FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Design and Implementation of a Fog Computing Framework

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Kevin Bachmann, BSc
Matrikelnummer 1126001

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Assistant Prof. Dr.-Ing. Stefan Schulte
Mitwirkung: Olena Skarlat, MSc

Wien, 10. Februar 2017

Kevin Bachmann   Stefan Schulte

Design and Implementation of a Fog Computing Framework

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Kevin Bachmann, BSc
Registration Number 1126001

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Assistant Prof. Dr.-Ing. Stefan Schulte
Assistance: Olena Skarlat, MSc

Vienna, 10th February, 2017

Kevin Bachmann Stefan Schulte

Erklärung zur Verfassung der Arbeit

Kevin Bachmann, BSc
Lerchenfelder Straße 46/1/4, 1080 Wien

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, 10. Februar 2017

Kevin Bachmann

Acknowledgements

At this place I want to use the opportunity to thank my advisor Assistant Professor Stefan Schulte and co-advisor Olena Skarlat for the invaluable technical support, constructive feedback, and honest comments throughout the actually pretty short time period of about six months. Precisely this short period of time often made great demands of my advisors that enabled me to finish my thesis in this time.

Beside the professional environment, I would also like to thank my friends for their moral encouragements and my fellow students for the endless technical discussions and valuable feedback. Finally, I would like to express my deepest gratitude to my family for their financial and moral support.

Kurzfassung

Die stetige Digitalisierung und Vernetzung von alltäglichen Gegenständen führte in den letzten Jahren zu einem enormen Wachstum an datenproduzierenden Geräten. Das resultierende Internet der Dinge (engl. Internet of Things, IoT) und dessen Verbreitung erfordern eine Veränderung in der Verwendung eben solcher Geräte, um eine erfolgreiche Datenübermittlung und Datenverarbeitung zu ermöglichen.

Aufgrund der Tatsache, dass diese Geräte nicht nur Daten generieren, sondern auch Rechen- und Speicherkapazitäten aufweisen, entstand ein neues IT-Paradigma. Dieser dezentralisierte Ansatz verlagert den Fokus von der zentralen Cloud-Computing-Umgebung and den Rand des Netzwerks und führt zur Einführung einer neuen Zwischenebene - dem *Fog Computing*. Die zentrale Idee von Fog Computing besteht darin, bestehende Ressourcen am Rande des Netzwerks zu nutzen, indem IoT-Dienste auf verfügbaren Geräten ausgeführt werden.

Die Entwicklung einer dynamischen Fog-Computing-Softwareumgebung birgt folgende Forschungslücken, die es zu lösen gilt: dezentrale Datenverarbeitung, Ressourcenvirtualisierung, IoT-Geräteverwaltung und die Bereitstellung von Diensten und Ressourcen sowohl in der Cloud-Umgebung als auch in der Fog-Landschaft. Das Schließen besagter Forschungslücken resultiert in einer Softwareumgebung, fähig die dynamischen Ressourcen am Rande des Netzwerks zu nutzen, den Datentransfer zu verringern und somit die Cloud zu entlasten und das IoT zu unterstützen. Eine derartige dynamische Softwareumgebung ist ein *Fog-Computing-Framework*.

Der aktuelle Stand der Technik konzentriert sich hauptsächlich auf Cloud Computing, Fog-Computing-Konzeptarchitekturen und Ansätze zur cloud-basierten Ressourcenbereitstellung. Da sich nur wenige wissenschaftliche Beiträge mit der Entwicklung eines konkreten Fog-Computing-Frameworks befassen, werden in dieser Arbeit IoT-Anwendungsfälle, verwandte IoT-Frameworks und Best-Practices im Bereich verteilter Systeme analysiert. Basierend auf dieser Analyse wird ein dynamisches und erweiterbares Fog-Computing-Framework entwickelt.

Das entwickelte Fog-Computing-Framework schafft die Vorraussetzungen für die Verwaltung zugehöriger IoT-Dienste in einer realen Fog-Landschaft. Darüber hinaus ermöglicht das Framework die Verwaltung der Geräte, Kommunikation zwischen den Geräten und realisiert die Bereitstellung von Ressourcen und IoT-Diensten in der Fog-Landschaft. Zusätzlich ist das Framework in der Lage, auf verschiedene Systemereignisse, z. B. Gerätebeitritt, Geräteversagen und Geräteüberlastung, zu reagieren. Die Evaluierung des Frameworks befasst sich neben der Handhabung besagter Ereignisse mit der Analyse weiterer Metriken, z. B. Kosten, Ausführungszeiten und variierenden Dienstanfragen. So bietet das Framework die Möglichkeit, die Dynamik der Fog-Landschaft zu bewältigen und führt zu geringeren Bereitstellungszeiten von IoT-Diensten und erheblichen Kostenvorteilen.

Schließlich erlaubt die Gestaltung des Frameworks zukünftigen Forschern, die Komponenten des Frameworks so zu konfigurieren, zu erweitern und zu verbessern, dass es den individuellen Anforderungen und Forschungsproblemen gerecht wird.

Abstract

The prevalence of ubiquitous computing devices encouraged by the expanding technology trend of the Internet of Things (IoT) demands a change in how these devices are used and where the generated data is processed. Since these computing devices not only generate data but feature computational and storage capabilities, a new paradigm evolved. The decentralized computing paradigm that shifts the focus of interest away from a centralized cloud computing environment towards the edge of the network is called *fog computing*. The main idea of fog computing is to utilize the processing and storage resources at the edge of network by deploying IoT services on available edge devices to reduce latency and processing cost.

The research challenges to be tackled in order to develop a dynamic software environment realizing the vision of fog computing are decentralized data processing, resource virtualization, service deployment, IoT device orchestration, and resource provisioning in both the cloud environment and the fog landscape. Such a dynamic decentralized software environment is a *fog computing framework*.

State of the art approaches are mostly focusing on cloud computing, conceptual fog computing architectures, and cloud-based resource provisioning approaches. Only few contributions deal with the development of a concrete fog computing framework. Therefore, in this work, IoT use cases, related IoT frameworks, and best practices in the area of distributed systems are analyzed in order to design and implement a dynamic, extensible, and flexible fog computing framework.

The fog computing framework provides the tools to manage IoT services in the fog landscape by means of a real-world test-bed. Furthermore, the framework facilitates the communication between the devices, fog device orchestration, IoT service deployment, and dynamic resource provisioning in the fog landscape. In addition to these main functionalities, the framework is able to react on various system events, e.g., device accedence, device failure, and device overload. The consequent event handling and assessment of other important metrics, e.g., costs, deployment times, and service arrival patterns, is performed in the evaluation of the framework. As a result, the framework provides the utilities to deal with the dynamism of the fog landscape and yields lower deployment times of IoT services and considerable cost benefits.

Finally, the design of the framework allows future researchers to configure, extend, and enhance the components of the framework to fit the individual requirements and research challenges.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Being an established paradigm, cloud computing allows customers to acquire on-demand computational and network resources on a pay-per-use basis and enables vertical and horizontal scalability [44]. In order to efficiently handle physical resources in cloud environments, cloud computing uses the concept of virtualization. Virtualization creates isolated virtual versions of physical resources enabling dynamic resource sharing amongst customers and applications [60]. Theoretically infinite virtual resources offered by the cloud along with a variety of distributed applications became a well-accepted practice in various application domains.

The prospering technology of the Internet of Things (IoT) leads to the increased use of IoT devices in various areas of private and industrial spaces. Such an expansion of the IoT and according applications, i.e., IoT services, requires changes in the existing approach of executing IoT services in a centralized cloud [8]. These changes result from the characteristics of data generated by IoT devices and according data processing services [15]. The required characteristics evolve from special IoT applications, e.g., augmented reality, smart grids, smart vehicles, and healthcare. An essential characteristic of IoT services is the necessity of a very low and predictable latency for communication purposes. Another noteworthy feature to be accounted for is that IoT devices are highly geo-distributed and require location aware large-scale distributed control systems [15, 16].

Despite this nature of the IoT, cloud computing is still seen as the central computational backbone of the IoT. However, cloud computing is not able to comply with the stated characteristics due to the lack of location-awareness, high latency, and missing geo-distributed data centers close to the IoT devices. As IoT devices feature their own computational and storage resources, it is a promising approach to virtualize these resources and use them to execute IoT services [16, 61]. Having in mind that the amount and capabilities of IoT devices is constantly increasing, neglecting these resources is ineffective and leads to processing cost in the cloud and considerable delays. To enable the utilization of those resources at the edge of the network, a novel distributed computational paradigm is required. Such a distributed paradigm is *edge* or *fog computing*. In the remainder of this thesis, the term fog computing is used.

Fog computing aims to reduce the cloud involvement by filtering and preprocessing data produced by the rising number of sensors and other IoT devices connected to the fog computing environment, i.e., the *fog landscape*. This improved utilization of edge resources leads to faster communication and task execution. A particular research challenge is to provide mechanisms for decentralized processing of data and for virtualized resource provisioning in fog landscapes. To address this challenge, a dynamic software environment, i.e., a *fog computing framework*, is required. This framework has to provide the functionality to execute, monitor, and analyze IoT services in addition to the deployment and migration of the virtualized IoT resources. Furthermore, in case of failure or violated service constraints, the framework has to be able to reconfigure the fog landscape.

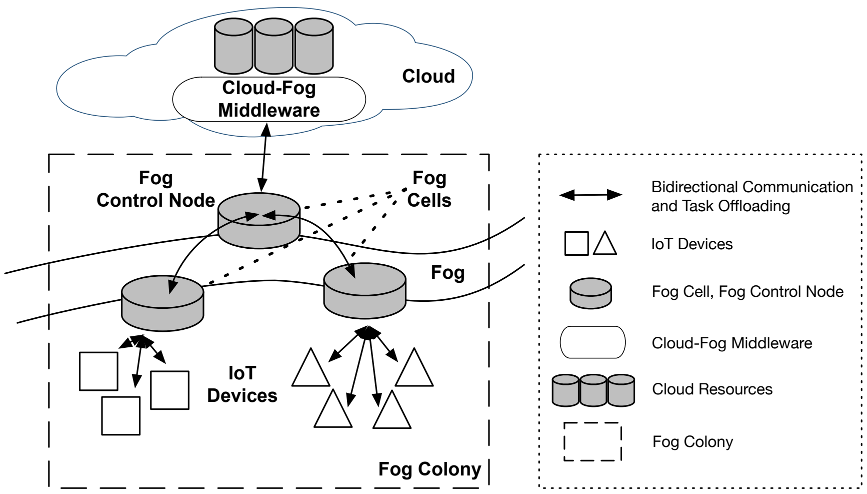


Figure 1.1: High-Level View on a Fog Computing Framework Architecture

1.2 Aim of the Work

The objective of this thesis is to implement a fog computing framework that is a real-world test-bed and serves as an environment to execute, monitor, and analyze IoT services in a fog landscape. The fog computing framework consists of three levels of communications: between IoT devices and the fog landscape, within the fog landscape, and between the fog landscape and the cloud (see Figure 1.1).

IoT devices which do not possess any computational or storage capabilities, e.g., sensors and actuators, are located at the bottom of the architecture. These IoT devices are connected to the fog landscape, i.e., to *fog cells* and *fog control nodes*. Fog cells are software components deployed and running on rich IoT devices that feature computational and storage resources and are able to process tasks. If a specific fog cell is not able to process the incoming tasks itself or the data needs to be stored in the cloud, e.g., for Big Data analysis, the fog cell propagates those tasks to a fog control node or the *cloud-fog middleware* for further processing.

Such a hierarchical construct of IoT devices, i.e., fog cells connected to a fog control node, is called *fog colony*. Fog control nodes are either directly connected to the cloud via a cloud-fog middleware, or to another fog colony via its fog control node. This forms a hierarchy of fog colonies. The task of the fog control node is to orchestrate the subjacent fog cells and to perform dynamic resource provisioning and service placement. If no cloud connection exists, fog colonies have to be able to work autonomously.

As already mentioned, the component on the top of the framework architecture is the cloud-fog middleware. It is responsible for processing task requests from connected fog colonies and managing cloud resources [56].

The major challenge of this diploma thesis is to develop a framework that enables the management of available resources in a fog landscape as described above. This includes the creation and destruction of virtual resources, providing a seamless on-demand task execution environment. The resulting environment is expected to improve edge resource utilization and to contribute to the extensibility and flexibility of the paradigm. This is needed for the future integration of not yet excessively researched fields as security, privacy, and reliability in the fog computing context.

Summarizing the described challenges, open questions are identified, which are of primary interest in order to implement a fog computing framework. These topics define specific problems to be solved in this thesis.

Requirements Analysis To develop a fog computing framework, a preliminary study on current deployment mechanisms in the cloud and fog has to be performed. This will allow to identify shortcomings of existing related work and to specify concrete software requirements for the framework. The analysis has to be focused mainly on the topics: cloud and fog computing architectures, component specification, communication, data storage, service placement, and service deployment. In order to enable a convenient technology selection for further work, the functional and non-functional requirements of the framework have to be elicited. The corresponding research question can be formulated as “What are the functional and non-functional requirements for a fog computing framework?”

Fog Computing Framework Architecture The next step is the design of the fog computing framework. During this step, a data model, architecture, and interfaces between the framework and its components have to be specified. The research question addressed here is: “What are the necessary components of a fog computing framework, and how can they be described in terms of their functional and technical specification?" An important consideration when developing a software design is to keep the framework loosely coupled, and thereby enable replaceability of the various components due to clearly defined interfaces. The assessment of suitable technologies has to be done according to the requirements analysis, i.e., the functional and non-functional requirements of the framework.

Implementation of the Framework The framework implementation is the realization of the functional and technical specifications defined in the previous step. The project will be divided into separate packages and sub-projects according to the specified components. Such a separation improves the replaceability and maintainability of the project by the means of specified interfaces between different components. The source code has to be readable and self explaining, hence state of the art code standards and guidelines have to be applied. The according research question is: "How to realize a fog computing framework that manages the fog landscape and executes IoT services?"

Evaluation of the Framework The last step in the thesis is an extensive evaluation of the developed framework. The research question of this part of the work is "How does the implemented fog computing framework improve the execution of IoT services compared to the execution in the cloud?" The objective of the evaluation is to show how the execution of IoT services in the fog landscape by the means of the developed fog computing framework differs from the execution in the cloud. Important metrics in the evaluation are amount of tasks deployed in the cloud and the fog environment, service deployment times, resource utilization in terms of RAM and CPU, and cost of execution.

1.3 Methodology and Approach

The procedure on how to complete this thesis is divided into three steps. In the boundaries of the individual steps, self control loops will help to estimate how the work conforms to the defined requirements.

Analysis of Existing Approaches The first step is gathering information about current research materials in the fields of cloud computing, fog computing, and the IoT in combination with resource provisioning, service placement, and task offloading. After the systematic gathering of the materials is done, the relevant information has to be extracted and analyzed to build the theoretical state of the art background for the design of the framework. The extraction of the information will be done by looking for predefined topics mentioned in Section 1.1. Further, an analysis of functional and non-functional requirements has to be performed.

Architecture Design As a next step, the design of the fog computing framework architecture is done. The design is a crucial point in the development of a framework, and therefore has to be self-checked and reviewed throughout the whole development process. This will ensure the correctness of the design decisions. During the design a suitable resource provisioning approach, as well as the software architecture including technologies and patterns, will be selected. Technologies are assessed in regard to the functional and non-functional requirements of the framework in combination with the dependencies to already chosen tools. The same approach is applied for best practices, patterns, and general design decisions.

Implementation and Evaluation As the implementation will be done within a one person team and no specific predefined processes or workflows exist, some aspects of well-known software development processes are extracted and tailored to manage the workload. For example, the management of programming tasks will be performed by means of an online workflow visualization tool, e.g., Kanban Flow[1]. In the course of the implementation, the state of the art Oracle code standards[2] will be followed. As the evaluation and implementation are not strictly divisible, the output of the framework has to be reviewed iteratively. After the implementation is finished, an extensive evaluation of the fog landscape execution of IoT services has to be done to show the benefits of the implemented framework.

1.4 Structure

The thesis is structured as follows.

In Chapter 2, the background knowledge, needed to understand the full extent of the stated solution, is presented. The chapter starts off with the explanation of the IoT technology and gets more fine grained over the course of the pages. In every section the most important basics, driving forces, and applications of fog computing are described.

Chapter 3 covers the most relevant work done in related scientific areas. The related work is classified according to several criteria specified at the end of the chapter. After the summarized work is presented, the results are discussed and compared.

Chapter 4 introduces the requirements and the general design of the fog computing framework. This chapter is split into functional and technical specifications and defines the most important design decisions and functionalities of the framework.

Within Chapter 5 the implementation of the envisioned fog computing framework is provided. After the requirements of the separate software components, the installation and execution instructions are explained.

The implementation is then followed by Chapter 6 providing an evaluation of the framework. The holistic evaluation aims to depict the benefits gained by the implemented fog computing framework at hand.

Finally, Chapter 7 concludes the thesis with an outlook of the work done and gives an insight into the future work in this area.

[1]`https://kanbanflow.com/`

[2]`http://www.oracle.com/technetwork/java/codeconventions-150003.pdf`

CHAPTER 2

Background

This chapter covers the background knowledge of this thesis and gives a general overview on needed technologies enabling fog computing, and more specifically, enabling the fog computing framework as specified in the introduction chapter.

As a starting point, the well-known *Gartner Hype Cycle* [64] is examined, as some relevant background technologies are in a crucial phase in research. A hype cycle is a predictive graph that shows currently trending research topics and their actual phase in research. The hype cycle is separated into five different phases. It begins with an innovation trigger, followed by a peak of inflated expectations, trough of disillusionment, a slope of enlightenment, and finishes with a plateau of productivity. Gartner Inc. argues that most of the innovations are forced to pass through these phases during their lifetime.

The annual hype cycle report, additionally, presents a mega trend analysis [19] pointing out the major trends of the year. In this analysis, the fields of cloud and mobile information technologies start to move out of the disillusionment phase, whereas technologies such as the IoT, mobile *Data Centers (DCs)*, and digital workplaces are located around the peak of the inflated expectations. Consequently, cloud computing and mobile information technologies have already been well-researched and have gained importance in the market. On the other hand, technologies as the IoT, mobile DCs, and digital workplaces are facing a crucial phase with lots of challenges tied to research efforts [31].

Fog computing and related approaches, e.g., mobile cloud computing and mobile edge computing, are situated close to the IoT and mobile DCs. The statement resulting from this mega trend analysis is that these technologies are currently at the peak of expectations, where a realistic assessment of the complete extent of the topics is difficult. This peak is then followed by a discovery of connected challenges and current issues to be tackled until first applicable solutions are successfully researched [31].

Due to the proliferation of data-producing IoT devices, e.g., sensors, mobile phones, and wearables, the huge amount of data sent over the network is a growing challenge for future applications. Not only the data processing gets demanding, but as well the data transmission from the devices to suitable DCs. Being able to transmit and process the future abundance of data, received by widely distributed devices, a new computing paradigm is needed. This evolving paradigm is fog computing. Fog computing supports the centralized DCs by filtering, preprocessing, and caching data, beside executing task requests at the edge of the network. The envisioned implementation of a dynamic, extensible, scaling, and fault-tolerant system to execute IoT services in a fog landscape, results in a fog computing framework. A fog computing framework provides the means to tackle the mentioned challenges of future data and device abundance [23].

The following sections comprise the most important technologies enabling the design and implementation of the fog computing framework. In Section 2.1, the prospering IoT technology, enabling novel smart applications, is described. Section 2.2 gives the basic background knowledge on cloud computing, including the empowering virtualization technology. Section 2.3 provides the basic knowledge about the crucial fog computing paradigm and related distributed computing paradigms. Section 2.4 introduces the general functionality of software frameworks and presents two related IoT frameworks. Finally, Section 2.5 briefly presents the most important aspects regarding resource provisioning.

2.1 The Internet of Things

The IoT is a currently prospering technology trend discussed all over the world. The term "the Internet of Things" was first used in a presentation at Procter & Gamble[1] in June 1999 [7]. In this presentation, Kevin Ashton combined Radio-Frequency Identification (RFID) with the, at that point in time, up and coming *Internet* resulting in the term: the Internet of Things. The vision of the IoT is to connect huge amounts of intelligent *things* to the Internet. These intelligent things can range from sensors, actuators, wearables, mobile phones over one-board computers to micro DCs. In the following, these things are called IoT devices. In order to connect the mentioned IoT devices to the Internet many technologies are applied, e.g., WiFi, Bluetooth, RFID, and Near Field Communication (NFC) [8]. The last two mentioned technologies, RFID and NFC, both are used in close proximity to IoT devices to identify, track, or authenticate them [31].

Looking back reveals that RFID was the first technology to draw attention to the IoT [7, 10]. Even nowadays RFID is one of the most pushing forces with respect to the appraisal of IoT applications. That is the case as the overall vision to connect everything to the Internet is straightforwardly applicable by so-called RFID tags. RFID tags are small uniquely identifiable and programmable microprocessors connected to an antenna to communicate with RFID reading devices. Additional components, e.g., batteries, GPS sensors, can be appended depending on application specifications [67, 10]. Tags can be placed on every imaginable item, e.g., food packaging, shippable products, and even animals or human bodies. The tagging of objects enables to identify, track, or authenticate them [8]. Examples of RFID tag applications are library systems, logistic product tracking, and access control systems. Aiming at a huge coverage of RFID tags on

[1] `http://us.pg.com`

everyday items, the amount of needed tags is very high. With the huge amount of tags required, the price becomes a crucial point of interest. Consequently, the rapidly falling prices of RFID tags are a positive force in increasing the interest towards applications using RFID [31, 58].

Beside RFID, the approach of Wireless Sensor Networks (WSNs) is an interesting related technology concerning the future trend of the IoT. Although WSN and the IoT seem to be competing technologies, many synergies exist. Sensor networks in a WSN consist of wireless interconnected sensors able to communicate. A common communication approach used in WSNs is the Peer-to-Peer (P2P) technology. Sensors need to discover other sensors in close vicinity, and build a network to intercommunicate in a highly geo-distributed sensor mesh. Examples for such WSNs are heat and smoke sensors in fire endangered forests. In case of fire or smoke, sensors send an emergency call either to a control middleware to recheck the emergency, or directly to a fire department [8, 62].

Another arising technology concept that is currently discussed in almost every company with manufacturing background is *Industry 4.0*. The IoT is closely related to the concept of *Industry 4.0*. Industry 4.0 is the vision of digitalizing and automating the manufacturing process starting from the first draft of a product over the complete supply chains to the final product. This includes the fields of Big Data analytics, digitalization, manufacturing, self-adaptation, artificial intelligence, etc. [41].

In the current situation, the IoT consists not only of sensors and actuators but, as previously mentioned, of a lot of heterogeneous devices connected to the Internet. This stresses the necessity of a standardized communication technique between IoT devices in order to control, monitor, and extract the accrued information. Additionally, energy efficiency and computational resources have to be taken into account. These novel requirements and application possibilities attract academies and industries to invest in the research of the IoT. Hence, many well-known companies like Gartner Inc., Siemens AG, SAP SE are researching to enable the realization of their own visions of the IoT [8, 10, 23].

Driving forces are crucial to novel technologies. In the case of the IoT, the most important driving forces are the decreasing cost of processing and storage power, the proliferation of IoT devices, elastic Big Data analysis in the cloud, and the convergence of the digitalized and the operational industrial world [5]. Furthermore, one of the more obvious driving forces is the usability gained by the new possible applications, e.g., smart cities, smart homes, smart cars, smart grids, smart healthcare, smart logistics, autonomous cars and robots, and virtual and augmented reality applications [8, 62]. Regarding smart homes, the functionality to control a complete house with a smartphone, tablet or wearable including the ability to close windows, change the temperature, monitor what products are in low supply, or observe elderly people, fits for the demands and activities of the future [62].

