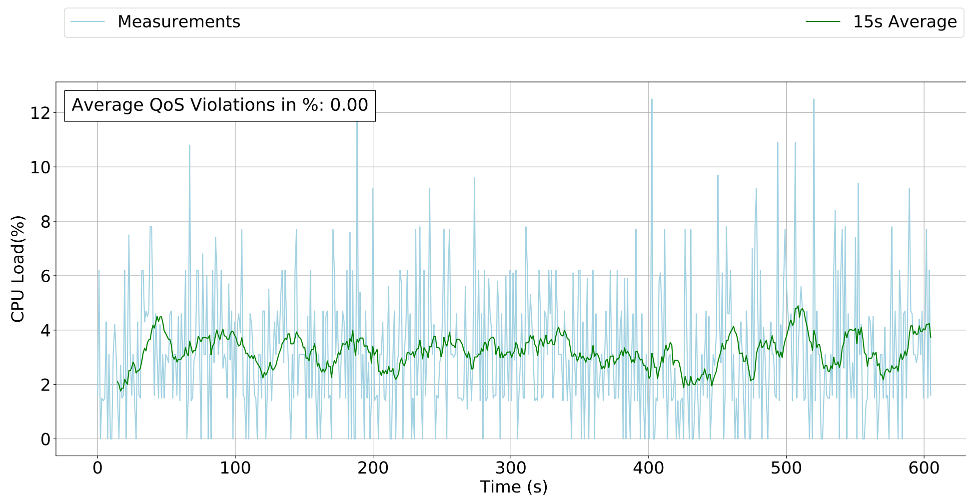

Figure 5.7: CPU Load Induced By the User Services When Completely Moving the Computation to the Cloud

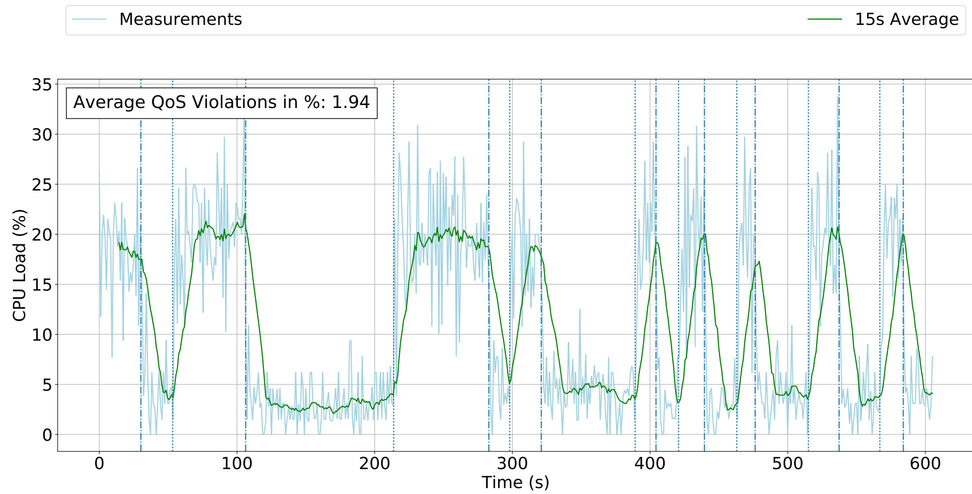

Figure 5.8: CPU Load Induced by the User Services When Using Runtime Adaptation to Switch Between Local and Cloud Scoring

device which can bee seen in Figure 5.5.

Round-Trip-Time of Scoring Requests

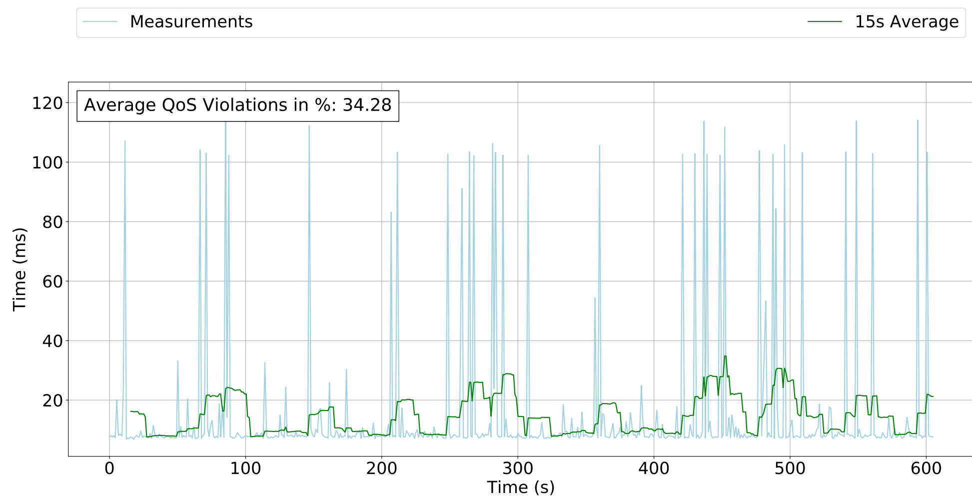

Figure 5.9: Round-Trip-Time for Scoring Requests When Using Local Scoring

Figure 5.9 shows the round-trip-time for scoring requests when only using local scoring. We measure the time that it takes to receive a result from the scoring service, once the received data was parsed. This includes transforming it into a common representation, sending it to the service, receiving the result and parsing it back. Figure 5.9 shows that for local scoring this takes around 120 ms at most, which more than over-performs with

regard to our defined goal of keeping the response time under one second. This low round-trip-time however, comes at the costs of a high CPU load for the edge device as Figure 5.3 and Figure 5.6 show.