In a forecast, done by Cisco Systems Inc. [62], the extent of the IoT device proliferation from the early 2000s until 2020 is depicted. Here the incredible growth of smart objects, i.e., IoT devices, from 6.3 billion in 2003 to approximately 50 billion in 2020 points out the extraordinary development of this sector [18].

The applications emerging of this steadily growing trend are distributed over diverse areas. In lots of application scenarios involving the IoT the word *smart* appears, indicating a self-adaptive approach. A self-adaptive system readjusts itself dynamically according to changes in the environment. In more detail, self-adaptation unifies the topics of self-healing, self-management, self-configuration, and self-optimization [34].

In conclusion, the IoT is a very promising novel technology that is gaining a lot of interest and research attention in many diverse areas. The possible applications and resulting benefits make this technology very powerful. Finally, the visions and evolving technologies coming with the IoT do not only affect specific products or industries, but are able to substantially change the view and requirements towards systems and applications [21].

2.2 Cloud Computing

Cloud computing is an extensively researched centralized computing paradigm [44] that places emphasis on the dynamic provisioning of computational and storage resources. These resources are located in centralized DCs. The selection of the DC location is thereby tied to multiple factors, e.g., ambient energy cost, temperature, and land prices. The resources provisioned by cloud providers include software services used via a web browser, developer platforms to create and deploy cloud applications, and complete server infrastructures handling *Virtual Machines (VMs)* running on cloud resources. These three service models of cloud resource delivery are called *Software-as-a-Service (SaaS)*, *Platform-as-a-Service (PaaS)*, and *Infrastructure-as-a-Service (IaaS)*. Beside these different service models, four different cloud deployment models have emerged: (i) private cloud, (ii) community cloud, (iii) public cloud, and (iv) hybrid cloud. The details of these deployment models are presented in Subsection 2.2.3. Aiming at the dynamic provisioning of huge amounts of on-demand resources on a pay-per-use basis, cloud providers use the *Virtualization* technology described in Subsection 2.2.4.

In the following, the cloud computing definition by NIST [44] is described. The definition includes characteristics, service and deployment models (see Figure 2.1).

2.2.1 Characteristics

The essential characteristics of cloud computing were introduced in the definition of NIST in 2011 and still cover the most important aspects of it [44]. In the following paragraphs the mentioned characteristics are described.

On-Demand Self-Service Cloud services can be acquired on-demand and will be provisioned without any human commitment as services and resources are provisioned autonomously.

Broad Network Access As the cloud computing DCs are spread all over the world, every device connected to the Internet can interact with the cloud and acquire the proposed services. The devices include mobile devices, wearables, and proprietary servers.

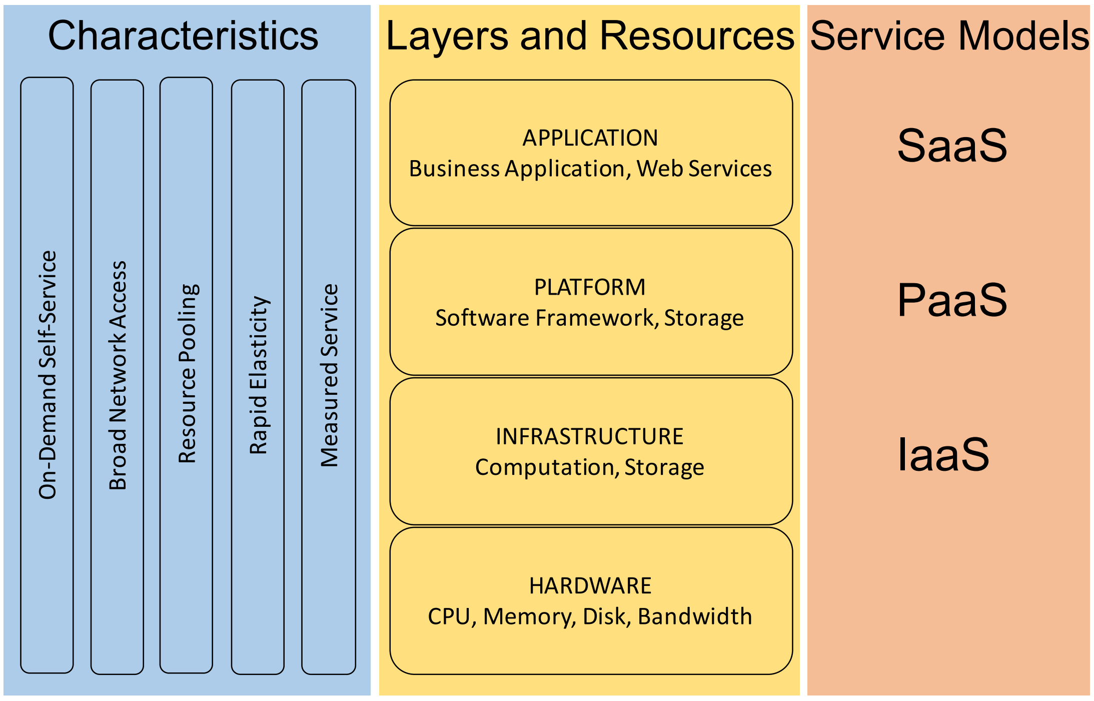


Figure 2.1: Cloud Computing Overview (adapted from [17])

Nowadays a lot of smart devices are connected to the Internet using cloud services. This is the case as lots of service providers already use centralized cloud resources, e.g., Netflix, Airbnb, Slack, to benefit from the dynamic resource handling and the virtually infinite resource capacity [9].

Resource Pooling Cloud computing customers want to acquire dynamic resources in the constellation of the current demand in order to minimize over and underprovisioning. Overprovisioning happens when the allocated resource capacity exceeds the current demand, hence, resources are wasted. Underprovisioning, on the other hand, is the scenario when there are too little resources available to serve the customers' demand. Clearly, one would like to avoid both situations and provide an optimal amount of resources.

With the resource pooling concept it is possible to utilize the physical resources according to the demand of customers. In Figure 2.2(a), a traditional DC with static resources can be seen. Traditional DCs have static resources as the proprietary servers cannot dynamically adjust without changing the infrastructure. On the opposite, Figure 2.2(b) shows a cloud computing DC with the capability to dynamically adjust the allocated resources according to time-dependent demand [12].

Aiming at an elastic resource environment the cloud consists of a pool of resources, i.e., servers with processing, and storage resources that can be configured to fit the requirements of demanding customers. The physical resources can be specified and managed easily due to resource virtualization. The physical servers thereby are used simultaneously by multiple costumers, i.e., in a multi-tenant way. The aim of having a virtual pool of resources is to obfuscate heterogeneous aspects in order to ease the usage and composition of generalized resources [6, 72].

Rapid Elasticity A very essential characteristic of the cloud computing paradigm is the rapid elasticity of the cloud. As the pool of physical resources is used via a resource virtualization technology, the resources can be scaled horizontally and vertically with minimal configuration and management effort. Horizontal scaling is the deployment of additional VM instances to simultaneously serve more clients, whereas vertical scaling is the adaption of specific capabilities of existing VMs. With these dynamic scaling capabilities the demand of customers can be handled more efficiently. Consequently, the energy consumption, wasting of resources, and overall cost can be decreased [6, 12].

Measured Service As cloud services are used on a pay-per-use basis, the complete communication and interaction is measured and monitored. This does not only serve billing purposes, but also enables cloud providers and customers to measure the *Quality of Service (QoS)*. Before acquiring cloud resources, the customer and cloud provider agree upon specific quality metrics of the service, i.e., *Service Level Agreements (SLAs)*. If these agreements are violated, the cloud provider may have to pay a penalty fee defined in the SLAs beforehand [6, 71].

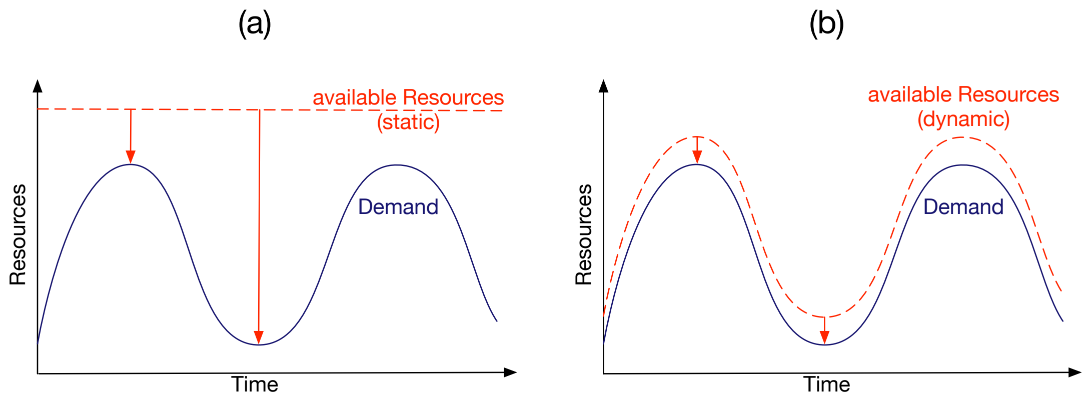


Figure 2.2: Resource Provisioning in a Traditional DC Versus a Cloud Computing DC

2.2.2 Service Models

The service models describe how cloud resources located in the cloud environment can be acquired and used. The location and ownership of resources is not considered in these differentiations [12, 44].

Software-as-a-Service The first model is called SaaS and is the most restricted possibility. SaaS customers access the service either via a web interface using a web browser or a programming interface. The physical cloud resources of the service cannot be managed or specifically configured by the customer. Examples for SaaS are Microsoft Office 365[2], and Google Apps[3].

Platform-as-a-Service A more flexible model specialized for developers, is PaaS. In this model, a development framework enabling developers to develop, test, and deploy applications in the cloud environment is provided. Aiming at a flexible software environment, the cloud deals with the management and control of the required physical resources. In addition to the general management and control of the resources, the cloud handles the dynamic scaling to ensure that the agreed SLAs are not violated at any point in time. Examples for PaaS are Google App Engine[4] and Windows Azure[5].

Infrastructure-as-a-Service The last and most flexible service model, is Infrastructure as a Service. Here, the customer has complete control over the VM running on physical cloud resources. The VM can be configured with the needed specifications as RAM, CPU, storage, and network functions. On this VM the user is able to deploy any kind of applications on a chosen operating system. Nevertheless, the user does not have complete control over the physical resources, but only on the acquired VM. Examples for IaaS are *Amazon Web Services*[6] and Open Stack[7].

2.2.3 Deployment Models

The previously introduced service models are hosted in several different deployment models enlisted here [12, 44]:

Private Cloud The private cloud is an approach where an organization uses a cloud environment explicitly dedicated to that very organization. The ownership, management, and operation of cloud resources can be handled by the organization itself, by a third party organization, or by a combination of the two of them. The advantage of this model

[2]https://outlook.office365.com
[3]https://apps.google.com
[4]https://cloud.google.com/appengine
[5]https://azure.microsoft.com
[6]https://aws.amazon.com
[7]https://www.openstack.org

is that the data sent to the private cloud is only stored in the environment used for that specific organization, meaning, sensitive data is not shared with any other organization associated to the environment. On the other hand, if that specific private cloud fails the data could be locked in this single point of failure.

Community Cloud The community cloud is based on the same idea as the private cloud, but the cloud environment is dedicated to the use within a specific community. Also, the ownership, management, and operation can be carried out by one or more organizations forming the community.

Public Cloud In a public cloud, the cloud environment is not dedicated to a specific organization or community. Hence, it can be used by the general public. The ownership, management, and operation can be handled either by the government, academic organizations, any other organization, or a combination of them. The advantage of the public cloud is the open accessibility for everyone without any restrictions to specific organizations. The negative impact of the open accessibility, consequently, is data and service security.

Hybrid Cloud The hybrid cloud is a combination between a private and public cloud. Hence, some cloud resources can be owned, managed, and operated in a private cloud, while others are used via an open accessible public cloud. For instance, sensitive data could be stored in a private cloud and more general non-critical data in the public cloud.

2.2.4 Virtualization

In a nutshell, virtualization describes the abstraction of the physical hardware resources of a computer system. The concept of virtualization is one of the major driving forces of cloud computing out of the following two reasons. First, the physical hardware can be shared across multiple customers in an isolated way. Thus, the advantages are that a single physical hardware system can serve multiple customers as a basis of their processing, storage and memory requests utilizing most of the machines hardware resources. This enables an efficient use of the resources at hand and eliminates locking a complete physical machine for a single customer.

There are different virtualization concepts [11], but in the course of this thesis only *full virtualization* and *operating system virtualization (container)* are considered.

Full Virtualization The abstraction of a full virtualization results in a virtual copy of the physical resources packed in a VM. Before deploying a VM, the user needs to specify the amount of CPUs, RAM, and other capabilities the VM should possess. Evidently, the amount of VMs created on one machine is restricted by its physical resources [11]. As the VMs running on the physical resources are completely isolated via a coordinating middleware, i.e., *Virtual Machine Monitor (VMM)*, the systems remain completely separated. Second, heterogeneous hardware components can be merged into a homogeneous resource pool. This resource pooling mechanism is used to offer diverse VM settings, e.g., CPU, RAM, and storage capacities [11, 60].

The complete virtualization of hardware resources, i.e., full virtualization, works as follows. Aiming at an isolated environment, the hardware is managed by the mentioned VMM. The VMM is responsible for monitoring, managing, and deploying VMs on top of it, preventing the direct hardware access to the physical hardware. Each VM consists of a separate operating system, i.e., a guest operating system, and arbitrary applications running on it (see Figure 2.3(a)). The advantages coming with this way of virtualizing the physical hardware resources are flexibility of deployment, simple provisioning of adaptable resources, and the ability to compose heterogeneous hardware. The negative aspects of this specific virtualization concept are the long starting times and the big amount of storage space needed. Both of these negative aspects are caused by the necessity of a separated operating system in every VM [11, 12]. A well-known enterprise, developing virtualization solutions, is VMWare[8].

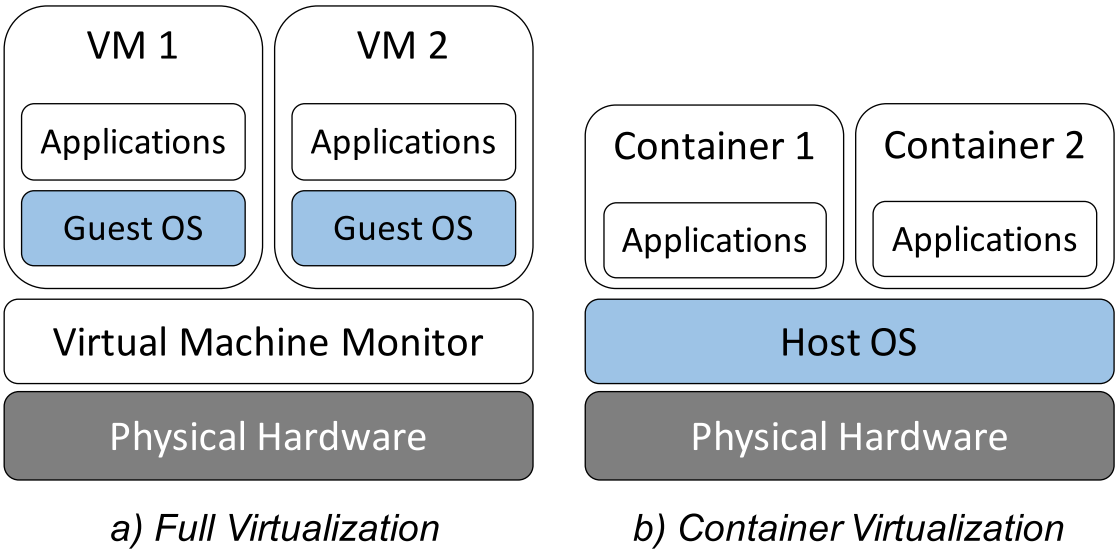


Figure 2.3: Virtualization Comparison (adapted from [12])

Operating System Virtualization (Container) In contrast to the previously mentioned virtualization concept, the operating system virtualization, or container virtualization, is a very light-weight virtualization concept. The container virtualization concept is built on top of the existing host operating system. The logically separated virtual environments, i.e., containers, run on top of the same host operating system and use the same kernel, and general physical hardware. Although these containers are running

[8]`http://www.vmware.com`

on the same operating system, the virtual environment is completely isolated. In other words, the container can just access storage spaces and processes associated to its own container (see Figure 2.3(b)) [12].

One of the most used container technology solutions is Docker[9]. Docker differentiates between Docker Images and Docker Containers. A Docker Image consists of several layers saving a snapshot of a Docker Container. Docker then uses the *union file system* [68] to merge these layers into one image. This image can be instantiated using the Docker runtime and results in a deployed Docker Container. The Docker Container consists of a base image (light-weight operating system), user-added files, and meta-data. This Docker Container can be started, stopped, or a snapshot, i.e., Docker Image, can be saved and used to deploy identical containers. Additional Docker orchestration tools to improve the capabilities of simple Docker Containers include, for instance, Docker Machine, Docker Swarm and Docker Compose[10]. Docker Machine helps to deploy Docker engines on the local machine, cloud providers or other DCs. Docker Swarm is used to cluster Docker Containers, and Docker Compose helps to run a distributed system consisting of multiple Docker Containers [57].

The advantage of containers in comparison to VMs is that containers can be started in no time as they do not need a separate operating system for every container. Containers are light-weight and can be deployed, released, and updated very fast as due to the special union file system only certain layers have to be updated [57]. Consequently, in the case of a fog computing framework, containers better fit the needs.

Docker is often used in combination with the currently thriving *Micro Services Architecture (MSA)*. The MSA is a novel approach of designing small, light-weight, independent, distributed software components. One separate software component is called *Microservice* and is deployed in an independent container. Some of the most important benefits gained by microservices are: technology heterogeneity, resilience, scaling, ease of deployment, and optimization for replacement [45]. As microservices are communicating using standardized formats like *JavaScript Object Notation (JSON)* and *Extensible Markup Language (XML)*, the technologies are completely independent. The standard communication technology used in MSAs is *Representational State Transfer (REST)* [45].

2.3 Fog Computing

In the last few decades the computational models switched back and forth between a centralized and decentralized computing approach. Starting with a mainframe approach in the 70s and 80s the model evolution was followed by a wave of decentralization by the client-server model in the 90s. This first wave was triggered by the falling prices of personal computers and the rising interest of possessing proprietary computational power [39]. At the beginning of the 2000s the computational model again switched from a decentralized to a centralized approach, namely the cloud computing paradigm [33, 66].

[9]`https://www.docker.com`

[10]`https://docs.docker.com`

Although cloud computing is prospering and is not going to be replaced in the nearest future, there is a force pushing towards a novel decentralized approach to solve the immanent problems of centralized systems, e.g., high latency, missing location-awareness. The difference compared to the preceding paradigm is that this approaching paradigm will not supersede the former one but extend it in order to improve specific capabilities. This current shift from the centralized cloud computing paradigm towards a decentralized computation paradigm marked the birth of fog computing [61].

Fog computing was introduced as a new technology for the IoT by Bonomi et al. in 2012 [16]. This novel distributed computing paradigm comprises the idea of providing computational and storage capabilities closer to data-producing IoT devices at the edge of the network. With the objective to decrease the distance between the end devices and the closest processing unit, this paradigm introduces an additional layer of resource-rich IoT devices, i.e., fog cells [61]. These fog cells possess their own computational and storage capabilities to process task requests, filter, and preprocess data. This creates a one-hop distance to the end devices and therefore decreases the latency and task execution time. Another crucial extension of fog computing is the high geographical distribution of these additional devices, aiming at a seamless and reliable service execution even when connected to moving devices, e.g., smart cars, mobile phones [15, 16, 61].

In more detail, fog cells are located at the edge of the network, aiming to reduce the latency, task execution time, and the amount of data sent over the network. Fog cells then receive task requests from connected IoT devices, e.g., sensors and actuators, or external initiators and decide where to process these requests. Depending on the fog cells' capabilities, a reasoning component decides whether the task request is processed locally or propagated to the cloud [24].

The communication inside the fog computing environment, i.e., the fog landscape, is not restricted to any form of communication and can therefore take place via WiFi, cellular networks, Bluetooth, or ethernet [61]. Towards a redundant environment, the fog landscape can also be connected to multiple cloud providers or use private clouds inside the fog landscape.

Figure 2.4, depicts a graphical representation of the fog landscape. At the bottom of the architecture diverse IoT devices are located, followed by routing devices connected to fog cells. These routing devices symbolize the possibility to reuse existing network devices, e.g., routers. Furthermore, fog cells can either be connected to other fog cells, or to public or private clouds [24].

2.3.1 Characteristics

The main characteristics of the fog computing paradigm, helping the IoT to exploit its potential, are the following [16]:

Low Latency and Location-Awareness As fog cells, located at the edge of the network, reduce the distance between IoT devices and cloud resources, the latency and task execution time can be reduced drastically. Furthermore, due to the location-

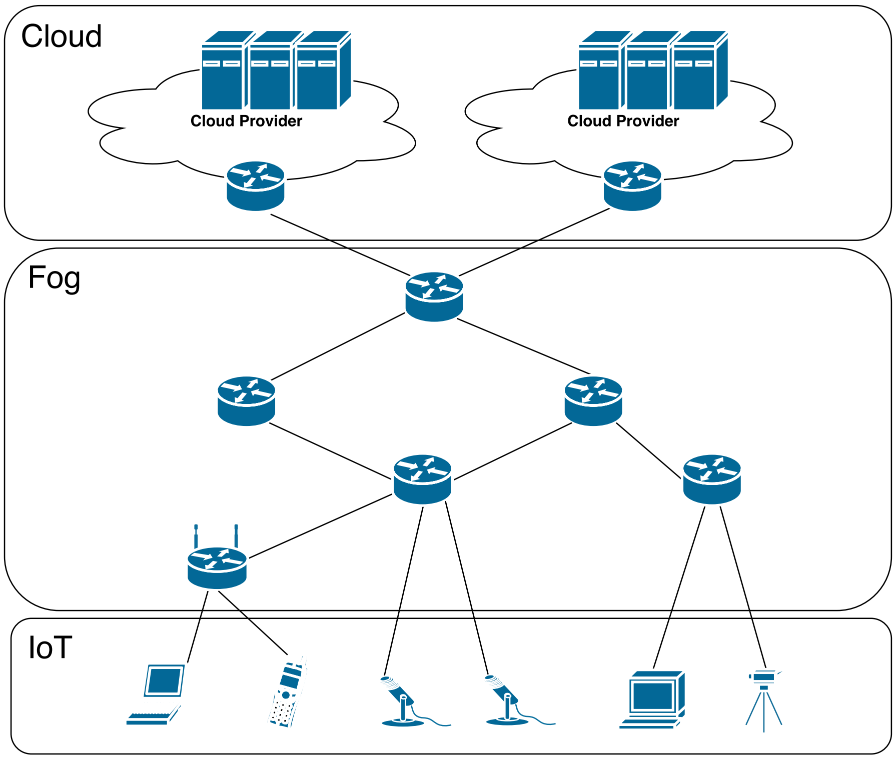


Figure 2.4: Fog Computing Landscape (adapted from [24])

awareness of fog cells, location aware services can be provided, e.g., caching of location dependent content. These services are able to preprocess, filter, or cache requested content and thereby further reduce the latency and task execution time [61, 16].

High Geographical Distribution IoT devices are highly geographically distributed. That is why the switch from centralized processing in the cloud to a decentralized processing in a fog landscape is needed.

Large-Scale Sensor Networks Large-scale sensor networks communicating with fog cells are one of the key scenarios of the fog computing paradigm. These sensors are able to send task requests to fog cells. Depending on the available fog cell resources, it either handles the requests itself or propagates them to other fog cells for further processing.

Mobility Support Beside the high distribution of IoT devices, one has to take into account the mobility of the participating end devices. Moving devices demand dynamic restructuring of the network topology according to the affected devices in the fog landscape. This is enabled via dynamic hierarchical fog landscape that allows to restructure the topology. Aiming at a high mobility support, the fog computing landscape needs to be able to communicate with mobile devices by mobile communication technologies, e.g., LISP[11] [16].

Device Heterogeneity Diverse IoT devices connected to the network come along with no standardized functionality, interfaces, or deployment types. Hence, many heterogeneous IoT devices need to be considered when handling requests in a fog landscape. The fog landscape aims to enable the communication between different types of IoT devices by the means of standardized communication and resource virtualization [16].

Being a very recent research topic, fog computing encounters a lack of methodologies and concrete solutions. Hence, still a lot of research effort is needed to enable the characteristics as stated above. An important association supporting and spreading the vision of fog computing is the *Open Fog Consortium*[12]. The principle of the Open Fog Consortium is to preserve an open fog computing architecture to enable the cooperative research with multiple organizations. The collaboration of diverse organizations, specialized in different areas, is a crucial point regarding the future of fog computing.

Fog computing is often mentioned as a provisioning technology for various applications of diverse fields, e.g., healthcare, augmented reality, caching, and preprocessing [25]. Being able to ensure the promising improvements arising with this paradigm, a lot of challenges still need to be tackled. Crucial challenges in research include resource provisioning, service placement, security and reliability, energy minimization, standardization, and programming models [25, 61].

Beside fog computing, two slightly different paradigms called *Mobile Edge Computing (MEC)* and *Mobile Cloud Computing (MCC)* have evolved [4, 29].

2.3.2 Mobile Edge Computing

MEC is based on a comparable idea as fog computing, but places an emphasis on mobile devices at the edge of the network. The key features of the entire MEC environment are task sharing between mobile devices and task offloading to the cloud. The mobile devices are able to intercommunicate and take task requests from each other. Additionally, so-called *MEC servers* are located at the edge of the network responsible for connecting the mobile devices with the cloud. In this approach, the edge of the network is the edge of the *Radio Access Network (RAN)*. Hence, the base stations of the RAN represent the MEC servers, i.e., the fog cells in fog computing. These MEC servers are owned by telecommunication companies and are able to process task requests [4, 14, 13, 50]. Due to the fact that most of the communication takes place in the RAN, the emphasized communication technologies in MEC are the cellular communication generations 3G, 4G, and 5G [47].

One of the major driving forces supporting the research in the area of MEC is the European Telecommunications Standards Institute (ETSI) [27]. This driving force,

[11]`http://www.lisp4.net/`
[12]`http://www.openfogconsortium.org/`

obviously, is connected to the heavy use of the telecommunication resources and further possibilities for the telecommunication industry.

Summarizing, the main goal of MEC is to reduce network latency and thereby improve the quality of mobile applications, e.g., augmented and virtual reality, on-demand video streaming, and mobile gaming [4].

2.3.3 Mobile Cloud Computing

MCC has emerged from the already well-researched technologies mobile computing and cloud computing. The driving force of this technology is a resource-rich cloud computing environment in combination with the proliferation of smart mobile devices. Although the processing power of modern mobile devices is increasing, there is still no satisfactory solution regarding the battery lifetime and data storage capacity. Hence, resource-intensive tasks, e.g., image processing and natural language processing [29, 52], need to be processed in the cloud to save energy and storage capacity. However, as mobile phones with unused processing units are ubiquitous, the efficient utilization of these resources is essential [29]. Currently, a huge amount of processing power is wasted by neglecting idle mobile device processors. Especially, during specific time periods, e.g., in the night, many resources could be used without conflicts.

The general idea of MCC is to offload compute-intensive tasks, reduce power consumption, and save storage space by propagating the data to the cloud, mobile cloud, or powerful IoT devices [26, 29, 52].

Three different types of researched MCC environments exist. First, a mobile device offloads a task request to the cloud and receives a solution in response. Second, several mobile devices form a so-called *mobile cloud* and process task requests coming from other mobile devices in need. Third, a resource-rich IoT device, located at the edge of the network, processes task requests of the mobile devices and returns the solutions [29]. In the following, the latter type is considered when talking about MCC, as this is the most related type with respect to the fog computing paradigm.

Application scenarios of MCC include mobile healthcare, mobile commerce, and mobile learning. These diverse application scenarios focus on different aspects of the MCC paradigm. In a mobile healthcare scenario the data gathered by mobile devices plays a crucial role because the data could contain life saving information that needs to be processed instantly. In mobile commerce, on the other hand, the new communication patterns and attainment possibilities are the most important prospects [26].

2.4 Software Frameworks

A software framework represents a base structure for application development and service execution in a specific software environment. The reusable base structure serves the purpose to enable a user to concentrate on a certain task to be developed instead of implementing the environment basics. Software frameworks often are extensible and configurable according to individual problems users want to solve by means of their applications [49].

In more detail, software frameworks can be divided into white-box and black-box frameworks [49]. Black-box frameworks are ready-to-use frameworks that do not need to be extended or further developed to be executed. White-box frameworks, on the other hand, demand to be configured by extending specific interfaces in order to execute the framework in the chosen environment. The most common frameworks are a mixture between white-box and black-box frameworks. In this thesis, this mixture is considered when talking about frameworks [49]. In the following paragraphs related software frameworks in IoT environments are presented.

In the work of Vögler et al. [63] a scalable large-scale IoT framework is introduced. The focused environment of this large-scale IoT framework is a smart city. The framework LEONORE, developed in this work, serves as an infrastructure and toolset for the deployment of IoT applications on IoT devices at the edge of the network. These functionalities are enabled by a modular architecture with provisioning handlers, application package repositories, application package management, IoT gateway management, dependency management, load balancing, and many other components. The whole architecture is built on light-weight microservices based on MSA principles. Beside these architectural aspects, the work focuses on distributing mechanisms to provision applications at the edge of the network in order to save bandwidth between the edge and the cloud, and enable a distributed and scalable IoT service deployment. The framework is primarily tested on a simulated IoT environment in a private cloud by deploying Docker Containers. Some parts of the evaluation are tested on a real-world IoT deployment test-bed of an industry partner of the authors.

Kim and Lee [38] developed an open source IoT framework to develop, provide, and execute IoT services in open source IoT environments. Hence, the stakeholders are developers, service providers, platform operators and general service users. The architecture presented in this paper is based on five platforms: open *Application Programming Interface (API)*, planet platform, store platform, device platform, and mash-up platform. The planet platform serves the purpose of managing the IoT device registration and monitoring, while the mashup platform handles the services to collect the data received from registered IoT devices. The collected data and developed services are stored in the store platform. The device platform is an embedded software application including a tiny web server to enable low-level devices to connect to the RESTful API and thereby to the other platforms. For all of these platforms Kim and Lee developed web applications to enable a convenient interaction. The work presented in this paper serves as a framework for all kinds of stakeholders to develop, provide, and execute applications using web-GUIs but does not take into account specific aspects like large-scale IoT landscapes and distributed application provisioning.

2.5 Resource Provisioning

Resource provisioning is a crucial topic in many diverse research areas. Obviously, it is important to use the resources at hand as efficiently as possible. In this context, efficiency is tied to various goals, e.g., optimization of cost, energy, runtime, and resource utilization. Resource provisioning is the procedure to orchestrate, allocate, deallocate, and monitor available system resources. These mentioned actions are crucial in order to enable an efficient and QoS aware resource management, i.e., resource provisioning. Hence, not only the time-dependent demand needs to be considered, but as well the QoS metrics have to be monitored and cross-checked with the SLAs. Depending on the workload fluctuation of the system, the resource provisioning procedure changes in complexity. With increasing fluctuations the procedure gets more complex because the resources need to be adapted increasingly [37]. In the following paragraphs the most important aspects of resource provisioning are described [37].

Monitoring System monitoring is needed to keep track of the status of running services and the software environment in general, e.g., CPU and RAM utilization, topology changes, failures. The gathered data is stored in a database for further processing. For the monitoring of a complete software environment, a system model is essential. It is crucial to find a necessary level of abstraction to model the heterogeneous components of the system [37].

Reasoning and Orchestration Reasoning and orchestration is the process where system monitoring information is analyzed and a resource provisioning plan is computed. A resource provisioning plan defines where specific services are deployed, and on which services the incoming task requests are processed.

Allocation and Deallocation According to the calculated resource provisioning plan, described in the previous paragraph, fog resources are allocated and deallocated.

The resource provisioning procedure can either be done in a *proactive* or *reactive* way [37]. A proactive scenario implies periodical checking of the system status, and according to the monitored data, the resource provisioning is done. Often, predictive mechanisms are applied to forecast future resource demands and to act before the system is stressed. A reactive provisioning scenario, on the other hand, deals directly with events like failures, new appearance of devices, watchdog events etc., and handles the provisioning accordingly. For instance, a watchdog component fires an overload event, meaning, a specific service CPU utilization exceeds a defined threshold, and the system reacts by deploying more instances of that very service [37].

Furthermore, the resource provisioning procedure can be done according to several different approaches. Depending on the optimization problem resulting of the system model, optimization goal, and further constraints, an adequate approach is chosen. The optimization problem can be formulated in terms of dynamic programming, more specifically linear programming, and can be solved either by exact mathematical methods, or by heuristic algorithms. Heuristics are algorithms close to real-world problems, that look for near-optimal solutions. In order to improve the solution of heuristics, often the problem scenario and environment need to be refined and described in more detail. Example heuristic algorithms are greedy algorithms, and local search [20, 51].

In a fog computing landscape, the resource provisioning is even more complicated compared to a cloud environment because the distributed heterogeneous devices need to be provisioned with minimal latency and task execution times. Additionally, a fog computing landscape contains a dynamic hierarchy. Thus, the device hierarchy can change during runtime and therefore is not available at the beginning of the service. Evidently, already very well-researched cloud computing resource provisioning approaches cannot be directly applied in a fog computing landscape. It remains to extract the general idea of these cloud computing approaches and use them for further specification in a fog landscape.

With the background knowledge, presented in this chapter, the reader should be able to understand the full extent of the concept, design, and implementation needs of the fog computing framework.

CHAPTER 3

Related Work

In this chapter, the most important related work is presented and discussed. Selected scientific papers are summarized and categorized in three major groups highlighting the main focus of the papers included in the literature review. After the related work is presented, some criteria to classify the papers in more detail are specified. This enables a clear view on the most important aspects of the diverse research papers compared at the end of the chapter.

3.1 Fog Computing Architecture and Concepts

This section provides a general overview on the current work done in the field of the fog computing paradigm including architectures, concepts, challenges, opportunities, and applications.

3.1.1 Initial Fog Computing Concepts by Bonomi et al.

A paper by Bonomi et al. [16] introduced the first concept of fog computing in 2012. In their work, the importance of a new paradigm of fog computing is explained. According to the authors, the main features of IoT environments which have to be accounted for are (i) low latency and location-awareness, (ii) high geographical distribution, (iii) large-scale sensor networks, (iv) mobility support, and (v) device heterogeneity. The proposed fog computing architecture aims to deal with those features and therefore consists of the following layers from the bottom to the top of the infrastructure: IoT devices, multi-service edge, where the fog cells are located, the core network, and the cloud.

The authors mention three main application scenarios suitable for the proposed architecture. First, connected vehicles, where cars intercommunicate with smart traffic lights and roadside units located along roads. The second mentioned scenario is smart grids. A smart grid depicts an intelligent power supply system with lots of widely geographically distributed suppliers and consumers. Third, wireless sensor and actuator networks consisting of widely geographically distributed sensor and actuator nodes. All three scenarios demand real-time processing, low energy consumption and a fault-tolerant distributed system, and would thereby profit by having a sophisticated fog computing solution.

In further research [15], the architecture gets more specified with a detailed description of functional and technical components and an abstract software architecture model. In this software architecture model a fog abstraction layer, fog service orchestration layer, and north-bound APIs are introduced. The abstraction layer creates an abstract version of the heterogeneous devices, enabling a more generalized service execution. The orchestration layer is responsible for service execution, task scheduling, and the deployment and destruction of software agents, i.e., fog cells. In order to enable the distributed communication between these software agents, north-bound APIs are specified. North-bound API indicates that a component is allowed to communicate with another component of a higher hierarchy level.

In another work, to which Flavio Bonomi contributed, the focus was on the performance improvement of web sites using the fog computing paradigm. In this paper, Zhu et al. [73] use a similar architecture and concept as in the aforementioned papers to improve the performance of web sites. This improvement is gained by using the fog landscape to reduce HTTP requests, minimize the size of web objects, and reorganize the webpage composition.

3.1.2 A Comprehensive Definition of Fog Computing by Vaquero and Rodero-Merino

Another idea regarding the architecture and the communication in an *edge cloud*, i.e., in a fog colony, is presented by Vaquero and Rodero-Merino [61]. They suggest using *Network Function Virtualization (NFV)* in combination with *Software-Defined Networking (SDN)* at the edge of the network. This suggestion aims at improving the flexibility and dynamic behavior of network services, as well as user services. NFV is a technology to virtualize network resources and to provide them according to the current demand. Hence, network resources as firewalls, databases, and routers can be deployed on-demand. The SDN is needed to create virtual networks and run software services on the deployed devices. This technique leads to a major flexibility improvement at the edge of the network.

Beside the adapted architecture, the authors address the topic of distributed management of devices in edge clouds. Within this topic, the P2P communication, between the devices in an edge cloud and between edge clouds, is mentioned.

Additionally, Vaquero and Rodero-Merino reveal a possible transition of trust, away from cloud providers towards edge clouds. Nowadays privacy is a critical issue, and fog computing leads to a crucial change concerning this topic. Fog computing enables a new privacy approach by not transferring critical data to a centralized DC but to a DC located at the edge of the network.

Finally, the work enlists current challenges in research including discovery and synchronization of applications, resource handling, distributed management, security, standardization, monetization, and programming models.

3.1.3 Focusing on Mobile Users at the Edge by Luan et al.

The work by Luan et al. [43] is another example for a concept using NFV in combination with SDN. The presented architecture consists of the following layers (bottom to top): 5G wireless, fog computing, NFV, and cloud computing. As this paper concentrates on mobile devices, the bottom most layer emphasizes on mobile technologies as 5G and WiFi. The structure of the network and the participating network devices are managed using a SDN approach. This enables the managing component located at a cloud to have a global network overview and to adapt the devices and services according to the current demand.

3.1.4 A Fog Computing Platform including a Real-World Test-Bed by Yi et al.

In the first paper of Yi et al. [70] they survey existing fog computing approaches and challenges, and summarize the most important facts. The mentioned approaches and challenges are comparable to the findings of Bonomi et al. [16] and Vaquero and Rodero-Merino [61].

In a follow-up paper [69] the authors introduce a simple three-layered architecture with the layers: user, fog, and cloud. The fog layer, i.e., *a fog platform*, consists of six components to handle task requests, service deployment, and network management. The stated components are: authentication and authorization, offloading management, location services, system monitor, resource management, and VM scheduling. The top and bottom layer (cloud and user), are somewhat similar to the aforementioned architectures.

After mentioning alike application scenarios as the previous authors, an experimental proof-of-concept solution is presented. The experimental fog platform consists of two fog sub-systems both connected to three separate servers in a *Local Area Network (LAN)*. These fog sub-systems are connected to a cloud (Amazon EC2[1]) via a *Wide Area Network (WAN)* connection. On both fog sub-systems separate Open Stack[2] environments are deployed. The evaluation of the test-bed is done by comparing the latency, bandwidth, VM migration performance, and the task execution time of a face recognition application. The results expose the significant improvement of the fog computing paradigm in comparison to the general cloud computing paradigm.

[1]`https://aws.amazon.com/de/ec2/`
[2]`https://www.openstack.org/`

3.1.5 Principles, Architectures, and Applications of Fog Computing by Dastjerdi et al.

A recent paper by Dastjerdi et al. [25] presents a layered fog computing architecture with ideas comparable to the work done in the previous papers. In their architecture a layer called *software-defined resource management* is introduced which is responsible for task scheduling, profiling, monitoring, and resource provisioning, and is located in between the cloud and the fog colonies. This newly introduced layer takes away the resource management responsibility from the fog control nodes. In addition to the divergent architecture concept, the authors mention important application sectors, and research directions and enablers.

The described applications scenarios are healthcare, augmented reality, and preprocessing and caching. In healthcare a fall monitoring system for stroke patients is the key scenario. Augmented reality is mentioned in combination with cognitive assistance using Google Glasses and brain interaction games. The last application scenario highlights the preprocessing and caching of data at the edge of the network. Programming models, security, resource management and energy minimization form the focused challenges in this work and get backed up by examples.

3.1.6 A Theoretical Fog Computing Model to support IoT Applications by Sarkar et al.

In this paper, Sarkar et al. [53] introduce a detailed theoretical model of a fog computing landscape. Like in many other papers, fog computing is presented as a supporting technology for IoT applications. First, a 3-tier architecture is introduced. Tier 1 represents IoT devices organized in clusters. Edge gateways, fog-cloud gateways, and fog instances are located in tier 2 followed by the cloud in tier 3. In the very detailed mathematical description of the model the authors take into account the status, type, and location of the IoT devices and define every possible state of the system and its components.

To measure the performance of the model, Sarkar et al. use two performance metrics: service latency and energy consumption. The performance evaluation is simulated on a specified scenario and shows the considerable improvements in comparison to a cloud computing approach.

3.2 Programming Models

In this part, the related work on fog computing programming models is examined. Regarding a programming model, it is important to investigate corresponding applications, as not every type of application fits in a fog landscape. The corresponding applications from the analyzed papers in this section are based on a distributed architecture and need to be separated into several independent microservices. Furthermore, the components have to be able to communicate via REST and need to be connected to the cloud.

3.2.1 A High Level Programming Model by Hong et al.

In the work of Hong et al. [35] a high level programming model is presented, including APIs and an easy resource provisioning approach based on the utility thresholds of fog resources. This high level programming model, i.e., *Mobile Fog*, is based on a PaaS approach, meaning, the model is hosted on a cloud provider and developers can implement, test, and deploy solutions on this platform. The presented model takes the responsibility for the dynamic scaling of the executed applications during runtime. The main design goals of the work at hand are the development of a high level programming model and the support of dynamic scalability.

An application of mobile fog is built upon mobile fog processes mapped to distributed computing instances. These instances are grouped in regions with respect to their location and network hierarchy levels. Every mobile fog process handles the requests of its dedicated region. A mobile fog process can be created by deploying an image, stored at a shared registry, adding an unique identifier to unambiguously identify the process.

The dynamic scaling of the environment is done by generating on-demand computing instances according to a specific scaling policy. This scaling policy depends on monitoring metrics, e.g., CPU utilization and bandwidth, which trigger the concrete scaling actions. To make dynamic transparent scaling possible, a spatio-temporal object store is needed to share application-wide data. Spatio-temporal storage means that a storage container stores and manages both space and time information.

Finally, the model is evaluated with a simulation using generated traffic patterns and randomly moving vehicles in a specified traffic area. The results of the evaluation show the enhancement of the model in comparison to a general cloud computing approach.

3.2.2 Incremental Deployment and Migration of Fog Applications by Saurez et al.

Equally important is the work of Saurez et al. [54] where the previous work of Hong et al. [35] is extended by implementing the envisioned APIs and adding algorithms regarding the discovery, migration, and deployment of fog cells and services.

The implemented model is called *Foglet programming model* and consists of a foglet runtime, and multiple foglets, i.e., fog cells. The foglet runtime consists of the following components: discovery server, Docker Registry server, entry point daemon and the worker processes. To create a new service a user has to save the developed Docker Image into the Docker Registry server with a system unique application key. After that, the application can be deployed via a application key, network region, network hierarchy level, resource capacity, and QoS parameters. The registry then starts the novel application in all available fitting foglet instances. The resulting foglet runtime contains four major capabilities: (i) automatic fog resource discovery and deployment, (ii) multi-application collocation, (iii) communication APIs, and (iv) resource adaptation and state migration.

Furthermore, an experimental real-world fog landscape was set up using Docker Containers deployed on several server instances to perform the evaluation.

3.3 Resource Provisioning

The most specific work classification in this chapter deals with resource provisioning. Resource provisioning is an extensively researched topic in many scientific areas, e.g., cloud computing and MCC. Some relevant papers in research tackling the resource provisioning problem in the environment of the mentioned areas are [55, 71] in cloud computing and [26, 42] in MCC. These works describe resource provisioning approaches, yet cannot be directly adopted for the usage in fog computing as fog landscapes consist of a dynamic hierarchy of heterogeneous IoT devices which have the characteristic to change during runtime. In addition to the dynamic behavior of the fog landscape, most of the papers in the cloud computing and MCC area do not consider maximizing the edge resource utilization in order to save cloud cost and decrease the network latency due to close-range communication. Hence, the general ideas of the approaches can be used to elaborate on resource provisioning approaches for fog landscapes.

3.3.1 A Resource Provisioning Approach for IoT Services in the Fog by Skarlat et al.

In the work of Skarlat et al. [56], along with the framework architecture, an optimization problem for resource provisioning in a fog landscape is introduced. The fog computing framework introduced in this paper consists of four different device types.

- resource-poor IoT devices, e.g., sensors and actuators, sending task requests to upper hierarchies
- resource-rich IoT devices, i.e., *fog cells*, to process task requests received from connected IoT devices
- *fog orchestration control nodes* to orchestrate fog cells and to handle the resource provisioning and task scheduling in associated fog colonies
- *cloud-fog control middleware* to act as a communication and control middleware for fog orchestration control nodes and cloud providers

The resulting hierarchies of IoT devices, fog cells, and fog orchestration control nodes are fog colonies.

In their work, the fog landscape is simulated by extending the CloudSim[3] simulation tool. In the evaluation, the default provisioning policies of CloudSim are compared to the enhanced approach, and the whole fog landscape execution of tasks is compared to the cloud execution showing the benefits of fog computing.

[3]http://www.cloudbus.org/cloudsim/

3.3.2 Dynamic Resource Provisioning through Fog Micro Datacenters by Aazam et al.

Aazam and Huh [1, 2] proposed a fog computing architecture based on a similar idea as has already been described in the introduction Chapter 1. The idea consists of three layers namely the cloud, the fog landscape, and IoT devices with different control nodes inside these layers able to orchestrate fog cells. The authors follow an approach of using micro DCs orchestrated by a smart gateway. Additionally, a detailed theoretical resource management model is included. The prediction of future resource demands is based on types of accessing devices, relinquish probabilities which are generated by historical access data, pricing models, and service types. This resource management model is not able to react to dynamic changes in the fog landscape. In their further work, Aazam et al. proposed an improvement of the theoretical resource management model in terms of specification of utilization and QoS in the context of multimedia IoT devices [3].

3.4 Discussion

In this section the considered research papers are analyzed based on the fulfillment of a selection of important criteria. In order to group and compare the research work appropriately, the criteria include general aspects, e.g., the base technology of the work, as well as specific topics tackled by the authors. The selected criteria for the classification of the related work are the following.

1. **The Internet of Things (IoT)**
   To fulfil this criterion, the research work needs to take the handling of the emerging IoT devices into account. Hence, at some point of the concept, IoT devices connected to the Internet need to be considered.

2. **Fog Computing Architecture and Concept (FC)**
   This criterion is only satisfied when the authors present an architecture and concept based on the general vision of the fog computing paradigm.

3. **Dynamic Topology (DT)**
   In order to satisfy this criterion, the paper needs to consider the dynamic restructuring of the system topology. In other words, the system needs to be able to adapt to the underlying topology during runtime. Static topologies defined before runtime, or topologies containing static centralized discovery services do not satisfy this criterion.