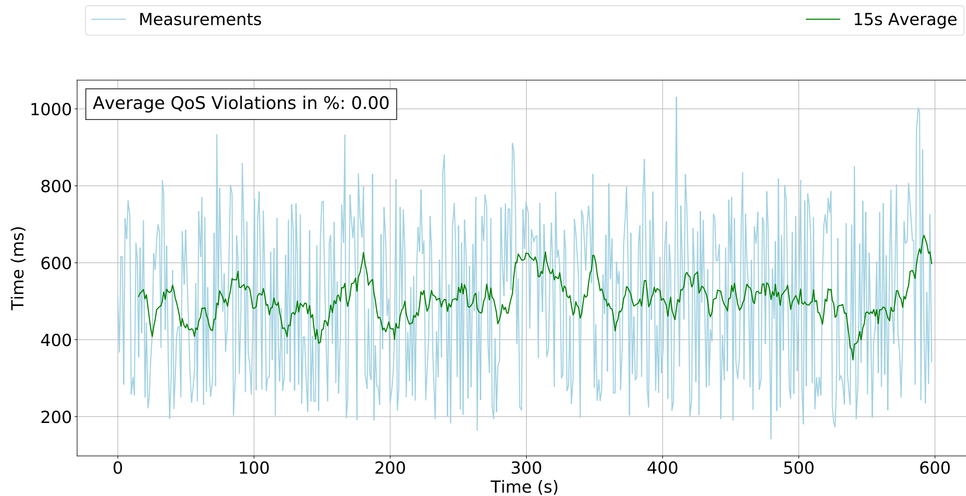

Figure 5.10: Round-Trip-Time for Scoring Requests When Using Cloud Scoring

Figure 5.10 displays the resulting round-trip-times for scoring requests when moving the entire computation to the cloud. As expected, the measurements show that it takes substantially longer to receive results when scoring in the cloud. This comes as no surprise, since the data has to be transferred to the cloud, handled there, and sent back again. Especially the sending and receiving of the requests introduce the main part of the latency. However, as Figure 5.4 and Figure 5.7 show, this increase in round-trip-time brings a drastic reduction in CPU Load.

Figure 5.11 shows the response time when using the runtime adaptation approach described above. We can see that the round-trip-time oscillates, very similar to the CPU load in Figure 5.8. Also similar is the fact that measured values change rapidly after switching from local to cloud execution and vice-versa. However, in contrast to the CPU load, the round-trip-time goes up when offloading the computation to the cloud to reduce the CPU load and is reduced when shifting back to local computation.

SLA Violations & Cloud Resource Usage

As mentioned at the beginning of Section 5.1, our goal is to keep the 15 second average of the edge devices' CPU load under 75%. Furthermore, we want to keep the response time for scoring requests under one second. These two goals can be seen as the SLAs of our application to which we want to adhere as best as possible. In addition to fulfilling these SLAs we also want to minimize the cloud resource usage, which in turn helps to

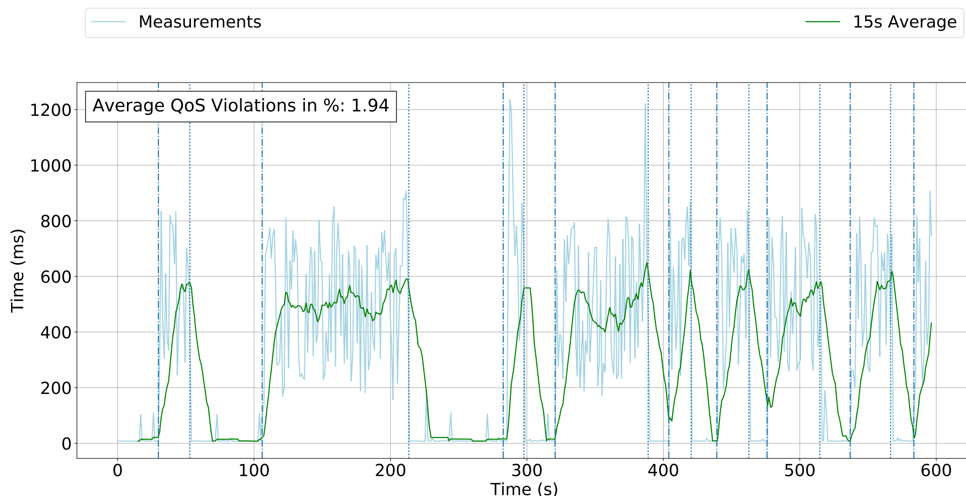

Figure 5.11: Round-Trip-Time for Scoring Requests When Using Runtime Adaptation to Switch Between Local and Cloud Scoring

Table 5.1: SLA Violations for Different Scenarios

Metric	Method	Minimum	Maximum	Average	σ
Latency	Local	0%	0%	0%	0%
	Cloud	0.33%	0.5%	0.39%	0.10%
	Adaptation	0.5%	1.33%	0.89%	0.42%
CPU Load	Local	21.17%	55.33%	34.28%	18.42%
	Cloud	0%	0%	0%	0%
	Adaptation	1.67%	2.17%	1.94%	0.21%

reduce the overall costs for the end user, since cloud services generally operate on a pay-as-you-go basis.