4. **Programming Model (PM)**
   The work needs to provide a programming model, stating how to implement a distributed system to improve comparable characteristics as mentioned in the introduction of a fog computing framework, e.g., latency, task execution time, edge resource utilization.

5. **Resource Provisioning (RP)**
   The authors have to provide a resource provisioning approach in a distributed environment to satisfy this criterion. A basic resource provisioning model is sufficient.

The above stated criteria were selected according to relevant research topics concerning a fog computing framework. As the IoT is a major driving force of fog computing it straightforwardly follows that it is an important factor when developing a fog computing framework. Hence, the IoT and IoT devices play an important role the related work needs to take into account. Another very important yet often neglected aspect is a dynamic topology. The criterion regarding dynamic topologies is chosen because changing environments need to be considered when developing a flexible framework in a fog landscape. Developing a framework is always based on a programming model and therefore this criterion is considered important. Also, because the aim of fog computing is to maximize the utilization of fog resources and reduce cost and latency, resource provisioning is an essential aspect that needs to be tackled and analyzed.

To present the results of the contrasted related work, Table 3.1 visualizes the criteria fulfillment of the analyzed papers. A field in the table is marked with a checkmark (✓) if and only if the paper in that line fulfills the criterion of the corresponding column. Consequently, the paper is marked with (x) if the paper does not satisfy the criterion. Out of reasons of space, the criteria acronyms, defined in the criteria specifications, are used as column labels.

Table 3.1: Comparison of the Related Work. Legend: ✓: fulfilled, x: not fulfilled

	IoT	FC	DT	PM	RP
Aazam et al. [1, 2, 3]	✓	✓	x	x	✓
Bonomi et al. [15, 16]	✓	✓	x	x	x
Dastjerdi et al. [25]	✓	✓	x	x	x
Hong et al. [35]	✓	x	x	✓	✓
Luan et al. [43]	x	✓	x	x	x
Sarkar et al. [53]	✓	✓	x	x	x
Saurez et al. [54]	✓	✓	x	✓	✓
Skarlat et al. [56]	✓	✓	x	x	x
Vaquero et al. [61]	✓	✓	x	x	x
Yi et al. [69, 70]	x	✓	x	x	x
Zhu et al. [73]	x	✓	x	x	x
Singh et al. [55], Zhan et al. [71]	x	x	x	x	✓
Dinh et al. [26], Lu et al. [42]	x	x	x	x	✓
Fog Computing Framework	✓	✓	✓	✓	✓

To the best of our knowledge, almost none of the related work in research cover all the challenges of the fog computing framework envisioned in this thesis. To be more specific, most of the related work covered in this section does not take into account dynamic network topologies that can change over time and resource provisioning approaches to maximize the fog resource utilization and minimize the network latency. In addition to the missing topology dynamics and resource provisioning aspects, only in a few papers the focus was placed on a sophisticated and extensible programming model to execute, test, and evaluate IoT services in a fog landscape. Beside the fact that almost none of the contributions provided an extensible programming model, only very few facilitate a real-world test-bed to evaluate and test their model. Consequently, the development of an application solving the stated research gaps would be a novel contribution. Hence, the envisioned fog computing framework is a not yet researched solution and would be a contribution to technological research in this area.

CHAPTER 4

Requirements Analysis and Design

This chapter presents the general idea of the fog computing framework followed by the concrete requirements and design decisions. The fog computing framework implements an architecture that enables the execution of arbitrary IoT applications in the fog landscape.

The fog computing framework enables developers, researchers, and general users to create, deploy, test, evaluate, and execute IoT applications, and apply resource provisioning approaches in a real-world fog computing test-bed. Because the framework already provides the general functionality of a fog computing landscape, a developer only needs the expertise and knowledge about the specific application scenario to be implemented. In case the user is only executing IoT services in the fog landscape, no specific expertise is required.

The envisioned fog computing framework developed in the course of this thesis provides the basic functionalities including the device topology creation, device communication APIs, a rudimentary resource provisioning and service placement approach, amongst other fundamental tasks necessary for the fog landscape to work as intended. Additionally, these stated functionalities can be adapted and extended to fit the need of the specific application scenario. Beside these functionalities, the fog computing framework has to conform to the following non-functional requirements: (i) scalability, (ii) extensibility, (iii) maintainability, and (iv) portability. Additional non-functional requirements including security, data integrity, usability, and reliability are not yet tackled but shall be considered in future work.

This chapter comprises the functional and technical specification of the fog computing framework. The functional specification in Section 4.1 describes the most important functionalities of the framework. The technical specification in Section 4.2 covers essential technical design and technology decisions, and describes the detailed architecture design, and APIs of the framework. The functional and non-functional requirements, tools, approaches, and general design decisions presented in this chapter are the result of the analysis of existing software frameworks in related areas. The results of two analyzed related software frameworks are described in the background chapter, in Section 2.4.

4.1 Functional Specification

The functional specification aims to define the functional and non-functional requirements, use cases, actors, and workflows to be satisfied by the fog computing framework. This part of the requirements analysis is crucial because sophisticated requirements ease the development significantly and prevent very cost-intensive requirement errors at the evaluation phase.

4.1.1 Functional Requirements

The enlisted functional requirements propose the overall functionality of the fog computing framework. Functional requirements, in contrast to non-functional requirements, state specific functions and behaviors a system has to execute. The following functional requirements are divided into cloud management and fog colony management functionalities. In every part, the functions of the according area are listed and briefly described.

1. *Cloud Management*
   Device, task request, and resource handling in the cloud environment.

   1.1. *Parent Identification*
   Determine the closest possible parent to a requesting device and send back the appropriate connection data.

   1.2. *Cloud Resource Provisioning*
   Deploy and release VMs and containers according to the resource demand of the connected fog colonies in the cloud.

   1.3. *Service Placement*
   Handle incoming task requests by deploying containers on started VMs.

   1.4. *Task Request Execution*
   Execute the task requests in containers running on the started VMs and store the resulting data for further analysis.

   1.5. *Service Data Storage*
   Store propagated service data in a beforehand specified database.

2. *Fog Colony Management*
   Device, task request, and resource handling in fog colonies.

   2.1. *Device Identification and System Topology Creation*
   Create a digital system topology of the connected devices by periodically pinging all children.

2.2. *Resource Provisioning*
Orchestrate, allocate, deallocate, and monitor resources in the associated fog colonies.

   2.2.1. *Reasoning*
   Compute a resource provisioning plan according to a specified resource provisioning approach.

   2.2.2. *Service Placement*
   Handle task requests by deploying containers across fog devices. The concrete service placement of this requirement depends on the developed resource provisioning approach.

   2.2.3. *Service Deployment, Undeployment*
   Deploy and stop services according to the resource provisioning plan.

   2.2.4. *Monitoring*
   Monitor the subjacent devices, e.g., IoT devices, fog cells, fog control nodes, and save monitoring data, e.g., CPU and RAM utilization, into the local database.

   2.2.5. *Service Migration*
   Migrate services from the cloud to fog, fog to fog, or fog to cloud according to changes in the fog landscape, e.g., if a new device appears.

   2.2.6. *Resource Replanning*
   Calculate a new resource provisioning plan according to the events (i) device accedence, (ii) device failure, and (iii) device overload.

2.3. *Shared Storage*
A distributed data storage to share data, e.g., resource utilization, and service images across multiple devices and topology levels.

2.4. *Watchdog*
Analyze the monitoring data and fire events when previously specified QoS thresholds, e.g., 80% CPU utilization, are exceeded.

2.5. *Task Request Handling*
Listen for task requests in the network and send them to the responsible root fog control node for reasoning purposes.

2.6. *Task Request Execution*
Execute the incoming task requests and return the result either to a fog control node or directly to the cloud.

2.7. *Task Request Propagation*
Propagate task requests from fog control nodes to other fog control nodes or the cloud-fog middleware.

2.8. *Service Data Propagation*
Propagate service data from the deployed services to parent fog control nodes or the cloud-fog middleware for further processing or long-time storage.

2.9. *Communication*
Most of the components of the framework provide consuming and exposing APIs to intercommunicate.

2.10. *Service Registration*
Register a service in the service registry for further usage, e.g., deployment on a fog cell or fog control node. The registered service is also distributed to all children fog control nodes.

2.11. *Service Deployment*
Pull a service from the service registry and deploy it on a compute unit of a fog cell.

4.1.2 Non-Functional Requirements

Non-functional requirements describe *how* the system executes specific functionalities and how the system should behave in specific situations. The following paragraphs define the most relevant non-functional requirements of the fog computing framework [30].

Scalability

Scalability is the capability of a software system to adapt its resources according to the volatile demand of the users.

Concrete non-functional requirements for the fog computing framework are: (i) scale the system (resource provisioning and service placement) that the CPU utilization stays beneath 80%, (ii) maximize the fog resource utilization to decrease cloud cost, (iii) enable a fast service deployment.

Extensibility

A software system is extensible if its components can be extended without too much effort. In other words, the system is built to be extended by loose coupling and clearly defined APIs between diverse components.

The concrete requirements concerning this topic are that the software needs to be built extensible. This aspect is achieved by defining a modular software component structure with clearly specified APIs. Additionally, using standardized communication technologies helps to make it easier to extend a software product.

Maintainability

Software maintenance is one of the most underestimated jobs in the software engineering process, although it is a very time consuming, expensive, and tedious task [22]. Thus, it is crucial to keep the software maintainable by documenting the code, writing readable code, refactoring, and documenting the general functionality and interaction between different parts of the system.

Portability

Portability is the capability of a software product to be used across diverse system environments, e.g., filesystems, operating systems, without the necessity to adapt the software manually. In other words, the systems' fundament should be built with platform-independent technologies that preferably are quickly deployable and migratable. Furthermore, platform independent communication technologies help to separate data from concrete technologies.

4.1.3 Actors

Actors are different types of users interacting with a system. As the resulting system is an open source software framework, the actors using the software are general *users*, *developers*, and *researchers*. The fog computing framework provides the fog computing environment to develop, test, deploy, and use IoT services in a fog landscape.

General users are actors using IoT services that already have been developed by developers or researchers. Developers are actors that develop and test new IoT services using the framework, and researchers develop novel approaches and extend the framework to enable new use cases.

4.1.4 Use Case Scenarios

A use case aggregates multiple use case scenarios executed by an actor on a specific system environment. Usually, use case scenarios describe actors executing a specific action in order to receive predefined responses or benefits. For instance, a customer requests several tasks to be executed in the fog landscape and gets back the result or a message about the deployment status.

Some essential use case scenarios of the fog computing framework are presented in the following paragraphs.

Use Case I: Development, Testing, and Evaluation of an IoT Service

- **Actors**: Developer, Researcher
- **Preconditions**: A developer downloads the fog computing framework from the public git repository and creates a desired fog landscape, e.g., a distributed device topology to test the needed functionality.
- **Postconditions**: A developer can deploy, test, and evaluate the created IoT service in a created fog landscape.
- **Scenario**: A developer implements a new IoT service to be executed in the fog computing framework and stores the created Docker Image in the service repository. After the service is registered with a unique service key, the developer can send a task request to the fog landscape. The responsible fog control node receives the task request and issues the deployment of the requested service according to the

computed resource provisioning plan. After the deployment the developer gets notified about the deployment status.

Use Case II: Execution of an IoT Application

- **Actors**: User, Developer, Researcher
- **Preconditions**: A user downloads the fog computing framework from the public git repository and creates a desired fog landscape, e.g., a distributed device topology to test the needed functionality.
- **Postconditions**: A user is able to request the execution of an IoT application in the provided fog landscape and to receive the expected results.
- **Scenario**: A user connects several sensors, e.g., humidity, temperature, sound, to the fog landscape and executes a specified amount of suitable sensor reading services. After the sensors are connected and the responsible fog cell is paired to a parent fog control node, the user can issue the corresponding task requests at the parent fog control node by issuing the desired service types via an API call. The fog control node creates a resource provisioning plan and deploys the services according to the fog cells' resource utilization. The resulting data of the sensors are processed in the requested service and sent to the cloud for further analysis and long-time data storage.

Use Case III: Development, Testing, and Evaluation of a Resource Provisioning Approach

- **Actors**: Researcher
- **Preconditions**: A researcher downloads the fog computing framework from the public git repository and creates a desired fog landscape, e.g., a distributed device topology to test the needed functionality.
- **Postconditions**: A researcher can deploy, test, and evaluate the implemented resource provisioning approach in a newly created fog landscape.
- **Scenario**: A researcher extends the resource provisioning module of the fog computing framework by implementing the *IResourceProvisioning* interface with a more sophisticated resource provisioning approach. This extended approach can then be tested and evaluated within the provided fog landscape. As the framework is kept extensible, even already existing resource provisioning approaches can be attached and subsequently tested and evaluated.

4.1.5 Workflows

In this part of the functional specification, essential workflows are described and presented on the basis of *Unified Modelling Language (UML)* sequence diagrams. These UML sequence diagrams show a chosen sequence of actions between the different system components described in Subsection 4.2.1. In a sequence diagram, the time is visualized in vertical direction while the communication between the specified components is visualized horizontally.

Pairing and Service Deployment

In this first workflow depicted in Figure 4.1, the device pairing and subsequent service deployment is visualized. At some point in time, a fog control node receives an asynchronous pair request from a new device that actively wants to join the fog colony. This device is instantiated as a fog cell and added to the set of children of the fog control node. In the meantime, a task request arrives and while the reasoner component computes an appropriate resource provisioning plan for the newly created fog cell. In this use case scenario we assume that there are pending task requests fitting the exact service type of the newly instantiated fog cell. The resulting plan specifies the service to be deployed on the fog cell according to the current QoS metrics and resource provisioning approach. After the fog control node deployed the service, the service is able to read, process, and propagate the data from connected IoT devices. From this point, the fog cell works autonomously without further controlling of the fog control node.

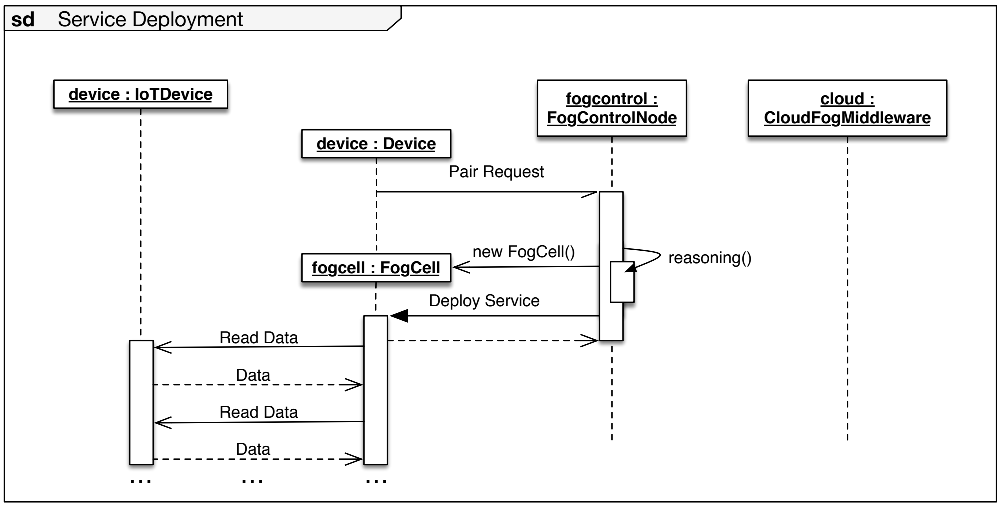


Figure 4.1: Pairing and Service Deployment

Task Processing in a Fog Cell

Figure 4.2 represents a workflow where a new task request is issued at a fog control node which then needs to reason about an appropriate fog cell to deploy the requested service to. The task request is sent from a user via the REST API to the fog control node, where the reasoning component calculates the resource provisioning plan to determine the first fitting fog cell to deploy the needed service. After a fog cell is found, the fog control node deploys the service on the fog cell which immediately starts executing intended functionalities. For evaluation purposes, the fog cell sends back a message after the service is successfully deployed.

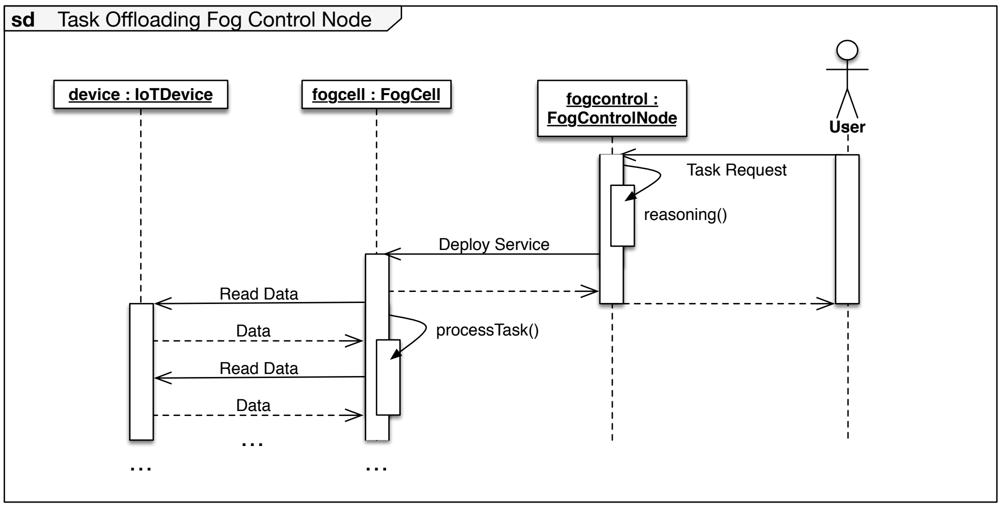


Figure 4.2: Task Processing in a Fog Cell

Task Offloading to the Cloud

The last workflow explains the sequence of actions needed to offload a task request to the cloud-fog middleware (see Figure 4.3). As before, a user sends a task request to a fog control node, which sends it to its parent fog control node for reasoning purposes. As this specific task request is flagged as cloud task because it needs high processing power, e.g., Big Data analytics, the fog control node decides to propagate the request to the cloud. The cloud-fog middleware then deploys appropriate resources to handle the task request appropriately and instantiates VMs in the cloud environment to process the request. In every case, even if no task results exist, the component where the service is deployed, sends back a status message whether the service is deployed successfully. The detailed service deployment process in the cloud is described in Section 5.2.2.

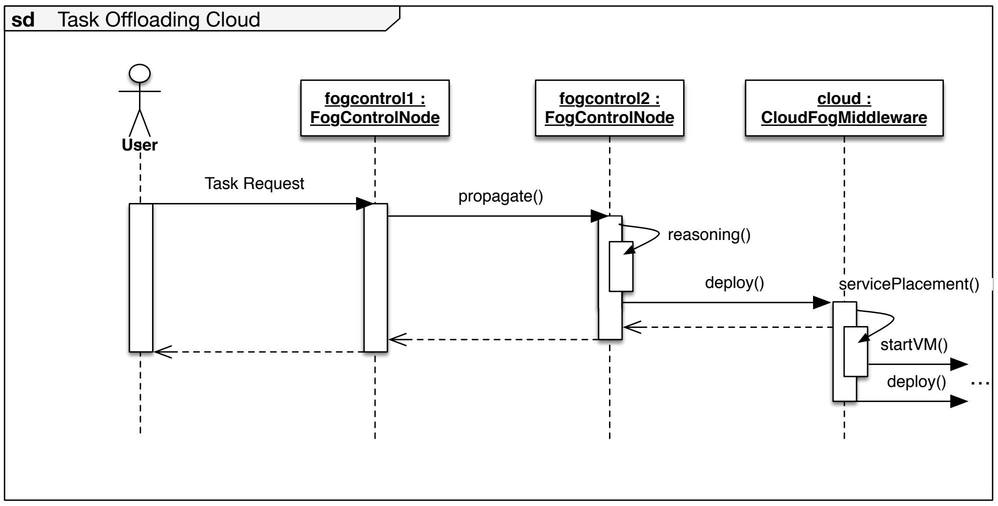


Figure 4.3: Task Offloading to the Cloud

4.2 Technical Specification

In the second part of this chapter, the essential technical specification is introduced. The technical specification defines how the functional and non-functional requirements of the system can be solved by the means of concrete design and technology decisions. The elements used to define the technical requirements include a general overview of the framework architecture, the most important design and technology decisions, and concrete APIs between the described components.

4.2.1 Fog Computing Framework Architecture

The overall architecture is a crucial part of the design and development of the fog computing framework. This section presents the fog computing framework architecture and all the included components. The hereby introduced concept and architecture is an adapted version of the conceptual architecture considered by Skarlat et al. [56].

In Figure 4.4, an overview of the fog computing framework architecture can be seen. The visualized device hierarchy, i.e., device topology, consists of two major layers and four distinct device types. The bottom layer, i.e., the IoT layer, consists of *fog control nodes*, *fog cells* and *IoT devices*. The second layer, on the top of the topology, is the cloud layer, and represents the cloud resources managed by the *cloud-fog middleware*. All mentioned device types can occur multiple times in the the according layers. Additionally, a structure consisting of a fog control node with an arbitrary amount of fog cells and IoT devices connected to it, is called a *fog colony*.

To avoid ambiguity, the terms *task request*, *service image*, *service*, and *application* are specified as follows. A *task request* is a computational job to be computed by fog resources.

Service images are not yet deployed service binaries stored in a shared storage component. *Services* are deployed and running computational software instances to process task requests in capable fog cells, fog control nodes or cloud VMs. Examples for such services include MapReduce applications, stream processing, caching, and distributed storage [56]. An *application* is a set of services that need to be deployed in order to successfully execute an application. The mentioned device types are described in more detail in the remaining subsections.

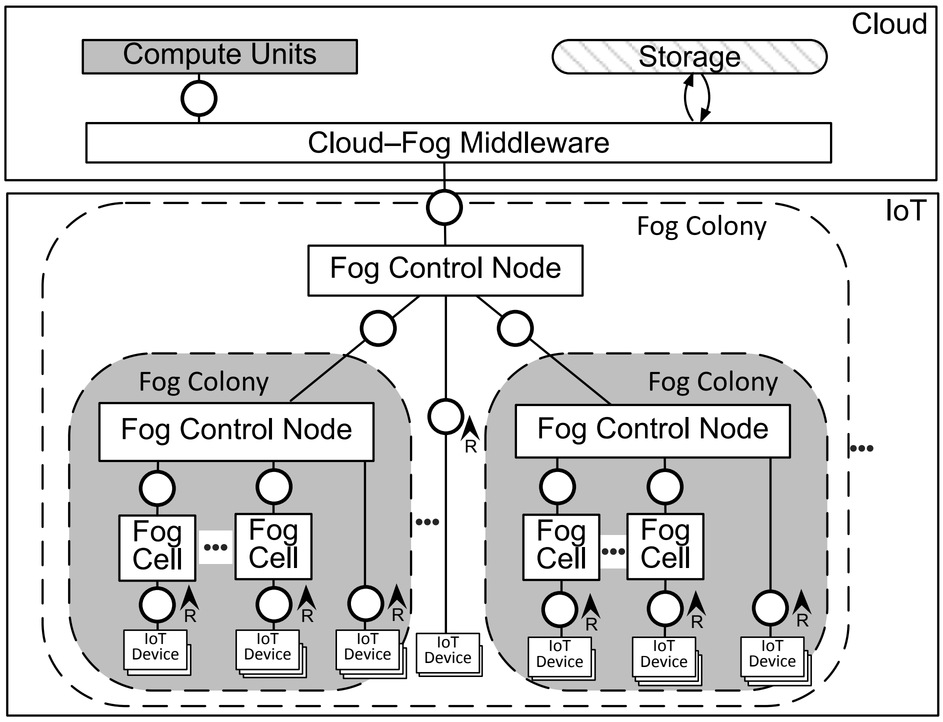


Figure 4.4: Fog Computing Framework Overview (adapted from [56])

Cloud-Fog Middleware

The cloud-fog middleware is deployed in the cloud and acts as a middleware between the cloud and the fog landscape. The middleware handles the resource provisioning, task request placement, and task execution in the specific cloud environment. Task requests processed in the cloud environment most likely are delay-insensitive jobs requested from subjacent fog colonies. These tasks often are resource-intensive, e.g., Big Data analytics, or image processing, and therefore need to be executed in the cloud environment with virtually infinite resources.