Thus we analyzed how often the CPU load threshold and the latency threshold are exceeded, when using each of the different methods and how much of the computation is done in the cloud.

Table 5.1 shows the number of QoS violations for the different scenarios. As one can see, the adaptation has a much lower count of instances where the CPU load exceeds 75% when compared to the edge-only scenario. More precisely, when using the runtime adaptation mechanism the average number of CPU-related SLA violations can be reduced by roughly 94%, from 34.28% to 1.94%. However, when using the runtime adaptation mechanism, more CPU load violations occur than when using the cloud-only scenario. Furthermore, it becomes clear that the adaptation scenario performs worse than both the edge- and cloud-only scenario when it comes to latency. The fact that the adaptation

Table 5.2: Percentage of the Computation by Platform for the Adaptation Scenario

	Cloud	Edge
Minimum	53.97%	40.04%
Maximum	59.96%	46.03%
Average	56.86%	43.14%
σ	3.00%	3.00%

scenario has more latency-related QoS violations than the cloud-only one stems from the fact that when switching to executing the scoring in the cloud the first few requests might take longer than the following ones, especially when switching after a long period of edge-only scoring.

Although, when examining Table 5.1 in isolation it may look like the adaptation scenario does not bring any benefits, Table 5.2 shows that we are able to reduce the amount of cloud resources needed drastically. In the best instance of the experiments this means nearly cutting the computation power for the service in half, while still keeping the requests that exceed the maximum latency below 1% and the time the CPU load threshold is exceeded below 2%. Compared to the scenario where the computation happens exclusively on the edge we can see that by accepting a slightly worse response time (for some instances), we can cut the average time that the CPU runs overloaded (according to the defined threshold) to a 20$^{\text{th}}$ of the original value.

This shows that when using the DDAD framework's runtime adaptation mechanism, one can reach a rather satisfying trade-off between saving costs and adhering to QoS parameters. Furthermore, it also demonstrates how extensively computing power at the edge can be harnessed without interfering with the edge devices' primary tasks. However, one has to be aware that there will always be some kind of trade-off. How much overloading or long running requests one can put up with will always depend on the particular use case.

Figure 5.12 shows the number of times the round-trip-time SLA was violated. This count is plotted against the amount of computation that was done in the cloud. Furthermore, it shows the regression line, which was obtained via the least-squares method [42], where one tries to find a line such that the sum of the squared distances between the individual measurements and the line is minimized. Furthermore, the Figure shows the boundaries of the 95% confidence interval. One can see that there seems to exists a correlation between the number of SLA violations and the amount of computation done in the cloud. This is not surprising since transferring data from and to the cloud generally induces an overhead, and the more data one transfers to the cloud, the higher the likelihood of a request's round-trip-time exceeding the previously defined QoS parameter becomes.

The number of times the 15 second average of the edge device's CPU load exceeded the previously set QoS value of 75% can be seen in Figure 5.13 plotted against the amount of