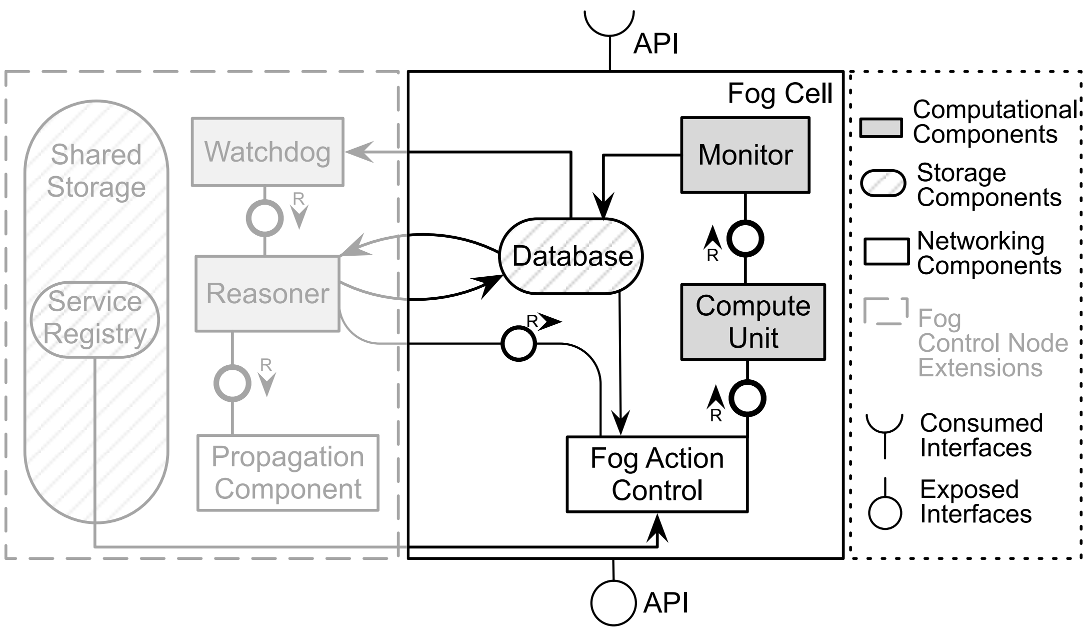


Figure 4.5: Fog Cell Architecture (adapted from [56])

Fog Cell

A fog cell is a software component running on a resource-rich IoT device, having its own computation and storage resources. Fog cells represent computing devices that communicate with a parent control component, i.e., fog control node, and connected IoT devices, e.g., sensors and actuators. The duties carried out by such fog cells include monitoring of connected IoT devices, monitoring of proprietary resources, executing of received task requests, and propagating service data to upper topology levels.

As can be seen in Figure 4.5, every fog cell features specific components. The *fog action control* component pulls a service image from a shared storage, specifically from a *shared service registry*, and deploys this service in the *compute unit* to read, process, and propagate the data from connected IoT devices. Furthermore, the communication with IoT devices and the complete data processing is done in the compute unit. Hence, the compute unit is the actual task processing part of the fog cell. The deployment of services is done according to a resource provisioning plan computed by a reasoning component of a fog control node. The execution of the deployed services and the corresponding resources are observed by the *monitor*. The gathered monitoring information is stored in the local *database* for further processing. Being connected to IoT devices and fog control nodes a fog cell needs an *API* for incoming as well as outgoing communication. The API enables other connected devices to get and set fog cell data, e.g., utilization, and empowers fog control nodes to deploy, release, start, stop, and delete services in the fog cells' compute unit. The exact API endpoint specifications are described in Section 4.2.3

Summarizing, fog cells are resource-rich IoT devices connected to multiple other IoT devices and one parent fog control node. Fog cells receive task requests which are processed locally. The result of the task request can then be propagated to the parent fog control node for further processing.

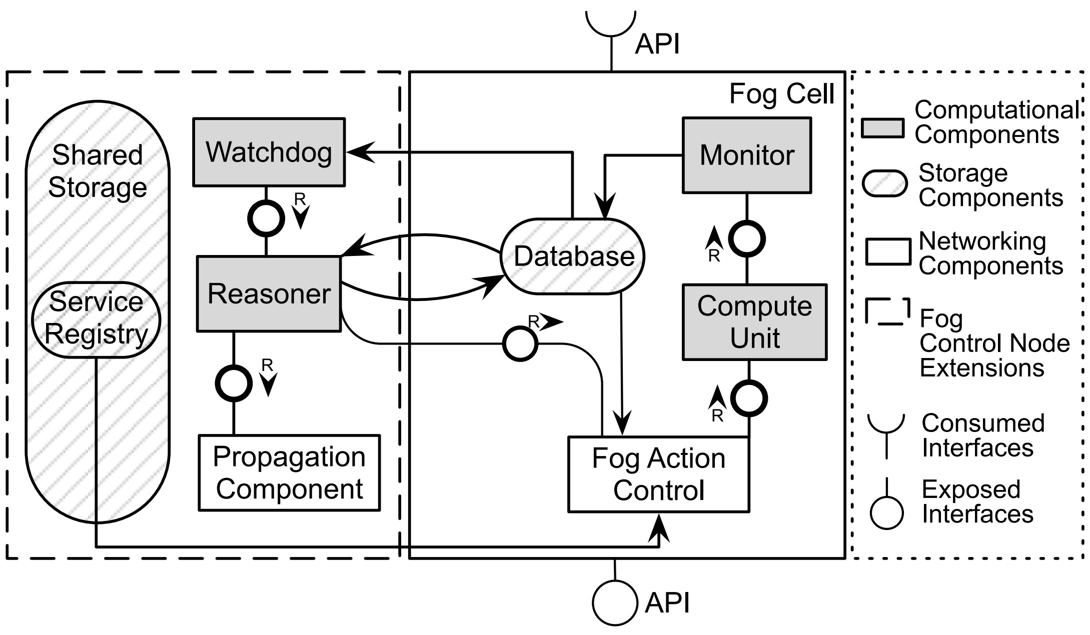


Figure 4.6: Fog Control Node Architecture (adapted from [56])

Fog Control Node

A fog control node is a powerful fog cell with additional capabilities as visualized on the left hand side of Figure 4.6.

Fog control nodes extend simple fog cells with the following components. The *watchdog* analyzes the persisted monitoring data against expected QoS metrics and triggers events accordingly, e.g., overload (CPU > 80%). The triggered events are consumed and appropriately processed by the *reasoner* component of the fog control node. The reasoner is a crucial component that handles the resource provisioning and task placement of the entire fog colony. Note, fog colonies can recursively inherit other fog colonies. In tangible terms, the reasoner computes a resource provisioning plan for associated fog colonies. A resource provisioning plan defines where selected services are deployed, i.e., on which fog cells, and in which service instance the incoming task requests are executed. Creating the plan, the reasoner takes into account watchdog events, monitoring data of the system, and additional aspects according to the implemented resource provisioning approach. The developed resource provisioning approach of this thesis is provided in Subsection 4.2.2. Additionally, the reasoning component reacts on specific system events, e.g., device accedence, device failure, overload, and immediately starts a resource replanning process. This replanning process analyzes the whole infrastructure and migrates suitable services. The *shared storage* component holds the *service registry* and additional information about every fog cell that needs to be shared across multiple devices and topology levels. The service registry holds service images to be deployed in fog cells. These service images have a unique identifier used to identify the registered services. The *propagation* component serves the purpose of propagating task requests and service data to higher levels, i.e., either to the cloud or to other fog control nodes in case a fog cell is not able to process a task request itself.

In a nutshell, fog control nodes can be connected to multiple fog cells, IoT devices, and a cloud-fog middleware. Fog control nodes receive task requests from users outside the fog landscape and determine where these task requests are processed. Furthermore, these components restructure and monitor the subjacent devices.

4.2.2 Design and Technology Decisions

The subsequent paragraphs explain the major design and technology decisions needed to fulfil the functional and non-functional requirements of the fog computing framework described in this work. The decisions range from deployment concepts over base technologies to detailed technical issues.

Deployment Concept

As already discussed in the background chapter, specifically in Section 2.2.4, the two relevant deployment concepts are *full virtualization* and *operating system virtualization* (i.e., *container virtualization*).

The advantages of a full virtualization, in comparison to the container approach, are (i) more secure system separation, (ii) more compatible operating systems and software, and (iii) fault-safe system separation. Regarding full virtualization, the system separation is more sophisticated because of the VMM which handles the separation in the virtual environment. Specifically, VMs can not access any data, memory, or processes of other VMs. Additionally, if one VM fails the others are not affected at all. This is crucial when thinking about security-critical and fault-critical applications. The disadvantages of VMs include huge storage and processing overloads due to the additional operating systems, slow startup times, and a verbose setup. Concerning the container concept, the resulting advantages are: (i) light-weight containers, (ii) fast startup time, and (iii) easy deployment and migration. As containers do not need proprietary operating systems, the needed storage space is substantially lower and the startup times are considerably faster [12, 54].

The components of a fog landscape are easy separable into small independent components and therefore suggest using a MSA. Consequently, in this specific application scenario, the priority is to use the container concept, as the whole framework should be

Table 4.1: Dropwizard vs. Spring Boot [32, 59, 65]

	Dropwizard	**Spring Boot**
HTTP Server	Jetty	Tomcat, Jetty, Underflow
REST Support	Jersey	Spring, JAX-RS, Jersey
JSON Support	Jackson	Jackson, GSON, JSON-simple
Dependency Injection	-	Spring Framework
Metrics	Dropwizard Metrics	Spring Framework
Persistence Tools	JPA	Spring Data, JPA
Dependency Mgmt	Maven	Maven, Gradle, Ant
Community Support	∼2,000 StackOverflow posts	∼20,000 StackOverflow posts
Official Integrations	around 10	more than 50 Maven Starters

kept light-weight, easily extensible, and on-demand scalable eliminating long starting and migration times. The concrete container solution selected for this project is *Docker*. The details regarding Docker have already been described in Section 2.2.4. It remains to be said that even Docker is not an optimal solution for a real-world IoT system, because even more light-weight solutions are needed. Nevertheless, Docker perfectly serves the purpose of creating a real-world fog computing framework test-bed in this thesis.

Base Technologies

Due to the small and independent components, i.e., microservices, a light-weight base technology to enable convenient application development is needed. As the microservices in this specific project are implemented by means of *Java 8*[1], the further technology assessment concentrates on *Java Micro Frameworks*. Java Micro Frameworks are light-weight Java frameworks that enable a developer to easily develop small independent Java microservices. In this specific case, the different components are coupled in microservices that intercommunicate via the microservice standard communication technology REST. More specific information on the communication technology is provided in the next subsection. As a result, we can further restrict the assessment criteria to light-weight Java Micro Frameworks enabling a RESTful application development.

There are numerous open source frameworks available that comply with most of the requirements of this specific application scenario, e.g., Spring Boot[2], Dropwizard[3], Ninja Web Framework[4]. In order to decide which framework fits best in this work, the two most popular and sophisticated RESTful Java micro frameworks are analyzed and compared. The selected two frameworks to be compared are *Spring Boot (SB)* and *Dropwizard (DW)*.

[1]https://docs.oracle.com/javase/8/docs/
[2]https://projects.spring.io/spring-boot/
[3]http://www.dropwizard.io/1.0.2/docs/
[4]http://www.ninjaframework.org/

Table 4.1 visualizes a few particularly crucial selected capabilities a Java Micro Framework should have. Not all proposed capabilities are strictly needed to develop microservices, but are convenient and widely used in this area, e.g., *Dependency Injection (DI)*. The follow-up paragraphs briefly discuss the capabilities and present relevant details about the chosen frameworks.

Hypertext Transfer Protocol (HTTP) servers imply the ability to host a web server and allow HTTP access via POST, PUT, GET, UPDATE, and DELETE commands. In DW only Jetty is available, while SB offers Tomcat, Jetty and Underflow, depending on the preference and requirements of the developer. REST, building the communication part in a MSA, is enabled using Jersey 2 in both DW and SB. SB additionally supports Spring and the *Java API for RESTful Services (JAX-RS)*. JSON handling is supported via Jackson in both tools, and in SB GSON and JSON-simple are available additionally. The DI is a convenient capability used to dynamically initialize project dependencies. In DW Google Guice is an often used light-weight DI solution that can be integrated but is not embedded in the standard framework. SB uses the sophisticated Spring framework as DI module that provides the developer with many convenient DI capabilities.

Regarding metrics, every tool uses its proprietary ones. Metrics serve the purpose of measuring the systems status and event-based or optimization-based alerting. In every framework, persistence tools are an essential topic. In DW only the well-known *Java Persistence API (JPA)* is supported as official persistence integration, while SB provides a very convenient persistence handling with Spring Data and JPA. Spring Data provides very easy methods to save a lot of time and effort when developing persistence functionalities [59, 65].

Dependency management is another crucial topic in distributed applications. In SB, the dependency management is solved by well structured components, while DW works with many community projects that can lead to compatibility problems. Furthermore, SB can be used with *Maven*, *Gradle*, and *Ant*, while DW sticks to Maven. The community support is sophisticated in both of the frameworks, although concerning StackOverflow[5] posts DW is mentioned in around 2,000 posts, while SB is mentioned tenfold. Finally, another important point is the availability of official integrations which help to solve well-known problems by integrating official tools. Here, again, SB has a huge advantage with the Spring environment and the accompanying community, while DW can provide only about 10 official integrations.

Concluding, in almost every aspect compared in Table 4.1, SB is a better fit for this work. The advantages of SB result from the well-known and sophisticated technology base, i.e., the Spring framework. Thus, most of the tools and background technologies are very well-developed and tested because they are also used in the Spring framework. Furthermore, the experience of development in a Spring background enables a very easy development start with Spring Boot.

[5]`http://stackoverflow.com/`

Communication

Since the deployment concept and architecture are already agreed upon, the selection of the communication technology is straight forward. Due to the fact that microservices are light-weight software components with heterogeneous technologies, the standard communication technology in MSA is REST.

REST is an architectural style describing how to develop web services, or microservices in this case, that communicate by the means of simple HTTP requests. The communication between microservices has to comply four principles: (i) statelessness, (ii) uniform interfaces, (iii) resource identification using *Uniform Resource Identifiers (URIs)*, and (iv) self-descriptive messages. Statelessness means that every communication between the components is independent and no specific state is stored in the components. Uniform interfaces indicate that every action done on the components is executed via clear well-defined *GET, POST, PUT,* and *DELETE* commands. These commands are executed by calling unique URIs mapped to execution methods on the component, i.e., the API. Finally, all messages exchanged with a RESTful server are self-descriptive, meaning that the structure and content of these messages are separable. Hence, messages can be delivered using different data types, e.g., XML, JSON, without losing any information [46, 48]. An extensive comparison between REST and other WS* technologies can be reviewed in a paper of Pautasso et al. [46].

Cloud Provider

In order to keep the framework as an open source project, an open source cloud technology is selected. The requirements to elicit an adequate open source cloud infrastructure provider are the following. The cloud provider needs to provide an open source API, a convenient deployment model, efficient scalability, low cost, private and public cloud support, large community support, and a reliable and well-tested system.

The currently most important open source cloud infrastructure providers with the required capabilities are OpenStack[6], Apache CloudStack[7], and OpenNebula[8]. Concerning this project, OpenStack is chosen for the following reasons: (i) modular architecture, (ii) huge community support, (iii) well-known contributing companies as, e.g., NASA, AT&T, CERN, Yahoo, (iv) mature system, and (v) interoperability of cloud services. The stated reasons emphasize that OpenStack is a very sophisticated and mature cloud infrastructure provider that even supports the interoperability of multiple different cloud services. Besides, the framework is built on top of clearly specified interfaces in order to keep it extensible and adaptable. Thus, the chosen OpenStack provider can be replaced or extended by others with little effort.

[6]https://www.openstack.org/
[7]https://cloudstack.apache.org/
[8]http://opennebula.org/

Shared and Local Storage

The technology used for the shared and the local storage in fog cells and fog control nodes of the fog computing framework should be light-weight, fast, and easy-deployable. As no specific structure is needed, the architecture suggests using a fast key-value database. Thus, a well-known in-memory data storage solution providing the needed non-critical functionalities called *Redis*[9] is chosen. Redis is fast, open source, light-weight, convenient deployable and embeddable using Java, and therefore matches the setup of the project perfectly.

Resource Provisioning

As a fog computing framework is responsible for the appropriate resource provisioning in the fog landscape, a general approach needs to be designed. This work only presents a basic resource provisioning approach as the resource provisioning is not in the focus of this thesis and therefore should just give a broad insight on the possibilities in the fog computing framework. The architecture of the framework is kept extensible and thereby allows developers to implement their own improved resource provisioning approaches for further usage. The hereby provided approach is a heuristic approach based on two major actions: *(i) threshold monitoring*, and *(ii) load balancing.*

Due to the dynamic task request workload requested by IoT devices, the framework needs to be able to dynamically orchestrate, allocate, release, and monitor the according resources. System monitoring enables to keep track of the current system utilization, and can trigger events according to predefined threshold rules. For instance, a fog cell is well-utilized having 70% CPU utilization and then an additional service is deployed on this very fog cell. As a result, the utilization exceeds a predefined 80% threshold and the resource provisioning component stops deploying more services on that fog cell. This approach makes sure that fog cells never get overloaded and the workload is balanced among the available resources. In case there are no available fog cells to schedule the task to, the fog control node sends the task request to the cloud-fog middleware which provisions appropriate cloud resources to process the request [35].

Furthermore, when a user issues a task request, the fog control node propagates it to the root fog control node for resource provisioning purposes. The root fog control node then analyzes all the subjacent fog cells and propagates the task request to a fog cell with available CPU and RAM resources. The algorithm used to select fog cells is the *first fit algorithm* [36]. This algorithm selects the first fog device with available resources to deploy the service. More details on the algorithm is presented in Section 5.6.

Due to the task distribution according to the monitored resource utilization, the task load is balanced between subjacent fog cells of the fog control node. In Figure 4.7, a scenario with two fog control nodes and two fog cells is visualized. In this specific scenario, a user sends a task request to a fog control node (fogcontrol$_2$) which is not the root fog control node. Consequently, this fog control node propagates the task requests to the root fog control node for reasoning and load balancing purposes. The requested

[9]`http://redis.io/`

tasks then are deployed on the first fitting fog cell (fogcell$_2$) and a deployment status information is sent back to the user. The user then sends another task request to the same fog control node that again propagates that request to the root fog control node. This time the reasoning decides to deploy the service on another fog cell (fogcell$_1$) because fogcell$_2$ has not enough resources to deploy another service.

The concrete algorithm solving the mentioned approach is described in the implementation chapter, specifically in Section 5.6.

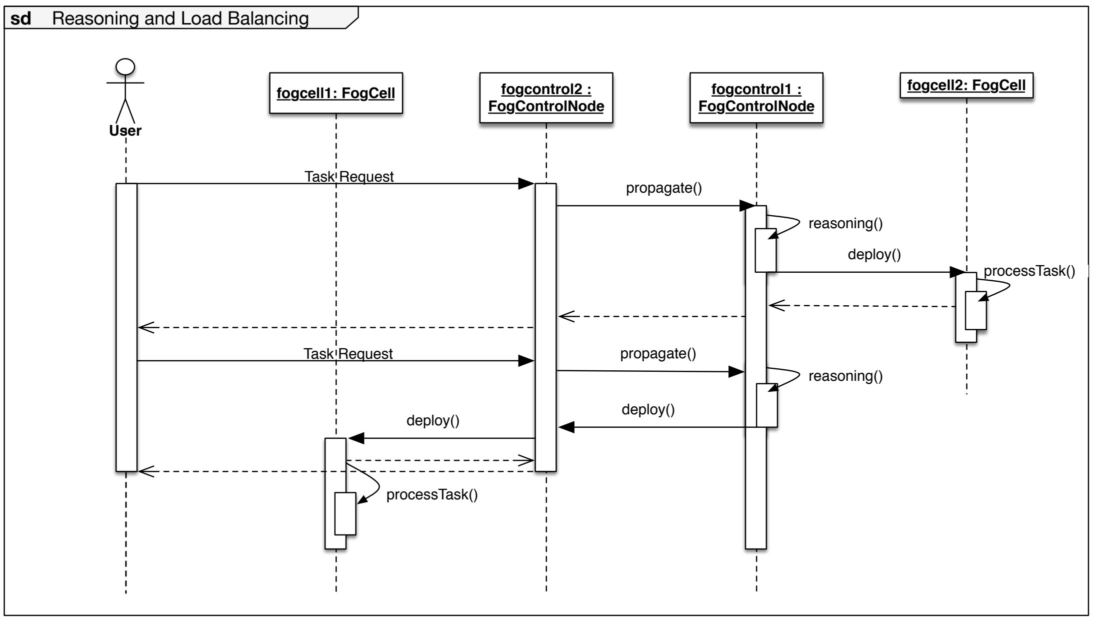


Figure 4.7: Reasoning and Load Balancing

Service Placement Replanning

In case the fog landscape changes, the service placement may have to change in order to ensure the correct execution of the requested applications or to use the fog resources efficiently. As already mentioned in the application definition, the deployment of an application is only successful when all services within that application are deployed successfully. Hence, if a device that runs a service of a specific application fails, this service needs to be deployed somewhere else to ensure the correct application execution. Another scenario, in case not all services can be deployed, the whole application fails and all services are stopped.

In the fog computing framework three distinct cases are distinguished: (i) *device accedence*, (ii) *device failure*, and (iii) *device overload*. A specific monitoring functionality built in the reasoner component analyses the fog landscape and fires events in case a new device joins the network, a paired device is not responding anymore, or a device is overloaded. This functionality provides the mechanism to react on such events and can then be used for any kind of policy to react and handle the placement replanning.

In the course of this thesis a basic policy is implemented: Case (i), when a new device joins the network, all connected devices are analyzed regarding the amount of services deployed. First, all services currently deployed in the cloud are migrated to the new device to reduce cloud cost. Second, all devices running the maximal amount of deployable containers are considered for the remaining service placement replanning. This replanning approach then migrates suitable services from the selected devices to the new device until the new device is filled to a previously defined amount of containers. Case (ii), when a paired device disappears, the reasoner checks in the service mappings whether the device had any services running and deploys them somewhere else in the fog landscape. Case (iii), when a device is identified as overloaded by the watchdog, the event is triggered at the root fog control node and the replanning mechanism migrates a random service from the overloaded device to a suitable substitute. This process is continued until the device is not overloaded anymore.

4.2.3 API Endpoints

In this section the most important API endpoints between the cloud-fog middleware and connected fog control nodes, a task requesting user and fog control nodes, and between deployed services and the fog cell it is deployed on, are presented. Table 4.2 and Table 4.3 depict the endpoints to be used by an external user. Table 4.4, Table 4.5, Table 4.6, Table 4.7, Table 4.8, and Table 4.9 are endpoints between the cloud-fog middleware and fog control nodes. Finally, Table 4.10 shows the interface that enables deployed services to propagate processed data to the fog cell and from there to fog control nodes and the cloud.