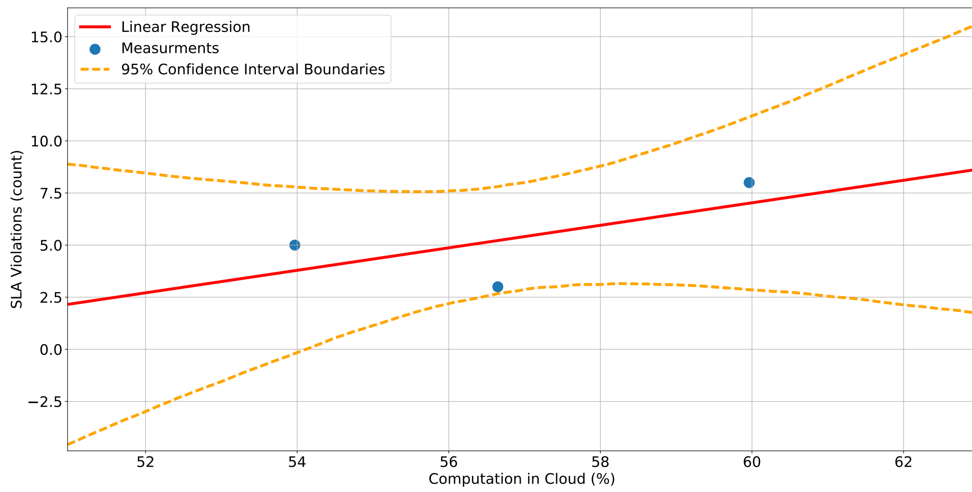

Figure 5.12: Round-Trip-Time SLA Violations During Runtime Adaptation

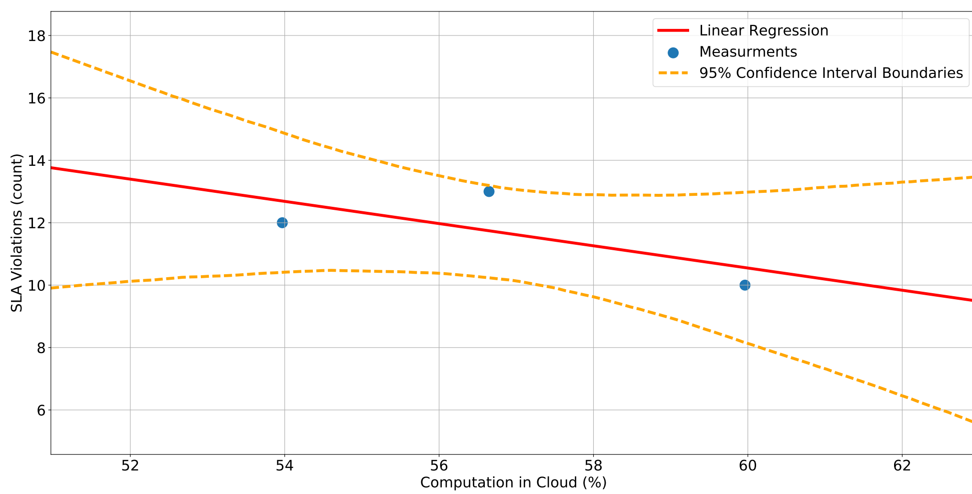

Figure 5.13: CPU Load SLA Violations During Runtime Adaptation

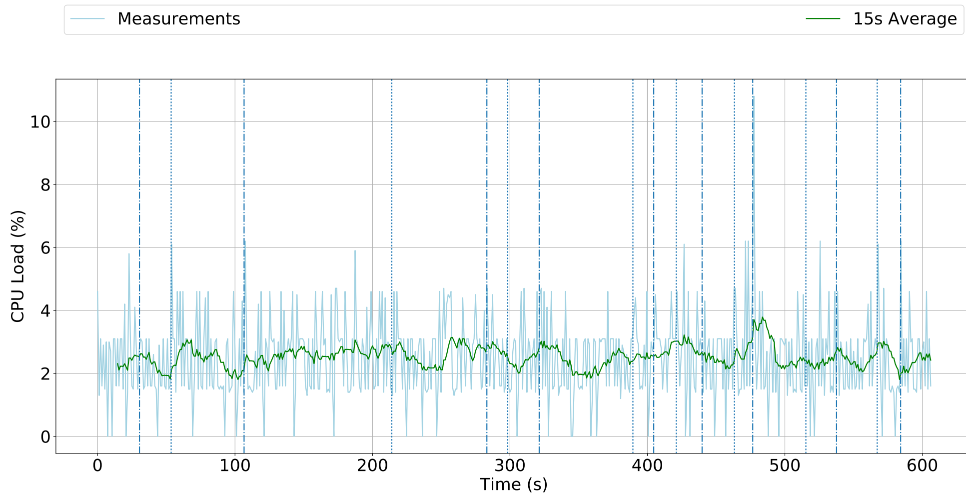

Figure 5.14: CPU Load of the Device Manager During Runtime Adaptation

computation done in the cloud. Additionally, one can see the regression line, which also was obtained by the least-squares method. The line shows a trend which indicates, that the more computation is done in the cloud, the fewer CPU-load-related SLA violations occur. This is not surprising because the more computation one moves away from the edge device the fewer resources are needed to achieve the remaining computational tasks.

5.2.3 Intrusiveness of Device Manager

Another factor when evaluating the framework was determining how intrusive the Device Manager was, in terms of consumed memory. This factor is important, because we do not want the manager to incur too much load onto the system, while not bringing immediate benefit to the user. A Device Manager that uses too much resources would discourage users from employing the framework since, computational power is generally a limited resource at the edge [17, 37]. Furthermore, it would invalidate the overall assumption that the Device Manager has a reasonable resource consumption.

Figure 5.14 shows the CPU usage of the Device Manager while running our test application as described above when using runtime adaptation, because with this option the Device Manager needs to do the most work. This stems from the fact that it has to receive and parse messages from the DDAD framework and inform the system accordingly. The figure shows that the manager uses between about 2 and 3% of the CPU which constitutes an acceptable amount for such an important service. Furthermore, the change indicators show that there appears to be no relation between a change of the scoring location and a change in the Device Manager's CPU consumption.

5.3 Summary

In this section, we answered the four question we posed at its beginning by showcasing the results of runtime experiments. These experiments included the execution of the CPS that is used to determine a valid deployment plan and the analysis of its compile- and runtime. Furthermore, we implemented and deployed a sample application that we used to determine the benefit of employing the DDAD framework's runtime adaptation mechanism over using cloud- or edge-only computation.

To evaluate the quality of the formulation of the CSP that determines the optimized deployment location for each service, and answer Question 1 we created a multitude of artificial, yet realistic, services, that need to be deployed to a set of hosts. The evaluation showed that the most time-consuming factor of solving the CSP was the compiling part. However, we argued in Section 4.5 that having a parameterizable problem that only needs to be provided with the data is an important feature of the Deployment Planner. Furthermore, when using other solvers, like the IBM CPLEX Optimizer[6], the data also has to be provided and the actual constraints have to be derived from it. Additionally, since the intended use case of our framework is for a plant operator to deploy services to their available devices, the scale, at which the experiment is conducted, is appropriate, and the runtime of the solving for 60 hosts and 300 services at about one minute, can also be seen as acceptable.

Next we answered Question 2 by evaluating how the runtime behavior of the application changes when employing the DDAD framework's runtime adaptation mechanism. The experiments show that there is a substantial reduction of CPU load incurred onto the edge device and the amount of computation done in the cloud can be substantially reduced, when changing the evaluation location of scoring requests based on the current state of the device. However, the round-trip-time of scoring requests, grows drastically. This fact however, was anticipated, and is acceptable, because the when moving computation from the edge to the cloud, one trades fast response times for reduced CPU load. The results also show that, once the device is instructed to not invoke the local service anymore, there is a rapid drop in CPU load, which is not surprising, since the data handler then only has to wrap and unwrap requests and responses respectively.

Table 5.1 and Table 5.2 show that we were able to move a substantial amount of computation from the cloud to the edge, while still adequately adhering to the previously defined QoS parameters, which answers Question 3. By allowing to move some of the computation to the edge, we enable users to achieve a trade-off between their QoS parameters and the operational costs of their system. More precisely, we were able to reduce the costs by 43% on average, as compared to the cloud-only scenario, while simultaneously reducing the number of CPU-load-related SLA violations by roughly 94%, when compared to the edge-only scenario. However, these numbers are first and foremost only averages, and Table 5.1 shows that standard deviation of the measurements can be

[6]https://www.ibm.com/software/commerce/optimization/cplex-optimizer/

quite high. Furthermore, our goals was to achieve a trade-off, which implies that every gain, usually comes with a loss. In this concrete case, the loss is a larger amount of latency-related SLA violations. More precisely, an increase by a factor of 2.28 on average as compared to the cloud-only scenario. However, the overall number of latency-related SLA violations is still below 1.5% on average, which is an acceptable number, given the cost savings when compared to the cloud-only scenario (which are roughly 40% on average) and the SLA improvements when compared to the edge-only scenario.

Lastly, Figure 5.14 shows that the resource consumption of the Device Manager can be described as reasonable, since its 15 second average never exceeds 4% and the maximum is around 12%, which also allows to answer Question 4.

CHAPTER 6

Discussion & Conclusion

6.1 Comparison to Related Work

To highlight, what separates our work from the presented, related work, we want to point out key differences between the holistic framework we propose and the approaches that are presented in the literature. The biggest being that our framework is a holistic approach that aims to cover the whole lifecycle of an application, from planning and executing the deployment, over monitoring the running services, to using the obtained information to realize runtime adaptation.

Bonomi et al. [6] provide a detailed insight into fog computing's conceptual basics. They achieve this by sketching out several use cases and deriving key requirements for an orchestration middleware layer, that facilitates fog computing. Although supplying a sound, theoretical framework, whose goal it is to exploit the complementary benefits of cloud and edge computing in a single system, we see a lack of an realization of this framework that could act as a reference implementation for fog computing. In contrast to that, we provide a prototypical implementation of a framework that enables users to harness resources present at the edge, and allows the combination of said edge-resources with cloud services in a sensible fashion.

Since, cloud and edge platforms both bring complementary benefits and drawbacks, one might be better suited as the execution environment of a task than the other one, based on different resource-, as well as, non-functional-requirements of the indivdual services [6]. Deciding, which part of an application can run on the edge, and which parts should be executed in the cloud is not a trivial task which can be approach in multiple ways. Hong et al. [19] choose a service as the finest level of granulairty which can be executed independently. Although, we agree with this approach, the authors assume that an applications can be split into a hierarchical structure that implicitly dictates where each

part of the application can run [19]. At a much finer level of granularity, Chun et al. [9] propose a static code analyzer that moves computation from Android applications to a nearby server. A similar approach is taken by Cuervo et al. [10], which use annotations to let the user decide which parts of a .NET based application can be offloaded to the cloud.

The most important difference between [19] and our proposed solution is, that in our framework the services' hierarchy does not directly dictate where a service should run. In contrast, we focus on resource- and non-functional-requirements to determine admissible deployment strategies. What differentiates the DDAD framework from the MAUI framework [10] and the approach taken by Chun et al. [9] is that it is not bound to any specific programming language or software ecosystem. Thereby, providing the possiblity to use a wider variety of tools and porgramming languages based on the users' needs. However, the granularity at which parts of an application can be offloaded is much more coarse-grained in our framework since the smallest conceptual part is a service. Urging users to employ a microservice architecture forces them to plan their service architecture accordingly, but unlike [19] it does not enforce any structure upon the application. Furthermore, Bass et al. [5] and Balalaie et al. [3] argue, that using such an architecture helps to cope with the complexity of highly distributed applications.

The question how individual components of an application (in our case these components are the individual services) should be deployed onto available infrastructure, is answered by van der Burg et al. [36]. They propose a tool that automatically deploys given services onto available infrastructure. Similar to the Disnix tool [36] we use a software agent on each device to which user services can be deployed. However, when using the approach described in [36], users have to define the mapping of services to hosts themselves, which is something we explicitly want to avoid, and which our framework takes care of in an automated way. Thereby, the amount of automation in the deployment process can be further increased which helps reducing errors and speeds up the time it takes until a change in a service makes it to production.

Gabbrielli et al. [16] and Matougui and Leriche [29] all employ a CSP based tool that determines the optimal location. This tool is used in combination with a deployment framework, that takes care of actually deploying the individual services to the determined locations. However, our knowledge graph not only represents resource and software requirements, but also takes non-functional-requirements (e.g., data-privacy, elastic-scalability, ...) into account when determining the admissible deployment locations for the individual services. Another key difference to our proposed framework, stems from the fact, that the other presented solutions do not concern themselves with monitoring and automatically adapting the system once it is in place. We identified these tasks to be at the core of an holisitc framework that supports the developers and operational staff throughout the lifecycle of a service once it has gone into production.

6.2 Limitations and Future Work

Although, we have determined our framework to be a possible starting-point for unifying cloud and edge services, its functionality is by no means complete and there are much possibilities for future work. The most important and apparent ones can be summarized as follows:

Optimized Monitoring Intervals We choose a fixed observation interval for all metrics that are collected by the Device Manager and a fixed interval for the delivery of metrics from the Device Manager to the QoS Watcher. However, the best intervals are likely not the same for each device and each metric [12]. Thus it would be better to dynamically change the time between individual measurements and between individual meteric deliveries, so we incur an optimal amount of overhead, similar to [12]. This way it would be possible to further reduce the intrusiveness of the Device Manager and improve the monitoring process.