This chapter presented the functional, and non-functional requirements, crucial design decisions, the framework architecture, the component design, essential use cases, and the API endpoints of the fog computing framework. Furthermore, the information written on the previous pages answers two research questions: (i) “What are the functional and non-functional requirements for a fog computing framework?” and (ii) “What are the necessary components of a fog computing framework, and how can they be described in terms of their functional and technical specification” stated in Section 1.2.

Table 4.2: Register Service Endpoint

Register Service	
Description	This endpoint enables users to register services in the shared service storage, located on fog control nodes, for future deployment.
Request	
URL	POST http://fcn1:8080/shareddb/register
JSON	{ "serviceKey": "busy-image", "dockerfile": "FROM rpiBusybox\n ...", "volumes": "/usr/lib/..:/usr/lib/...", "exposedPorts": ["8105"], "privileged": true }
Response	
Description	Returns a status flag if the registration was performed successfully.
JSON	201: { "status": true } - successfully registered the service 200: { "status": false } - either the service key is already used or an error occured

Table 4.3: Send Task Requests Endpoint

Send Task Requests	
Description	This endpoint enables users to send a list of task requests to a fog control node.
Request	
URL	POST http://fcn1:8080/reasoner/taskRequests

JSON	{ "duration": 5, [
	{ "serviceType": "t1", "serviceKey": "busy-image", "cloudTask": false, "fogTask": false },
	{ "serviceType": "t2", "serviceKey": "temp-hum", "cloudTask": false, "fogTask": true }
	] }

Response	
Description	Returns a status message that indicates whether the services, needed to execute the task requests, could be deployed.
JSON	201: { "status": true } - successfully deployed the needed services 200: { "status": false } - either an error occurred, or the fog colony is overloaded and no cloud-fog middleware is connected

Table 4.4: Get Responsible Parent Endpoint

Get Responsible Parent			
Description	This endpoint enables new fog devices to request the closest parent device of the fog colony to connect to. If no parent is in range, the cloud-fog middleware returns itself.		
Request			
URL	GET http://cfm1:8082/locator/parent/<latitude>/<longitude>		
Parameters	Name	Description	Example
	latitude : Long	North-South geographical coordinate of the requesting device	48210033
	longitude : Long	West-East geographical coordinate of the requesting device	16363449
Response			
Description	Returns a fog-device object in JSON format that at least contains IP address and port of the parent to connect to.		
JSON	200: { "id": "...", "ip": "192.168.1.105", "port": "8080",... } - successfully found an appropriate parent 200: { "id": "...", "ip": "192.168.1.101", "port": "8082", ... } - the fog-device object corresponding to the cloud-fog middleware is returned		

Table 4.5: Fog Control Node Propagator Endpoint

Propagate Service Data and Task Requests	
Description	This endpoint enables fog control nodes to propagate task requests and general data to parent fog control nodes. In case of service data this endpoint can also be used with the cloud-fog middleware.
Request	
URL	POST http://fcn1:8080/propagator/propagate
JSON	{ "sender": "<device : FogDevice>", "key": "<key : String>", "data": "<list : List<HashMap> >" "requests": "<requests : List<TaskRequests> >, ..." }
Response	
JSON	200: empty

Table 4.6: Cloud Fog Middleware Propagator Endpoint

Propagate Task Requests to Cloud	
Description	This endpoint enables fog control nodes to propagate task requests to the cloud-fog middleware.
Request	
URL	POST http://cfm:8082/propagator/propagateTaskRequests

JSON	{ [
	 { "serviceType": "t1", "serviceKey": "busy-image", "cloudTask": false, "fogTask": false },
	 { "serviceType": "t2", "serviceKey": "cloud-service", "cloudTask": true, "fogTask": false }
	] }

Response	
JSON	200: empty

Table 4.7: Cloud Fog Middleware Stop Service Endpoint

Stop Service in the Cloud			
Description	This endpoint enables fog control nodes to request the stopping of a service in the cloud.		
Request			
URL	POST http://cfm:8082/cloud/stopService/<containerId>		
Parameters	Name	Description	Example
	containerId : String	Unique Docker Container Id	asd9ewaf-awf6ixiy-d33ksa
Response			
JSON	200: empty		

Table 4.8: Get Children Endpoint

Get Children Devices	
Description	The fog control nodes and cloud-fog middleware use this endpoint to get the connected child devices of a connected child device. This is needed to build the topology and provision resources for the tasks. This call works recursively to the bottom of the hierarchy.
Request	
URL	GET http://fcn1:8080/localdb/children
Response	
JSON	200: [] - device without children 200: [{fogdev1: Fogdevice}, {fogdev2: Fogdevice}]

Table 4.9: Get Resource Utilization Endpoint

Get Resource Utilization	
Description	Enables the fog control node and cloud-fog middleware to get the resource utilization of a specified device.
Request	
URL	GET http://fc1:8081/localdb/utilization
Response	
JSON	200: { "cpu": 0, "ram": 0, "storage": 0 } - not yet monitored device 200: { "cpu": 12.34, "ram": 45.23, "storage": 10.87 } - monitored utilization

Table 4.10: Fog Cell Propagator Endpoint

Propagate Service Data	
Description	This endpoint enables deployed services to propagate service data to fog cells which then can propagate it to fog control nodes.
Request	
URL	POST http://fc1:8081/compunit/serviceData
JSON	{ "key": "<key : String>", "data": "<list : List<HashMap> >" }
Response	
JSON	200: empty

CHAPTER 5

Implementation

This chapter serves the purpose of explaining the concrete implementation of the fog computing framework. The next pages cover a general overview of the developed system followed by specific requirements of the necessary software components of the system, installation instructions, exemplary use case scenario executions, and the explanation of the implemented resource provisioning approach. The first section gives a general overview of the system including the technologies, design decisions, and the communication between its components. Section 5.2 describes the concrete service deployment mechanisms used in the cloud and the fog landscape. In Section 5.3, the requirements of the diverse subcomponents are described and explained, enabling a reproducible setup of the system and all its corresponding components. Section 5.4 explains the most important installation instructions in order to get the developed framework up and running. These steps include the installation of the required Raspberry Pis, the cloud environment, and the general system environment. Section 5.5 explains the concrete execution of three described use case scenarios from the design chapter (Section 4.1.4). Including concrete sample commands and files, the execution can be reproduced. In the last part of this chapter, a basic resource provisioning approach including the general idea and the concrete first fit algorithm is explained in more detail.

5.1 Bird View

This first section in the implementation chapter provides a broad overview on the implementation of the fog computing framework. In more detail, the class diagrams of the cloud-fog middleware and of the fog control node are presented. Because the complete fog cell class diagram is included in the fog control node class diagram (Figure 5.1), a fog cell diagram is omitted. The following class diagrams provide an overview on the concrete implemented software components and the interfaces which these components implement. The class diagrams provide future developers the main structure of the implementation at a glance. Furthermore, these diagrams illustrate the extensibility of the software by using interfaces to replace specific components or approaches, e.g., the resource provisioning service. For the sake of clarity and simplicity, the succeeding diagrams only model the most important classes without attributes and methods.

In class diagrams, the naming of entities is self-descriptive. *Services* represent components or separable tasks, e.g., ReasonerService. *Controllers* contain the API endpoints of specific services, to enable the communication with other devices or listen to task requests from outside the systems' environment, e.g., ReasonerController. Software interfaces are recognizable by a preceding letter "I" in the entity name, e.g., IReasonerService. The different components and classes are not described in detail, as the selected class names already point out the related components described in Section 4.2.1.

Figure 5.1 depicts the fog control node application with all its components and interfaces. The separate components are the ReasonerService, PropagatorService, FogActionControlService, ComputeUnitService, DatabaseService, MonitorService, WatchdogService, SharedDatabaseService, and CommunicationService. Most of these services come with an interface to easily extend or replace them with substituting source code implementing the specified interface methods. In the system, general-purpose services are envisioned, e.g., the DatabaseService, which introduce an additional level of inheritance. The added level of inheritance empowers these components to be exchangeable by classes implementing the specified Java interfaces, e.g., to be used by different database technologies.

As already mentioned, the fog cell class diagram is contained within the fog control node class diagram. However, the classes PropagatorService, SharedDatabaseService, and the WatchdogService belong only to the fog control node class diagram.

The cloud-fog middleware, visualized in Figure 5.2, consists of a DatabaseService, CloudService, LocationService, and CommunicationService. The CloudService introduces an additional level of inheritance with the OpenStackService. This class provides the possibility to exchange the implemented OpenStack solution with any other cloud provider solution by implementing the depicted ICloudProviderService interface.

In addition to these classes, the components share a *fogdata* module with generally used models and utility functions. This is kept in a separate model in order to avoid code repetition and empower the developer to change the model at a central point for all components.

5.2 Service Deployment

A crucial task of the designed and implemented fog computing framework is the deployment of the received task requests resulting in running software services. The requested task is mapped to a service that is deployed on an elicited device. All services are deployed in Docker Containers on calculated host devices. Due to two different environments, cloud and fog, we have to differentiate between fog service deployment and cloud service deployment. The stated environments differ in terms of processor architecture, host resource possibilities, and deployment mechanisms.

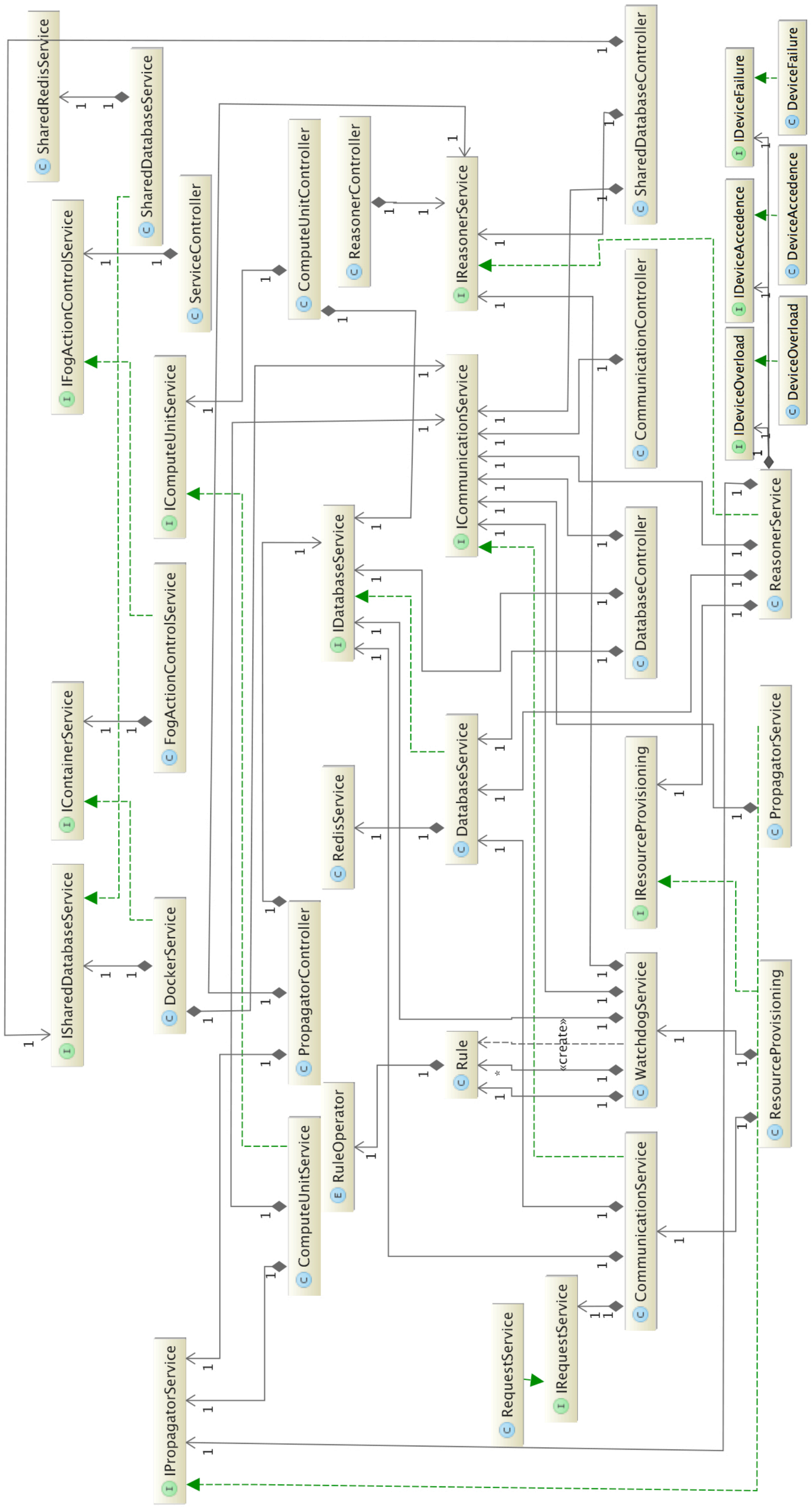


Figure 5.1: Fog Control Node Class Diagram

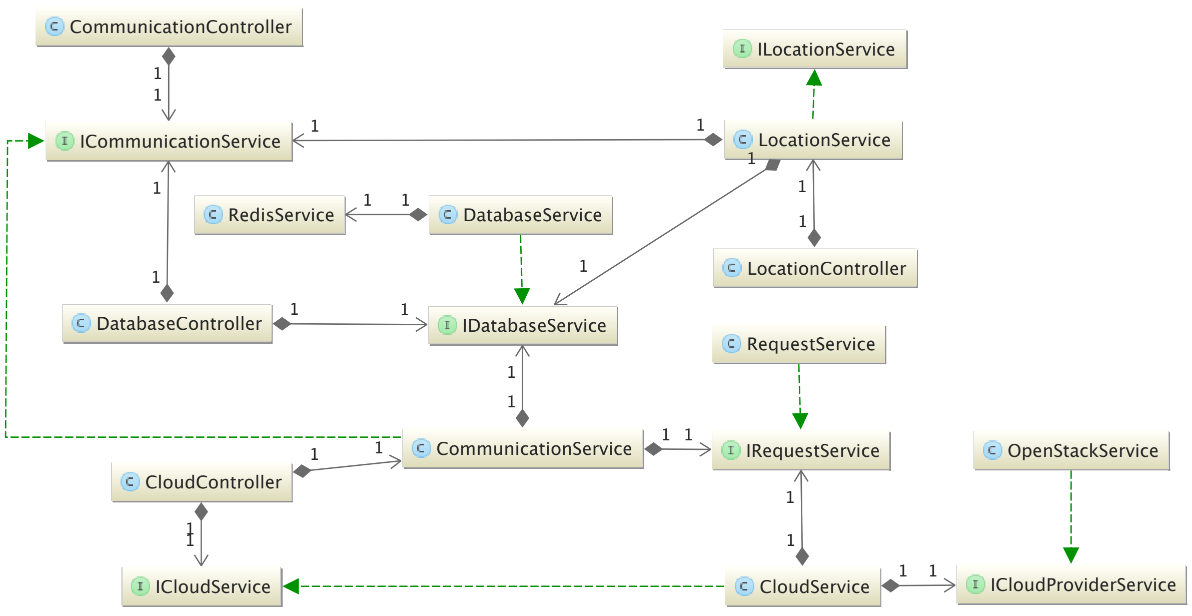


Figure 5.2: Cloud-Fog Middleware Class Diagram

5.2.1 Fog Service Deployment

In the fog landscape, services are deployed on fog cells and fog control nodes. These components are assumed to be running on Raspberry Pis[1] with an ARM processor architecture[2]. Although the chosen programming language Java is platform-independent, the base images of the Docker Containers to deploy still depend on the processor architecture they run on. Hence, to deploy Docker Containers in the fog landscape, special ARM-compatible base images are required.

The service deployment setup in the fog landscape is the following. Every fog device, i.e., fog cell or fog control node, runs on a Raspberry Pi with an operating system called *Hypriot*[3]. Hypriot runs a Docker runtime that enables the service deployment in Docker Containers. The fog cell and fog control node applications run as Docker Containers in the Docker runtime provided by the host operating system. In order to empower these main applications to start and stop further Docker Containers on the host, a specific Docker hook is needed (see Figure 5.3 and 5.4). This hook enables the main applications, i.e., the fog cell or fog control node application, to use the host Docker runtime. As a result, the fog cell and fog control nodes are able to start and stop containers on the same operating system using the same Docker runtime. Therefore, the main applications can start and stop neighbouring Docker Containers from inside their own Docker Container.

[1]https://www.raspberrypi.org/
[2]arm.com/products/processors/instruction-set-architectures/index.php
[3]https://blog.hypriot.com/

Furthermore, we have to differentiate between fog cells and fog control nodes. The difference between these two device types are the deployed database containers. Figure 5.3 visualizes the fog cell having an additional Container called *Redis FC* running a Redis database to persist and read locally needed data, e.g., device utilization as RAM, CPU, and storage. On the other hand, concerning the fog control node in Figure 5.4, there are two database containers *Redis FCN* and *Redis Shared*. Redis FCN is the local database similar to the one in the fog cell, while Redis Shared is the shared database holding the Docker Images of the deployable services of the corresponding fog landscape. The fog control node database containers are intentionally kept separately to preserve flexibility and replaceability. As a consequence, the shared database could be replaced by any other database technology hosted anywhere, due to direct IP communication.

Monitoring of the Raspberry Pi resources is an important task, since the service deployment depends directly on the requested monitoring data stored in the local Redis database. To be able to monitor the resource utilization of the Raspberry Pi instead of the separated containers, a monitoring application directly running on the host system is needed. This light-weight monitoring application running on the Raspberry Pi operating system periodically monitors the systems' resources and saves the resulting utilization data into the local Redis database for further processing.

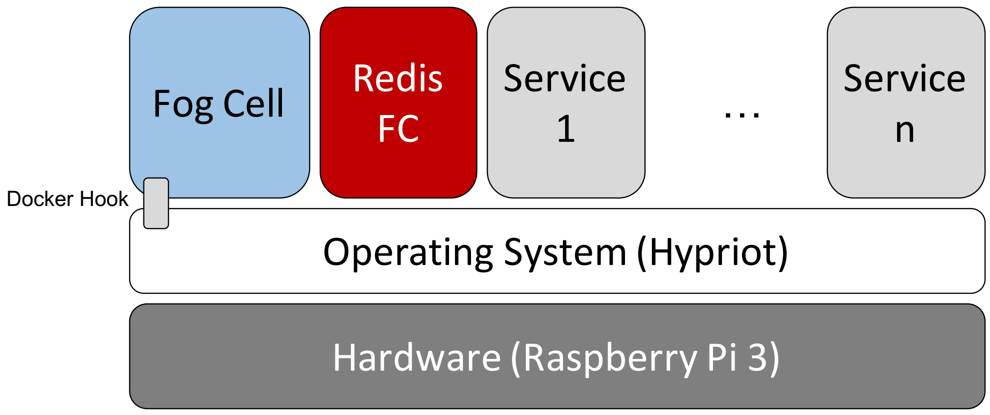


Figure 5.3: Fog Cell Deployment on the Raspberry Pi

5.2.2 Cloud Service Deployment

In the cloud environment, the service deployment is different due to several reasons. First, the services are deployed on dynamic VMs that need to be deployed, managed, and stopped according to dynamic cloud resource demand. Second, the VMs in the cloud are based on an Intel processor architecture [40]. Third, all the Docker Images, needed to deploy services in the cloud environment, have to be pushed to the Docker Hub[4] image repository before requesting deployment.

[4]`https://hub.docker.com/`

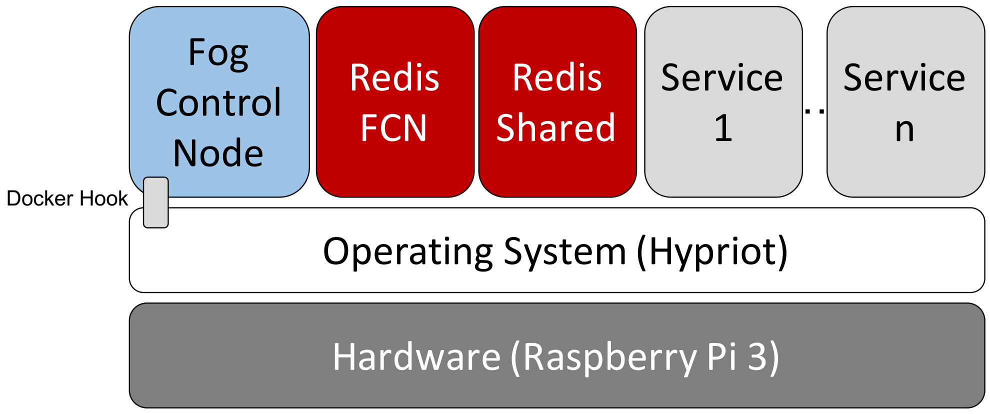


Figure 5.4: Fog Control Node Deployment on the Raspberry Pi

The developed policy to create and release VMs in the cloud is performed as follows. When a task request reaches the cloud-fog middleware and no VM is running yet, the cloud service starts a new VM. The VMs are based on a light-weight CoreOS[5] operating system running a Docker environment for service deployment by default. When a VM is running, it is filled with Docker Containers until a defined maximum number of containers is reached. If there is no free space for another container, a new VM is created. The same policy in the other direction. Hence, if some containers are stopped, resulting in a VM with no deployed containers, the VM is released to save cloud resources.

Figure 5.5 visualizes a possible scenario where the cloud-fog middleware receives two task requests from a fog control node located in the fog landscape. At the time the first service is requested, no VM is running in the cloud environment. Consequently, the cloud-fog middleware needs to start a new VM to deploy the requested service container on. After the VM booted, and the service is deployed successfully, the second task request appears. Due to the fact that a VM is already running and this VM has enough resources to deploy another service, the second service also is deployed on the same VM. Finally, after both services are finished, the cloud service stops the VM and releases its resources.

5.3 Component Requirements

The next sections comprise the requirements for the installation of the diverse software components working together in the fog computing framework. The three layers of operations, i.e., fog, cloud, and IoT, have varying setup requirements that need to be considered when deploying the components required for the fog computing framework.

To enable the developer to change implementation-specific parameters without editing source code, a standard property file, i.e., *application.properties*, is used. In addition to this executable embedded property file to change more advanced settings, e.g., con-

[5]`https://coreos.com/`

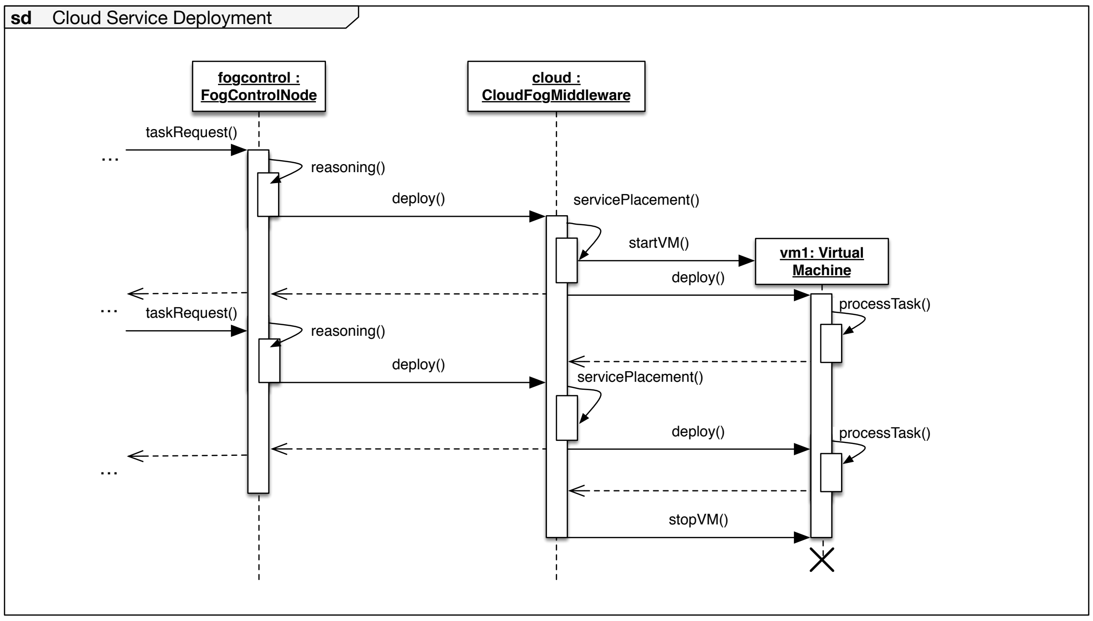


Figure 5.5: Cloud Service Deployment

nection timeout, network port, there exists an additional property file to change more flexible implementation independent data, e.g., device IP. This second property file, i.e., *main.properties*, is not embedded in the executable and therefore enables the developer to change settings without the requirement of rebuilding the whole application. The additional requirements depending on the diverse components are described in the following.