Dynamic Detection of Joining and Leaving Devices For the purpose of this work we assumed that the topology at the edge and in the cloud remains stable between deployments. But, since the edge can be rather changing with regards to what devices are available, it would be favorable to automatically react to devices joining and, more importantly, leaving the network. This would mean that in the cloud one of the services (most likely the QoS Watcher or the App Model) would have to keep track of heartbeat messages of the Device Managers and react accordingly for example, by triggering a new deployment.

Improved Adaptation Strategies The responsiblity of defining actions that need to be taken should the system move towards an undesireable state was shifted to the users of the DDAD framework. However, defining generic adapation strategies as proposed by Huber et al. [21] would increase the advantages for the users, since they could potentially be shared through a central repository. Thereby, removing the need for users to define their own actions for common adaptation scenarios. Furthermore, it would be desirable to not only completely activate or deactivate hosts and edge devices, but to set a maximum percentage of workload that clients are allowed to route there. This would allow for a more fine grained adaption of the services. To decide which services should actually be invoked by others, an approach similar to what Chen et al. [8] propose, would yield a better runtime adaptation. However, this would imply, that all service would have to obtain additional information about other services, or collect this information themselves.

Automatic Detection of System Health Deterioration To define what an undesirable system state is can be done by defining QoS parameters to which individual applications or the system as a whole have to adhere. However, defining what event or chain of events indicate such a movement is also left to the user in our framework. It would be much more desirable to have a way to automatically derive rules for the users.

This could for example be done by using a pobabilistic model, as done by Chen et al. [8]. Another possibility to free users of the burden of manually defining and fine-tuning rules that indicate a deterioration of the system's health would be to employ machine learning techniques or the use of artificial intelligence, to proactively take action should a SLA violation be immanent, similar to [24, 26].

Automatic Installation of Missing Software The inability to automatically install needed software packages onto the target edge devices as done by Vögler et al. [38], is another limiting factor of the DDAD framework. This limitation could be overcome by either using an approach similar to the one presented in [38] or by using Docker-based artifacts. Since the data model of the App Model is very flexible it would be possible to model the individual services in way that they have a runtime dependency to the Docker engine being installed on the target device and defining the artifact of the services as a Docker image. However, it would be necessary to make adjustments to the Device Manager, since accessing a Docker registry works different from simply downloading a file from a server.

6.3 Summary

In this work we aimed to provide a holistic framework that allows users to profit from computational resources available at the edge of the network, while still being able to make use of the virtually unlimited power of the cloud. The combination of the benefits (and the overcoming or mitigation of the drawbacks) of both platforms should happen transparently to the users, freeing them of the burden of tailoring their application to either one of the platforms. Furthermore, the framework should support the users in all stages of operating a system based on a microservice architecture. This includes (i) planning and executing the deployment process automatically, (ii) monitoring deployed services, and (iii) gathering runtime metrics about the deployed services, thereby influencing (i).

First, we provided the necessary background for this work by introducing fog computing as an emerging paradigm to utilize cloud and edge resources in a way that exploits their complementary benefits and drawbacks. Furthermore, we presented the microservice architectural style and how systems that employ such a style differ from traditional monolithic applications. Lastly, we gave an overview of the DevOps methodology and how it colludes with a microservice architecture to help cope with the complexity inherent to distributed systems.

After establishing a background knowledge, we gave an overview of the current state of the art in the subfields which were of interest for the course of this work. This includes a more detailed discussion of current approaches in fog and edge computing, as well as a deeper look into the current state of automatic deployment mechanisms. Furthermore, we presented approaches how efficient and unintrusive runtime monitoring can be realized, especially for cloud-edge scenarios and QoS relevant monitoring. Lastly, we discussed

how the runtime behavior of individual services and an application as a whole can be influenced dynamically by employing different approach to realize runtime adaptation.

Next, we conducted a detailed discussion about the design and implementation of the proposed DDAD framework. We started by identifying its key requirements, namely (i) the abstraction of heterogeneous edge devices to allow transparent access to the computational resources. (ii) automatically determining and executing optimized deployment strategies in cloud-edge scenarios. (iii) the definition and implementation of an unintrusive and efficient runtime monitoring mechanism for IoT applications, as well as (iv) a mechanism for dynamic runtime adaptation of said applications. Once we identified the requirements we presented our key design decisions along with their justifications.

Thereafter, we discussed the individual components of the framework, how they communicate with each other, and which data they exchange. In the course of this discussion we first presented each component in separation and then detailed their interaction during the individual stages of an application's lifecycle (i.e., deployment planning, deployment execution, monitoring, and adaptation).

To show the validity of our approach, we implemented a sample application, that realizes a use case in an industry context. Furthermore, we defined a set of QoS parameters which the services needed to adhere to. We then conducted a number of runtime experiments to show that when using the proposed framework and employing its runtime adaptation mechanism, one can achieve a trade-off between SLA violations and costs incurred by using cloud resources.

Based on the results of the experiment we concluded that a rather satisfying trade-off between QoS adherence and reduced costs for computational cloud resources can be reached by using runtime adaptation. One can either choose to have no costs for cloud resources by keeping all computation locally, while simultaneously regularly overloading the edge devices. Alternatively, it is possible to pay “the full price“ by having all computation in the cloud which brings the benefit of never overloading ones devices but also introduces latency-related SLA violations. However, we assume that users want to find a trade-off between regularly violating SLAs by using only edge devices and incurring high monetary costs by using only cloud computing. Thus, we also conducted the experiments while employing the DDAD framework's runtime adaptation mechanism. We defined a set of rules based on the previously defined QoS parameters to switch back and forth between using cloud and edge resources in a way that aimed to keep SLA violations at a minimum, while still allowing the maximum amount of computation to be done at the edge. This allowed us to reduce the amount of computation done in the cloud to by 43% on average while still keeping the average number of SLA violations below 2%.

List of Figures

1.1 Logical View of the Motivating Example 6

4.1 Architecture of the Framework 26
4.2 Integration of the DDAD Framework into a DevOps Workflow 30
4.3 Component Diagram of the Device Manager 34
4.4 Component Diagram of the QoS Watcher 41
4.5 Structure of the DDAD Framework 43
4.6 Process of Deploying, Monitoring, and Adapting a Service 44
4.7 Sequence Diagram for the Planning Process 45
4.8 Sequence Diagram for the Deployment Process 48
4.9 Sequence Diagram for the Monitoring Process 49
4.10 Sequence Diagram for the Adaptation Process 50
4.11 Deployment Diagram of the DDAD Framework 51

5.1 Logical View of the Motivating Example 61
5.2 Compile and Solving Time for the CSP 62
5.3 CPU Load of the Edge Device When All Computation Is Done on the Edge Device 65
5.4 CPU Load of the Edge Device When Completely Moving the Computation to the Cloud 65
5.5 CPU Load of the Edge Device When Using Runtime Adaptation to Switch Between Local and Cloud Scoring 66
5.6 CPU Load Induced by the User Services When Using Only Local Scoring 67
5.7 CPU Load Induced By the User Services When Completely Moving the Computation to the Cloud 67
5.8 CPU Load Induced by the User Services When Using Runtime Adaptation to Switch Between Local and Cloud Scoring 68
5.9 Round-Trip-Time for Scoring Requests When Using Local Scoring 68
5.10 Round-Trip-Time for Scoring Requests When Using Cloud Scoring 69
5.11 Round-Trip-Time for Scoring Requests When Using Runtime Adaptation to Switch Between Local and Cloud Scoring 70
5.12 Round-Trip-Time SLA Violations During Runtime Adaptation 72
5.13 CPU Load SLA Violations During Runtime Adaptation 72

5.14 CPU Load of the Device Manager During Runtime Adaptation 73

List of Tables

4.1 Variables Used in the CSP Formulation With Their Intended Meaning . . . 53

5.1 SLA Violations for Different Scenarios 70
5.2 Percentage of the Computation by Platform for the Adaptation Scenario . . . 71

Listings

4.1 Structure of a Deploy Command Message 32
4.2 Structure of a Service Update Command Message 32
4.3 Example For a Metadata File 33
4.4 Static Definition of a Service in the App Model 35
4.5 Example Definition of an Edge Device 36
4.7 Example for Rule Definition With Callback to the App Model 40
4.8 Example For an Actual Rule That Triggers a Callback 40
4.9 Example Input of the CSP 46

Acronyms

API Application Programming Interface.

CD Continuous Delivery.

CEP Complex Event Processing.

CI Continuous Integration.

CIL Common Intermediate Language.

CSP Constraint Satisfaction Problem.

DDAD Data-Driven Automatic Deployment.

DSL Domain Specific Language.

EC2 Elastic Compute Cloud.

IaaS Infrastructure as a Service.

IoT Internet of Things.

NFR Non-Functional-Requirement.

PaaS Platform as a Service.

QoS Quality of Service.

REST Representational State Transfer.

SaaS Software as a Service.

SCM Source Code Management.

SLA Service Level Agreement.

SOA Service Oriented Architecture.

UI User Interface.

URL Unique Resource Locator.

VM Virtual Machine.

Bibliography

[1] Mohammad Aazam and Eui-Nam Huh. Fog computing and smart gateway based communication for cloud of things. In *Future Internet of Things and Cloud (FiCloud), 2014 International Conference on*, pages 464–470. IEEE, 2014.

[2] Jean-Paul Arcangeli, Raja Boujbel, and Sébastien Leriche. Automatic deployment of distributed software systems: Definitions and state of the art. *Journal of Systems and Software*, 103:198–218, 2015.

[3] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[4] Arijit Banerjee, Xu Chen, Jeffrey Erman, Vijay Gopalakrishnan, Seungjoon Lee, and Jacobus Van Der Merwe. Moca: a lightweight mobile cloud offloading architecture. In *Proceedings of the eighth ACM international workshop on Mobility in the evolving internet architecture*, pages 11–16. ACM, 2013.

[5] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[6] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. Fog computing: A platform for internet of things and analytics. In *Big Data and Internet of Things: A Roadmap for Smart Environments*, pages 169–186. Springer, 2014.

[7] Eric Brewer. A certain freedom: thoughts on the cap theorem. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 335–335. ACM, 2010.

[8] Manman Chen, Tian Huat Tan, Jun Sun, Jingyi Wang, Yang Liu, Jing Sun, and Jin Song Dong. Service adaptation with probabilistic partial models. In *18th International Conference on Formal Engineering Methods*. Springer, 2016.

[9] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[10] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.