5.3.1 Cloud-Fog Middleware

The cloud-fog middleware can either be deployed on a cloud VM, or locally on any device connected to the root fog control node and the cloud. Due to the fact that the cloud-fog middleware needs to be empowered to start and release VMs in the cloud environment, the application needs to authenticate itself with corresponding cloud credentials. In case of the implementation setup with an OpenStack cloud, the OpenStack credentials are required. The credentials need to be added to the *credential.properties* file located in the cloud directory in the resource folder of the cloud-fog middleware Java project. A sample credentials file specifying the needed keys is provided.

Before running the application, it is crucial to start a Redis database on the same host using the port specified in the property file. The standard port defined is 6382. A script to deploy a Redis container is provided in the root directory of the cloud-fog middleware Java project. Another important requirement includes the look up of the host IP and the subsequent definition in the *application.properties* file for further usage.

5.3.2 Fog Cells

In the case of fog cells, several other properties make sure the application works as intended. The additionally required property file for fog cells is called *main.properties* and needs to be located in the same directory as the Java executable. The exact location and setup description is described in Section 5.4. The property file has to contain the following information: cloud IP, cloud port, fallback parent IP, fallback parent port, device IP, latitude, longitude, service types. A sample property file is provided in the project files.

The cloud IP and port are needed to connect to the cloud-fog middleware and dynamically request the responsible parent to pair to. In the case when the cloud does not respond, or the system does not consist of a cloud endpoint, the fallback parent property is needed. The fallback parent embodies a fallback mechanisms used either if no cloud exists, or the returned parent does not respond. This mechanism enables the fog cell and fog control node to keep operating even though the cloud does not work or the returned responsible parent does not exist. In the case when the fallback parent does not respond as well, the application can not be successfully started and is terminated with an error message explaining the concrete error scenario. The device IP property obviously states the IP address of the host, i.e., Raspberry Pi, the corresponding application is running on. This property needs to be specified with the IP address of the network interface that connects the host with the network connecting all the fog components. To enable a location-aware framework, the latitude and longitude of the device need to be considered. The last property required to run a fog cell is a list of service types the fog cell is able to process.

In addition to the mentioned properties, a fog cell requires a running Redis database on the same host. A Redis container is started automatically when executing the *run.sh* startup script provided and explained in Section 5.4.

5.3.3 Fog Control Nodes

Concerning fog control nodes, most of the requirements are similar to the requirements of fog cells, yet some additional properties are specified in the *main.properties* file: location range upper limit and location range lower limit. These two additional properties define the location range a fog control node covers and is responsible for. Hence, if a device in the fog landscape requests a parent at the cloud-fog middleware, the location service of the cloud-fog middleware returns the parent device that covers the location range the requesting device fits into. More details on the location service and a visualization of the location grid is provided in Section 6.2.

Also, fog control nodes need two running databases. The local Redis database and the shared Redis data storage. As described in the fog cell requirements, prepared Docker Containers get started automatically via a fog control node-specific startup script.

5.4 Installation Instructions

This section includes the setup of the developed fog computing framework, its environment, and all the needed components to execute the framework as intended. The next subsections describe the Raspberry Pi setup, the cloud setup, and the setup of the whole environment around the specific devices.

5.4.1 Raspberry Pi Setup

Aiming at a fast and reliable Raspberry Pi setup, all technologies have been selected with respect to the capabilities of these small one-board computers. In more detail, due to rather low CPU, RAM, and storage resources of the Raspberry Pi, the light-weight Linux-based operating system *Hypriot* was chosen. Hypriot was one of the first Debian-based operating systems that enabled the execution of Docker on a Raspberry Pi.

An important first setup step regarding the operating system is to register the wireless network in the *device-init.yaml* after flashing the operating system to the memory card. Once the operating system is flashed successfully, the Raspberry Pi is running and connected to the WiFi, the next task is to make sure the system is up to date. Meaning, the already installed packages, package lists, and tools need to be updated to the newest stable versions. Furthermore, the open source developer kit Open JDK 8 has to be installed to enable the execution of Java Archive (JAR) files. A JAR is a compressed Java archive file format used to execute Java programs including meta data and dependencies.

In case of fog cells, an additional sensor setup is required. The sensor setup activates the ports and tools needed to communicate with the connected sensor board. Specifically, the needed package is called *i2c-tools*. After successfully installing these tools, the I2C bus needs to be activated by enabling the I2C field in the *raspi-config* menu. To finish the sensor setup, the system requires a reboot to reinitialize the changed settings and enable a correct communication with the connected sensor board.

Listing 5.1 visualizes an example setup script of a fog cell. In case of a fog control node only lines 1 to 3 are required.

Listing 5.1: Fog Cell Setup Script

```
1 #!/usr/bin/env bash
2 sudo apt-get update -y
3 sudo apt-get install openjdk-8-jdk -y
4 # additional fog cell sensor setup commands
5 sudo apt-get install i2c-tools -y
6 sudo raspi-config
7 sudo reboot
```

Beside this basic setup, some more specific settings need to be taken into consideration. Being able to communicate and update the files on the Raspberry Pi in a more convenient way, a SSH public key authentication needs to be set up. As the Docker Image to deploy the device dependant Docker Container is created locally on every Raspberry Pi, the files required to create the image and deploy the containers need to be copied to the device. The necessary files to deploy the application are the following: *hostmonitor.jar*, *Dockerfile*, *main.properties*, *run.sh*. Depending on the device type an additional JAR file called *fogcell.jar* or *fogcontrolnode.jar* has to be added. This last file is the executable for the main application on the corresponding device.

Starting the base containers, i.e., fog cell or fog control node container, and the database containers, is done by running the already mentioned *run.sh* script. An example script of a fog control node is presented in Listing 5.2.

Listing 5.2: Fog Control Node Run Script

```
1  #!/usr/bin/env bash
2  docker run -d -p 6380:6379 \
3      --name redisFCN hypriot/rpi-redis:latest
4  docker start redisFCN
5
6  docker run -d -p 6383:6379 \
7      --name redisShared hypriot/rpi-redis:latest
8  docker start redisShared
9
10 nohup java -jar hostmonitor.jar > hostmonitor.log &
11
12 docker build -t fogcontrolnode .
13 docker rm -f fogcontrolnode
14 docker run -ti -p 8080:8080 \
15     --name fogcontrolnode \
16     -v /usr/bin/docker:/usr/bin/docker \
17     -v /var/run/docker.sock:/var/run/docker.sock \
18     --link redisFCN fogcontrolnode
```

Line 1 marks the script as a bash file for execution purposes. In lines 2 to 4, a Redis Docker Container with port 6380, name *redisFCN*, and base image *hypriot/rpi-redis:latest* is created and started in background. Line 4 is needed in case the container was already created before and just has to be started. Lines 6 to 8 serve the same purpose for the shared database. On line 10, the host monitor application is started and its output is redirected to the specified log file. Line 12 builds a Docker Image with the tag *fogcontrolnode* by interpreting and executing the Dockerfile in the current working directory. The mentioned Dockerfile is explained in the next paragraph. The remaining lines remove possibly available old containers and start a container with the previously created Docker Image. In more detail, the last three lines of the same command add the Docker Hook by defining two volume commands, and links the database to the container.

To build the Docker Image, a construction plan, i.e., Dockerfile, is needed. The Dockerfile to create the fog control node image is stated in Listing 5.3. The differences between the fog cells' Dockerfile and the fog control nodes' Dockerfile are the name of the

JAR file and the ports to expose. In case of a fog cell, the exposed port would be 8081.

Listing 5.3: Fog Control Node Dockerfile

```
FROM hypriot/rpi-java
ADD fogcontrolnode-0.0.1-SNAPSHOT.jar .
ADD main.properties .
EXPOSE 8080
ENTRYPOINT ["java","-jar","fogcontrolnode-0.0.1-SNAPSHOT.jar"]
```

The base image *hypriot/rpi-java* builds the basis for the main application Docker Containers running on the Raspberry Pis. This very image was chosen due to its light-weight Java runtime based on the ARM processor architecture. After the base image definition in the first line, the next two lines add the JAR executable and the needed property file to the container. Making the container accessible from outside the container, port 8080 is exposed. At the end of the file, the initial entry point command that is executed right after the deployment of the service is defined.

5.4.2 Cloud Setup

The cloud requires a few setup steps to work as described in the previous chapters. Since this project utilizes an OpenStack private cloud, only the setup of this very cloud software is covered. The basic setup of the OpenStack project is described in a tutorial provided by the cloud environment providers, i.e., the Distributed Systems Group Vienna. This tutorial explains how to create a key pair to connect the cloud, configure security groups, download cloud credentials, and manually start instances and assign a public IP, i.e., a floating IP.

In the use case scenario applied to evaluate the developed framework, it is needed to manually deploy a CoreOs instance and assign a floating IP to it. After connecting to the VM via SSH, a Redis database needs to be started. An example command to easily start a Redis database without further downloads is stated in Listing 5.4.

Listing 5.4: Start Redis DB Container

```
docker run -d -p 6385:6379 --name redisCloud redis:latest
```

To make the service and cloud database accessible from outside the VM, one needs to add the ports 6385 and 8200 to the used security group of the just started VM. This VM embodies the cloud database used for the *fogframe/cloud-service* Docker Image. A crucial point regarding the setup of this *fogframe/cloud-service* is to make sure the image utilizes the correct floating IP of the created cloud database in the properties file.

Beside the OpenStack setup, we need to configure and start the cloud-fog middleware. Having fulfilled the requirements described above, only the database container has to be started before starting the Java application. The provided bash script *startDB.sh* starts the database containers.

5.4.3 Environment Setup

The environment setup is a crucial installation part, since the whole device communication and interaction relies on it. The environment consists of a wireless Access Point (AP) and the connected fog and cloud devices, resulting in an interconnected environment that enables the communication in the set up network topology. First, an AP has to be configured correctly. The most important points when configuring the AP are the WiFi security that needs to be set to WPA2 to ensure no external intruder can penetrate the network, and the static IP mapping of the connected devices. The static IP mapping is needed to simplify the connection and setup of the diverse devices. Having set up the AP successfully, all the devices can be booted and connected to the newly created wireless network. To make sure every device is correctly connected to the network, the IP table of connected devices in the web interface of the AP can be investigated.

After all the Raspberry Pis are running and connected to the network, one needs to manually start the corresponding application containers. When starting the applications, it is substantial to start the containers from top to bottom according to the topology. This condition is required because the devices need to pair with their parent during the application startup before executing anything else. Hence, in an example scenario consisting of the cloud-fog middleware, one fog control node, and two connected fog cells, the order would be as follows: cloud-fog middleware, fog control node, fog cell$_1$, fog cell$_2$. The order of the bottom-most children is not relevant though. In the case when a device with any connected children fails, the failing device and all connected children have to be restarted or manually repaired to ensure correct operation.

5.5 Execution

The subsequent sections provide detailed information on how to use the developed fog computing framework with the three use case scenarios described in Section 4.1.4. These exemplary execution samples give an insight on how to use or extend the framework for future work.

5.5.1 Development, Testing, and Evaluation of an IoT Service

The first use case scenario explains how to register a newly developed service in the fog landscape and execute it without further knowledge of the infrastructure, communication, and service deployment. The fog computing framework enables users to develop services by providing a service key, Dockerfile, Docker Volumes, ports to expose, and a flag indicating whether the container requires privilege rights. The service key is a unique identifier to unambiguously identify a Docker Image. Docker Volumes empower the container to use resources located in the host file system. Privilege rights equip the Docker Container with the almost same capabilities as the host. This enables such use cases as using Docker within Docker.

After the elicitation of the Docker Image information, a JSON file to register the service has to be created. The JSON file needs to be well formed as the example JSON in Listing 5.5 demonstrates.

Listing 5.5: Docker Image JSON Sample

```
{
  "serviceKey": "temp-hum",
  "dockerfile": "FROM jonasbonno/rpi-grovepi\n
    RUN pip install requests\n
    RUN git clone https://github.com/keyban/fogservice.git\n
    ENTRYPOINT [\"python\"]\n
    CMD [\"fogservice/service.py\"]",
  "volumes": "/dev/i2c-1:/dev/i2c-1",
  "exposedPorts": ["8105"],
  "privileged": true
}
```

This example JSON registers a service with service key *temp-hum* that pulls a specific git repository and executes the included *service.py* file. Additionally, a volume mapping is listed, the port to get exposed is 8105, and the service requires privilege rights to operate correctly. This stated Docker Image is the construction plan for the temperature and humidity reading service used in the evaluation chapter.

Enabling the execution and evaluation of this service requires the registration of the Docker Image in the fog landscape. Therefore, the user needs to send a POST HTTP request to a fog control node in the system. To make this possible, the user requires IP address information of a fog control node running in the fog landscape. With the IP address and the API URL, the URL is built as follows: *http://<IP>:8080/shareddb/register*. When sending the request, it is essential to add the content type as header and the created JSON file as body. An example request, using *curl*[6] as HTTP data transferring tool, can be seen in Listing 5.6.

Listing 5.6: Docker Image Request Command

```
curl -H "Content-Type: application/json; charset=UTF-8" -X \
     POST -d @image.json http://<IP>:8080/shareddb/register
```

In case the URL was correct and the corresponding fog control node was up and operating correctly, the response contains a header with the URL and a status flag indicating if the service could be created or not. If the flag is false there could be several problems, either the service-key is already assigned, or some other error occurred. For more detailed error information the console output of the specified fog control node needs to be checked.

Assuming the registration worked correctly, it is now possible to request the execution of the just registered service. To achieve the deployment of the registered service, one

[6] https://curl.haxx.se/

needs to create an application JSON and send it to a fog control node. An application consists of a total service duration to define when the services have to be stopped, and a list of task requests defining the services to be deployed. The duration field is specified in minutes and can be set to infinity by setting it to -1. A task request has to contain the service type, service key, and two flags indicating whether the service can only be executed in the cloud or the fog. The URL to send this request to is *http://<IP>:8080/reasoner/taskRequests*. The Listings 5.7 and 5.8 provide an example command and the corresponding JSON file required to execute the request.

Listing 5.7: A Command to Send the Task Requests for Execution

```
curl -H "Content-Type: application/json; charset=UTF-8" -X \
     POST -d @app.json http://<IP>:8080/reasoner/taskRequests
```

Listing 5.8: Single Task Request Application JSON Sample

```
{
  "duration": "-1",
  "requests": [
    {"serviceType:"t1", "serviceKey":"temp-hum",
      "cloudTask":false, "fogTask":true}
  ]
}
```

The successful deployment of the service is confirmed with a JSON response containing a status flag indicating the deployment success, a header with the URL, and a payload with the deployment time in seconds. To check whether everything worked as intended, either the command lines of the running devices, or the HTML status web page of every component can be investigated. The status web page contains device-dependent information including IP, port, device type, children, parent, registered Docker Images, and running Docker Containers. To visit the status web page of a device, the following URL with the corresponding device IP and device port have to be entered in a web browser: *http://<IP>:<PORT>/*

5.5.2 Execution of an IoT Application

Another use case scenario is the execution of an IoT application consisting of several already registered Docker Images. To achieve the deployment of several services, a JSON file representing the application needs to be built and sent to the same URL as in the previous use case scenario. A sample JSON file consisting of four task requests, forming the application, is presented in Listing 5.9.

Listing 5.9: Multiple Task Requests Application JSON Sample

```
{
  "duration": "5",
  "requests": [
    {"serviceType:"t1", "serviceKey":"temp-hum",
      "cloudTask":false, "fogTask":true},
    {"serviceType:"t1", "serviceKey":"temp-hum",
      "cloudTask":false, "fogTask":true},
    {"serviceType:"t2", "serviceKey":"busy-image",
      "cloudTask":false, "fogTask":false},
    {"serviceType:"t4", "serviceKey":"cloud-service",
      "cloudTask":true, "fogTask":false},
  ]
}
```

The response is equal to the one explained in the previous use case scenario. An important difference to the previous use case scenario is the execution condition an application has to follow. An application is only successfully deployed if all consisting task requests are deployed without error. Meaning, if one task request can not be deployed, the whole application fails and all the already deployed services are stopped.

The result of the application deployment can be checked on the status web pages and console outputs of the different devices, and the database contents of the cloud database. The contents of the cloud database can be requested by calling the following URL: *http://<IP>:8200/db/*. The exemplary curl request is stated in Listing 5.10.

Listing 5.10: Get Cloud Database Contents

```
curl -X GET http://<IP>:8200/db/
```

5.5.3 Development, Testing, and Evaluation of a Resource Provisioning Approach

The last and most advanced use case scenario is the development, testing, and evaluation of a resource provisioning approach. To improve, exchange, or extend the currently implemented resource provisioning approach, the developer or researcher needs to create a new Java class implementing the following interface in Listing 5.11.

Listing 5.11: Resource Provisioning Java Interface

```
public interface IResourceProvisioning {
    ApplicationAssignment handleTaskRequests(
     Set<Fogdevice> children, Set<TaskRequest> requests)
 throws InterruptedException, Exception;
}
```

Further conditions the implementation must conform to are: (i) the developer needs to make sure the *MAX_CONTAINERS* constant is set, the method needs to return an application assignment consisting of (ii) the successful task assignments, and (iii) the open task requests that could not be deployed. The open task requests then are propagated to the cloud for deployment.

The new resource provisioning approach can be tested and evaluated by sending several task requests into the system using the same commands described in the previous scenarios. The reasoner will execute the implemented resource provisioning approach and deploy the services accordingly. The service deployment resulting of the newly implemented resource provisioning approach can be checked by investigating the status web pages or the console output of the running devices.

5.6 Resource Provisioning

The basic resource provisioning approach implemented in the course of this work, is a first fit heuristic algorithm. Since the resource provisioning is not in the focus of this work, the approach is kept simple and extensible to be improved or replaced by further resource provisioning approaches. The general idea of the developed resource provisioning algorithm stated in Algorithm 5.1 is to loop over the sorted children fog devices and sorted incoming task requests, check whether the child is able to host another service and deploy it to the first fitting child.

The algorithm takes a set of children fog devices, which can be fog control nodes or fog cells, and the incoming task requests as input. Lines 1 and 2 initialize the needed assignments and the round counter field. In lines 3 and 4 the requests and children are sorted according to the service type in order to improve the loop performance and deployment time. Lines 5 to 7 start the loops over the sorted children followed by the loop over the children's service types and the sorted requests. Hence, iteration over all children and all their service types followed by all requests. Thus, every child iterates over all its service types and gets assigned all the suitable requests before continuing to the next child.

To make sure a child only gets assigned services with the correct service type, line 8 checks if the child service type fits the request service type. In lines 9 and 10, the RAM, storage, and CPU utilization, and the amount of already deployed containers is requested from the child. With this information, line 11 checks if the utilization conforms the defined watchdog rules, e.g., CPU $< 80\%$, and if the maximum amount of deployable containers is not exceeded.

In case the child is able to host another service, a deployment request is sent to the child. In the event of successful service deployment, the child sends back detailed data on the deployed Docker Container needed to stop or migrate it. Being able to keep track of all the deployed services and their location, an assignment consisting of the child, task request, and the deployed container is created and returned at the end of the algorithm. Line 14 removes the successfully executed task request from the input set making sure a task request is deployed only once.

Line 19 is reached after a child looped through all its service types and all input task requests. This if-condition is only valid if the outer most children loop is finished, the provisioning round counter is smaller than the maximal defined rounds, and if there are still unhandled task requests. If this is the case, the round counter is increased and the children iterator is re-initialized to restart the provisioning with the remaining task requests. This fault tolerance mechanisms serves the purpose of avoiding utilization and other reading errors.

After all the requests are handled or the maximum amount of provisioning rounds is exceeded, the created assignments and the still open requests are returned. The open requests are the remaining task requests in the sorted request set, since all the handled requests are removed from the set after successful deployment.

The information presented in the current chapter gives an overview on the designed and developed fog computing framework, explains essential component requirements and setup steps to execute the framework, and describes exemplary use case executions followed by the implemented resource provisioning algorithm. This content includes the answer of the third research question “How to realize a fog computing framework that manages the fog landscape and executes IoT services?” stated in Section 1.2.

Algorithm 5.1: First Fit Resource Provisioning Algorithm

```
Input: Set<Fogdevice> $children$, Set<TaskRequest> $requests$
Output: List<TaskAssignment> $assignments$, List<TaskRequest>
        $openRequests$
// init fields
1  $assignments \leftarrow []$;
2  $round = 0$;
// sort children and task requests according to the service type
3  $sortedRequests \leftarrow$ sortByServiceType(requests);
4  $sortedChildren \leftarrow$ sortByServiceType(children);
// loop over children, service types, and task requests
5  for $child \in sortedChildren$ do
6      for $serviceType \in child.serviceTypes$ do
7          for $request \in sortedRequests$ do
8              if $serviceType == request.serviceType$ then
9                  $utilization \leftarrow$ getUtilization($child$);
10                 $containers \leftarrow$ getContainerCount($child$);
11                 if $checkRules(utilization)$ and
                   $containers < MAX\_CONTAINERS$ then
                       // deploy container and save the assignment
12                     $container \leftarrow$ sendDeploymentRequest($child, request$);
13                     assignments.add($child, request, container$);
14                     sortedRequests.remove($request$);
15                 end
16             end
17         end
18     end
       // check if the iterator is finished, the max rounds is not yet
          reached, and not all requests are assigned
19     if !$sortedChildren.hasNext()$ and $round < ROUNDS$
       and $sortedRequests.size() > 0$ then
           // increase the round counter and reinitialize the children
              iterator
20         $round = round + 1$;
21         sortedChildren.reStart()
22     end
23 end
// return the assignments and the still open requests
24 return $assignments, sortedRequests$
```

CHAPTER 6

Evaluation

Evaluation is the process to assess whether a subject, tool, or general outcome serves the aimed purpose and fulfils the goals according to accurately defined measurement metrics. This chapter covers the evaluation of the designed and developed fog computing framework based on an evaluation setup and specified evaluation scenarios. These scenarios are evaluated with respect to the chosen metrics discussed and supported by generated plots and data tables.

The chapter starts off with the definition of the evaluation setup in Section 6.1 followed by the definition of diverse evaluation scenarios in Section 6.2. Section 6.3 presents the metrics to evaluate the scenarios and critically discusses the results of the corresponding scenarios.

6.1 Evaluation Setup

In order to thoroughly evaluate this project, an evaluation setup with clearly specified boundaries, metrics, and included devices needs to be defined. The realized evaluation setup in this work is a network topology including the required devices to properly analyze the developed fog computing framework. Figure 6.1 gives an overview on the established real-world evaluation setup used to conduct the experiments. In more detail, the topology consists of four distinct devices, a cloud-fog middleware, fog control nodes, fog cells, and sensors. The cloud-fog middleware builds the top level of the topology and is executed on a Macbook Pro Early 2011 connected to the OpenStack cloud environment used to upload and resolve task requests. The rest of the components included in the setup are deployed on Raspberry Pis as has already been described in the previous chapter. Furthermore, fog control node 1 (FCN_1) is directly connected to the cloud-fog middleware (CFM) and embodies the root fog control node for the subjacent fog devices. The fog colony controlled and orchestrated by FCN_1 consists of FCN_2 and FCN_3. Both, FCN_2 and FCN_3, supervise connected fog cells which process data from connected IoT devices.