[11] Roberto Di Cosmo, Michaël Lienhardt, Ralf Treinen, Stefano Zacchiroli, and Jakub Zwolakowski. Optimal provisioning in the cloud. technical report of the aeolus project. 2013.

[12] Vincent C Emeakaroha, Ivona Brandic, Michael Maurer, and Schahram Dustdar. Low level metrics to high level slas-lom2his framework: Bridging the gap between monitored metrics and sla parameters in cloud environments. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 48–54. IEEE, 2010.

[13] Niroshinie Fernando, Seng W Loke, and Wenny Rahayu. Mobile cloud computing: A survey. *Future generation computer systems*, 29(1):84–106, 2013.

[14] Martin Fowler and Matthew Foemmel. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, 122, 2006.

[15] Martin Fowler and James Lewis. Microservices. *ThoughtWorks. http://martinfowler. com/articles/microservices. html [last accessed on July 04, 2017]*, 2014.

[16] Maurizio Gabbrielli, Saverio Giallorenzo, Claudio Guidi, Jacopo Mauro, and Fabrizio Montesi. Self-reconfiguring microservices. In *Theory and Practice of Formal Methods*, pages 194–210. Springer, 2016.

[17] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric computing: Vision and challenges. *ACM SIGCOMM Computer Communication Review*, 45(5):37–42, 2015.

[18] Sara Hassan and Rami Bahsoon. Microservices and their design trade-offs: A self-adaptive roadmap. In *Services Computing (SCC), 2016 IEEE International Conference on*, pages 813–818. IEEE, 2016.

[19] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*, pages 15–20. ACM, 2013.

[20] Pengfei Hu, Sahraoui Dhelim, Huansheng Ning, and Tie Qiu. Survey on fog computing: architecture, key technologies, applications and open issues. *Journal of Network and Computer Applications*, 2017.

[21] Nikolaus Huber, André van Hoorn, Anne Koziolek, Fabian Brosig, and Samuel Kounev. Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments. *Service Oriented Computing and Applications*, 8(1):73–89, 2014.

[22] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010.

[23] Shih-Hao Hung, Chi-Sheng Shih, Jeng-Peng Shieh, Chen-Pang Lee, and Yi-Hsiang Huang. Executing mobile applications on the cloud: Framework and issues. *Computers & Mathematics with Applications*, 63(2):573–587, 2012.

[24] Dragan Ivanović, Manuel Carro, and Manuel Hermenegildo. Constraint-based runtime prediction of sla violations in service orchestrations. In *International Conference on Service-Oriented Computing*, pages 62–76. Springer, 2011.

[25] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[26] Philipp Leitner, Anton Michlmayr, Florian Rosenberg, and Schahram Dustdar. Monitoring, prediction and prevention of sla violations in composite services. In *Web Services (ICWS), 2010 IEEE International Conference on*, pages 369–376. IEEE, 2010.

[27] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[28] Fang Liu, Jin Tong, Jian Mao, Robert Bohn, John Messina, Lee Badger, and Dawn Leaf. Nist cloud computing reference architecture. *NIST special publication*, 500(2011):292, 2011.

[29] Mohamed El Amine Matougui and Sébastien Leriche. A middleware architecture for autonomic software deployment. In *ICSNC'12: The Seventh International Conference on Systems and Networks Communications*, pages 13–20. XPS, 2012.

[30] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

[31] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, 2008.

[32] Jakob Puchinger, Günther R Raidl, and Ulrich Pferschy. The multidimensional knapsack problem: Structure and algorithms. *INFORMS Journal on Computing*, 22(2):250–265, 2010.

[33] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4), 2009.

[34] Dmitry I Savchenko, Gleb I Radchenko, and Ossi Taipale. Microservices validation: Mjolnirr platform case study. In *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on*, pages 235–240. IEEE, 2015.

[35] Olena Skarlat, Stefan Schulte, Michael Borkowski, and Philipp Leitner. Resource provisioning for iot services in the fog. In *Service-Oriented Computing and Applications (SOCA), 2016 IEEE 9th International Conference on*, pages 32–39. IEEE, 2016.

[36] Sander Van Der Burg and Eelco Dolstra. Disnix: A toolset for distributed deployment. *Science of Computer Programming*, 79:52–69, 2014.

[37] Luis M Vaquero and Luis Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *ACM SIGCOMM Computer Communication Review*, 44(5):27–32, 2014.

[38] Michael Vögler, Johannes M Schleicher, Christian Inzinger, and Schahram Dustdar. A scalable framework for provisioning large-scale iot deployments. *ACM Transactions on Internet Technology (TOIT)*, 16(2):11, 2016.

[39] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: concepts, applications and issues. In *Proceedings of the 2015 Workshop on Mobile Big Data*, pages 37–42. ACM, 2015.

[40] Emre Yigitoglu, Mohamed Mohamed, Ling Liu, and Heiko Ludwig. Foggy: A framework for continuous automated iot application deployment in fog computing. In *AI & Mobile Services (AIMS), 2017 IEEE International Conference on*, pages 38–45. IEEE, 2017.

[41] Rostyslav Zabolotnyi, Philipp Leitner, Waldemar Hummer, and Schahram Dustdar. Jcloudscale: closing the gap between iaas and paas. *ACM Transactions on Internet Technology (TOIT)*, 15(3):10, 2015.

[42] Kelly H Zou, Kemal Tuncali, and Stuart G Silverman. Correlation and simple linear regression. *Radiology*, 227(3):617–628, 2003.