The connected IoT devices, in this setup, are sensor modules consisting of a temperature and humidity sensor. These sensor modules are connected to the respective Raspberry Pis by sensor modules called GrovePi[1].

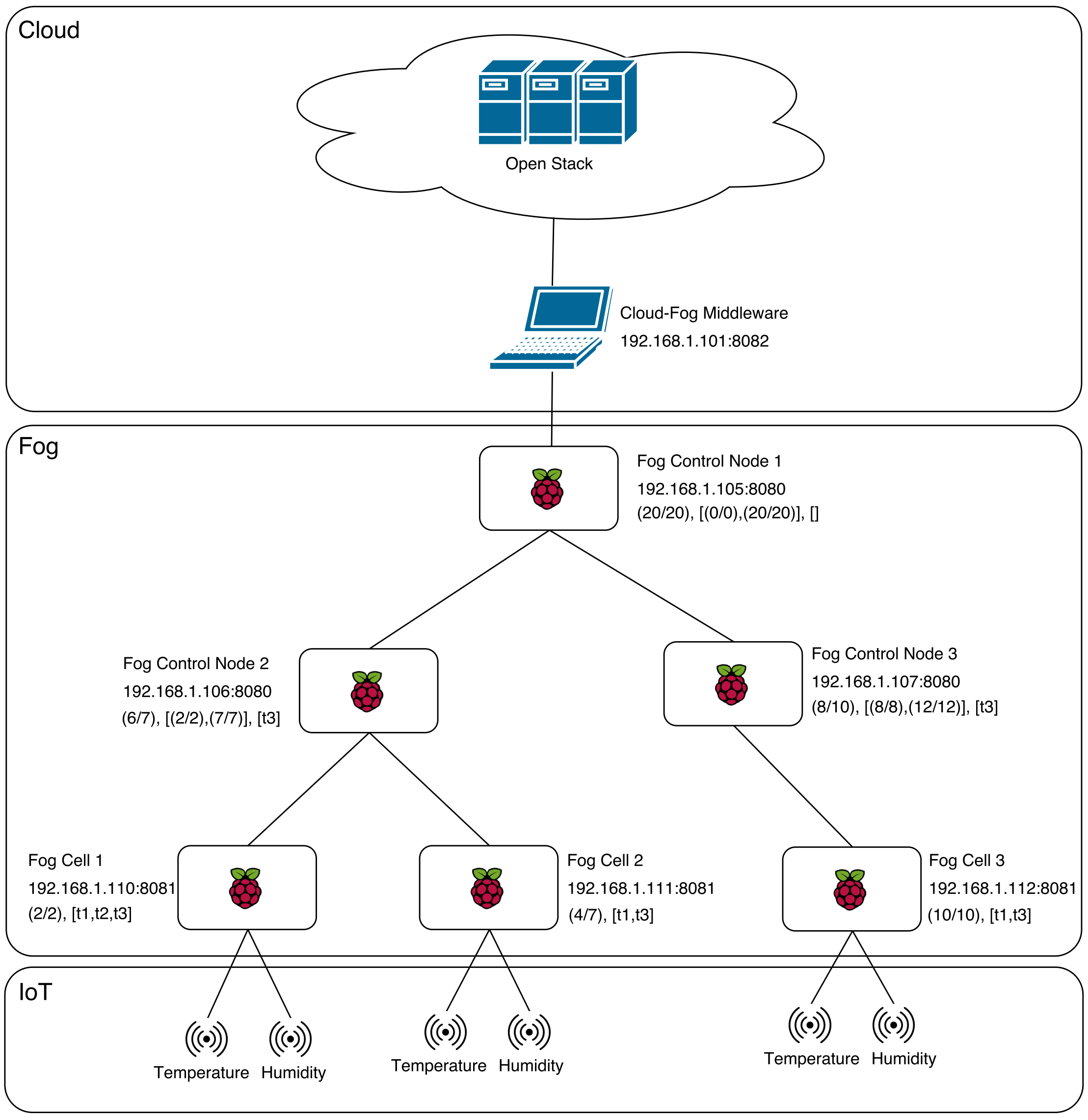


Figure 6.1: Evaluation Setup

The topology network is set up as a wireless LAN network provided by a powerful Linksys wireless AP. This AP is connected to the Internet and works as a gateway to connect every Raspberry Pi to the Internet as well. In the developed real-world test-bed

[1]https://www.dexterindustries.com/grovepi/

every component needs to be connected to the Internet since the fog services require the ability to download Docker Image data in order to create and deploy dynamic services.

In the network topology to evaluate the framework, every device in the fog layer contains various information (from top to bottom): (i) device name, (ii) IP address and port, (iii) device location, (iv) location range, and (v) the list of service types the device is able to process. The last line of the device information can differ according to the device type, e.g., only fog control nodes have a location range. The device location and location range are necessary to assess the responsible parent for newly joining fog devices. The location range parameter defines the geographical area a fog control node is responsible for. Hence, every device requesting a parent with its own device location gets the parent returned that covers the area the requesting device is located in. Figure 6.2 presents the location grid of the evaluation setup.

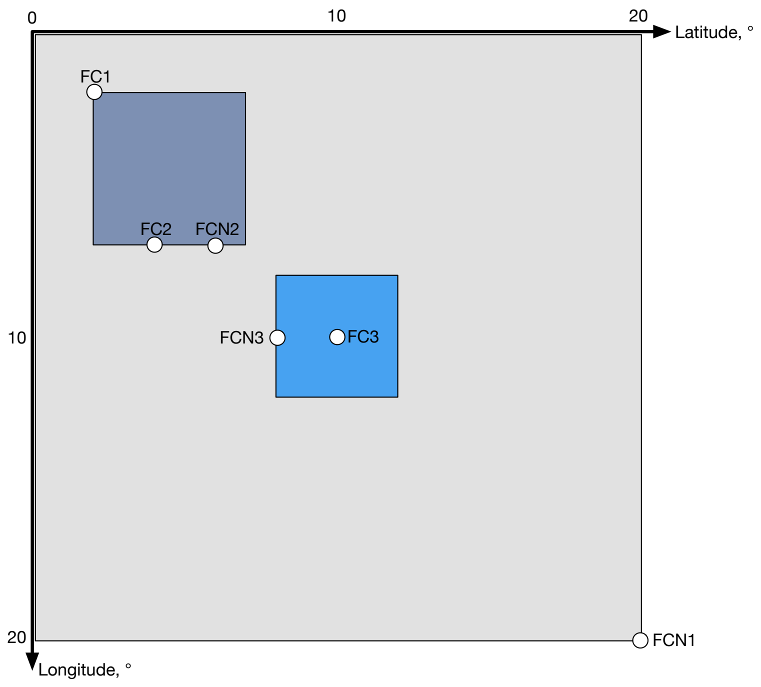


Figure 6.2: Device Location Grid for Parent Assessment

To empower a reproducible evaluation, we predefine application requests consisting of a set of task requests and an application duration, to be requested at a fog control node and deployed in the system. The chosen applications are designed to show the implemented functionalities and mechanisms, and to evaluate the system with respect to the metrics defined in Section 6.3.1. The application requests sent to the system have to comply with the following conditions: (i) the requested service types of the fog services need to conform to the service types of the fog devices, (ii) the requested cloud services need to be pushed to the Docker Hub repository with the prefix *"fogframe/"*

before requesting, and (iii) all required application fields have to be filled according to the exemplary execution of the use case scenarios provided in Section 5.5. Even if the application execution complies with these stated conditions, an application may fail. Possible reasons are (i) the OpenStack cloud environment is overloaded, (ii) the floating IP addresses are exhausted, (iii) the fog landscape is overloaded, or (iv) the heuristic does not place the services optimally, resulting in overloaded fog devices. The amount of deployable cloud VMs is theoretically unlimited, however, in practice is limited by available cloud resources and the floating IP limitation of three assignable floating IPs. The concrete applications to be used in the remaining evaluation are:

1. **Data processing application with sensor readings (A1)**
   Task Requests:
   (Service Key: temp-hum, Service Type: t1, Amount: 5),
   (Service Key: busy-image, Service Type: t2, Amount: 8),
   (Service Key: busy-image, Service Type: t3, Amount: 26)
   Duration:
   5 minutes

2. **Data processing application (A2)**
   Task Requests:
   (Service Key: busy-image, Service Type: t3, Amount: 5 to 80)
   Duration:
   5 minutes

3. **Cloud-fog data processing application (A3)**
   Task Requests:
   (Service Key: busy-image, Service Type: t3, Amount: 15),
   (Service Key: cloud-service, Service Type: t4, Amount: 15)
   Duration:
   1 minute

4. **Varying data processing application (A4)**
   Task Requests:
   (Service Key: busy-image, Service Type: t1,
   Amount: according to input function),
   (Service Key: busy-image, Service Type: t3,
   Amount: according to input function)
   Duration:
   varying between 1 and 5 minutes

All evaluation scenarios described in the next section utilize these defined applications. The used applications in the respective scenarios are specified in the scenario description.

6.2 Evaluation Scenarios

Aiming at a holistic evaluation of the developed fog computing framework, well-elaborated evaluation scenarios are required. To cover the evaluation of the whole range of functions the framework provides, six evaluation scenarios are investigated. These six experimental evaluation scenarios range from the evaluation of specific functionalities and mechanisms to more specific assessments of processing cost, deployment time, and varying service deployment.

6.2.1 Device Failure

The first experimental scenario evaluates the situation when a connected and successfully paired fog cell loses its connection to the fog landscape due to a device failure. The device failure can be a hardware, software, or communication problem. The device failure is simulated by stopping the main fog cell application container at the chosen failing device. In this experiment, the data processing application *A1* is used, and the device to fail is FC_1. This setting enables the investigation of specific device failing events and the resulting replanning mechanisms. The reason FC_1 has been chosen is because of its service types. FC_1 is the only fog cell with the capability of processing service type *t2* and is therefore unique in its capabilities. If this fog cell fails, the requested services with service type *t2* can not be deployed in the fog landscape anymore and need to be propagated to the cloud. As a result, this scenario shows the device failure event and its according service migration mechanisms in the cloud and fog environment.

6.2.2 Device Accedence

In order to show the device accedence event and its coherent mechanisms and service migrations, a new fog cell that is not yet in the fog colony is added. The selected fog cell to join the fog landscape is FC_1. It was chosen because of the same reasons stated in the previous scenario. Furthermore, this scenario also uses application *A1*. What happens in this experiment is that the new device is joined and able to have these special *t2* services deployed. Consequently, the *t2* services which were previously deployed in the cloud are stopped and migrated to the newly started FC_1. Additionally, the VM hosting these services remains with no running services and can therefore be stopped.

6.2.3 Overload

This scenario shows the developed mechanisms of how the framework reacts when a fog device is overloaded. Again, the topology setup and the application to be deployed are similar to the first scenario. Since the fog control nodes are in charge of both deploying services and handling the communication traffic coming from connected fog devices, the fog control node is the likeliest device to overload. Especially FCN_2 is endangered to overload because of two connected fog cells. The overload of FCN_2 is simulated by opening several SSH connections and running a small but resource-intensive script that prints out a random String infinitely. After some parallel executions of this script, the resource utilization exceeds at least one defined utilization rule and thereby triggers the overload event. This overload event executes the following overloading policy. The reasoner migrates a random service from the overloaded device to another one. With this mechanism services are migrated to suitable devices until the overloaded device is not overloaded anymore.

6.2.4 Cost Assessment

Since the scenarios are not only used to analyze and evaluate specific functionalities of the fog computing framework, but to show other benefits and planned outcomes, the service deployment cost, depending on the number of deployed services, is assessed. In other words, this scenario uses the second predefined application type *A2* and issues this application with a service amount ranging from 10 up to 80 services of type *t3*. This specific service type is applicable in both cloud and fog environment, and thereby enables a suitable comparison. Regarding cost calculation, the cost is calculated by the number of deployed VMs in the cloud environment and the Billing Time Unit (BTU). The assumed BTU and cost used for this scenario are *one hour* and *0.30$* per BTU. In addition, we assume the ownership of the fog devices. Therefore, the fog cost can be neglected.

6.2.5 Deployment Time Assessment

This scenario investigates the service deployment times in the different environments. Aiming at an expressive evaluation scenario, the deployment times, split into cloud and fog service deployment times, are compared and discussed. The deployment times are automatically calculated by the framework and are statistically processed over five evaluation rounds. The application used for this scenario is *A3* with an equal amount of task requests deployed in the cloud and the fog landscape. This application setting shows the enormous deployment time differences between deploying services in the contrasting environments.

6.2.6 Varying Service Deployment

To show the service deployment distribution in time, different input functions of requested applications are applied, i.e., constant and pyramid input functions. In this experiment, the *A4* application is used with varied application durations and service amounts depending on the input function. In general, *A4* consists of two types of services deployable in the cloud as well as in the fog landscape. Additionally, the duration of the application varies according to the input specifications. In this scenario the aim is to demonstrate how services are distributed on running devices over time.

6.3 Results

After specifying the relevant evaluation metrics, this section includes the concrete evaluation results of the previously described evaluation scenarios.

6.3.1 Metrics

Evaluation metrics serve the purpose of measuring the performance and applicability of the provided outcome, and to show whether the results fulfil the stated goals and success criteria. An essential information regarding the evaluation metrics is that the cost of the upfront building of the fog landscape and all its devices and resources is omitted. Hence, it is assumed that these resources are owned by the users of the framework. The crucial metrics to evaluate the fog computing framework are described in the following list:

- **Number of deployed services in the cloud environment**
  Let $services_{cloud}$ be the number of deployed services in the cloud environment.
- **Number of deployed services in the fog landscape**
  Let $services_{fog}$ be the number of deployed services in the fog landscape.
- **Total number of deployed services**
  The total number of deployed services is the sum of the deployed cloud and fog services formalized in equation (6.1).

  $$services_{total} = services_{fog} + services_{cloud} \tag{6.1}$$

- **Cloud service deployment time**
  The cloud service deployment depends on several conditions. First, the deployment time depends on whether there have already been free VMs running, or if a new VM has to be started. Let res_{VM} be the amount of free VMs in the cloud environment. Furthermore, the deployment time depends on the condition whether the required Docker Image is locally available or not. Let d_i be a Docker Image with a service key i. Equation (6.2) describes the three cases to differentiate. First, in case no free VM is available, the cloud service deployment time equals the sum of the VM booting time t_{boot}, of the time to pull the Docker Image $t_{pull\_d_i}$, and the startup time of the Docker Container t_{start}. Second, a free VM is available but the Docker Image has to be pulled. Third, both, a free VM and the required Docker Image are available.

  $$t_{cloud} = \begin{cases} t_{boot} + t_{pull\_d_i} + t_{start}, & \text{if } res_{VM} = 0 \\ t_{pull\_d_i} + t_{start}, & \text{if } res_{VM} > 0 \text{ and } d_i \text{ is not available} \\ t_{start}, & \text{if } res_{VM} > 0 \text{ and } d_i \text{ is available} \end{cases} \tag{6.2}$$

- **Fog service deployment time**
  The fog service deployment time differs from the cloud deployment time since

in the fog landscape no VMs have to be started before deploying the service containers. Hence, equation (6.3) includes the cases when the Docker Image is not locally available on the according fog device, and when the Docker Image is locally available.

$$t_{fog} = \begin{cases} t_{pull\_d_i} + t_{start}, & \text{if } d_i \text{ is not available} \\ t_{start}, & \text{if } d_i \text{ is available} \end{cases} \tag{6.3}$$

- **Total service deployment time**
  The total service deployment time is the sum of the cloud and the fog service deployment time.

$$t_{total} = t_{fog} + t_{cloud} \tag{6.4}$$

The deployment time is a time interval between the point in time when the task request is issued until all requested services are deployed. This metric is then reduced to a service deployment time per request differentiating between the deployment time in the cloud and the fog landscape. Additionally, the measured values are statistically analyzed.

6.3.2 Discussion

In this part of the thesis the results of the defined evaluation scenarios are analyzed and discussed. In the following evaluation, the efficiency of the fog computing framework performing in various situations is shown and supported by visualization and interpretation of the received results.

Scenario I: Device Failure

The experiment of the first evaluation scenario analyzes the case when a fog device connected to a second-level fog control node fails. The setup of this evaluation is the following. The whole network topology visualized in Figure 6.1 is set up and operating correctly. After all services of the requested application *A1* are successfully deployed, the failure of FC_1 is simulated. The expected result involves the migration of all services, deployed on the failing device, to other fog devices or a cloud VM with suitable capabilities. Since in this case no other host in the fog landscape has the suitable capabilities to deploy services with service type *t2*, the services need to be migrated to a cloud VM. Consequently, a VM is started in the OpenStack cloud environment and the services are migrated.

The concrete result of the experiment is visualized in Figure 6.3 that depicts the number of deployed services in correlation with specific time events. In Figure 6.3 there are six different service processing devices. Event 1 embodies the request of the predefined application *A1*. After all services are deployed successfully, Event 2 takes place. Event 2 symbolizes the failing of FC_1 and is followed by the activation of the device failure mechanisms, described in Section 4.2.2, in Event 3. Event 4 shows the automatic stopping of the application after the specified application duration of 5 minutes is over. The root fog control node (FCN_1) is not depicted in the plot since it would be overloaded by additional services due to the reasoning, replanning, and communication processes.

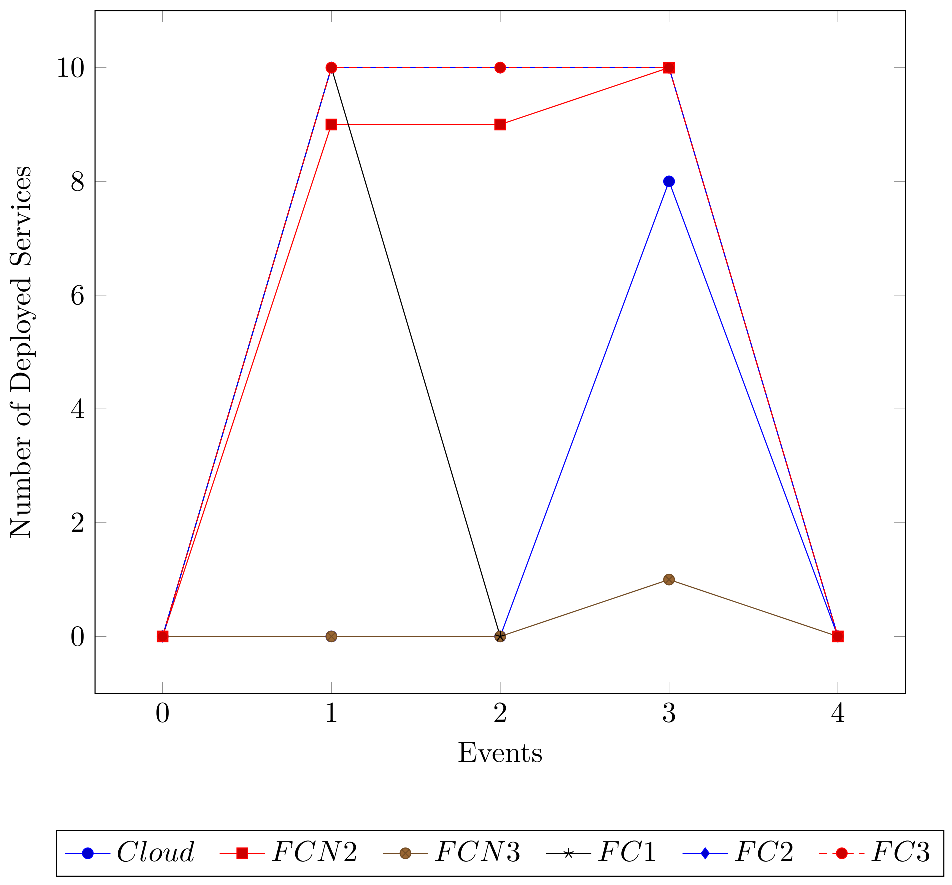


Figure 6.3: Device Failure over Time Events and Number of Deployed Services

As a result of the resource provisioning, one can observe that after the first event, the placement is as follows: FCN_2=9, FCN_3=0, FC_1=10, FC_2=10, and FC_3=10. Because of the failing FC_1 in Event 2, the service count of FC_1 decreases to 0. At Event 3, the system starts the replanning mechanism and migrates the failed services of FC_1 to other devices. Consequently, FCN_2 and FCN_3 get one service of service type *t3* assigned, and the rest of the services with the service type *t2* are deployed in the cloud because no device in the fog landscape has the capability to deploy them. After the specified period of 5 minutes, Event 4 shuts down all the services.

This mechanism shows how the implemented framework reacts, when a device with a specific set of running services fails. The framework makes sure that all failed services are deployed on new devices, and the fog resource utilization is maximized. Meaning, the cloud resources are used only if the fog resources are fully utilized or do not have the required capabilities.

Scenario II: Device Accedence

In this evaluation scenario, a new device joins the evaluation network during the execution of the predefined application *A1*. It is essential that all services of the requested application are successfully deployed at the time the new device joins the fog landscape. The setup for this scenario is similar to the setup in the previous scenario, but the fog cell FC_1 is not running and joined later on. The expected result of this scenario is that the migration of services takes place from the cloud to the acceded FC_1. In more detail, it is expected that the services of service type *t2*, which are currently running in the cloud, are migrated to FC_1. Moreover, the developed policy is expected to iterate over fog devices having the maximum amount of services deployed and migrate one from every affected device to FC_1. Finally, due to the fact that there are no remaining services running in the cloud, the VM is expected to be stopped after the successful service migration.

The actual result of the execution of this evaluation scenario is presented in Figure 6.4. The axes of the diagram are similar to the one used in the first scenario. Event 1, again, is the initial application request at the root fog control node. Event 2 symbolizes the joining of the new device followed by the device accedence mechanisms depicted in Event 3. Event 4 shows the planned shutdown of the application after 5 minutes.

After the successful deployment of the application, the service placement looks as follows: FCN_2=1, FCN_3=10, FC_1=0, FC_2=10, FC_3=10, and Cloud=8. The placement remains the same for Events 1 and 2 as the device accedence mechanisms take place between Event 2 and 3. Due to the accedence of FC_1, the system starts to replan the current service placement in order to save cloud resources and maximize the fog resource utilization. Consequently, since the device accedence policy migrates one service from every device having the maximum amount of containers to FC_1 and tries to migrate all cloud services, the subsequent steps are initiated. First, the cloud services are migrated to the new FC_1 resulting in a deployed service count of eight services. Second, the fog devices with the maximum amount of deployable containers are considered. The current situation of three devices with the maximum amount of containers, FCN_3, FC_2, and FC_3, indicates that the replanning mechanism does not only take into account the joined device but as well the already existing topology. Hence, the reasoner migrates one service from FC_2 and FC_3 to FC_1, and one service from FCN_3 to FCN_2. Thus, at the end of the service migration, the final service placement is: FCN_2=2, FCN_3=9, FC_1=10, FC_2=9, FC_3=9, and Cloud=0.

This experiment shows the device accedence mechanism of the developed fog computing framework. As visualized and described in this evaluation scenario we can sum up that the framework is able to adapt its service placement according to the current demand and available devices in the topology, resulting in cost saving operations because of releasing unnecessary cloud resources. A noteworthy point for future work in this scenario is to account for BTUs in order to fully use the time of the VMs that already have been started.

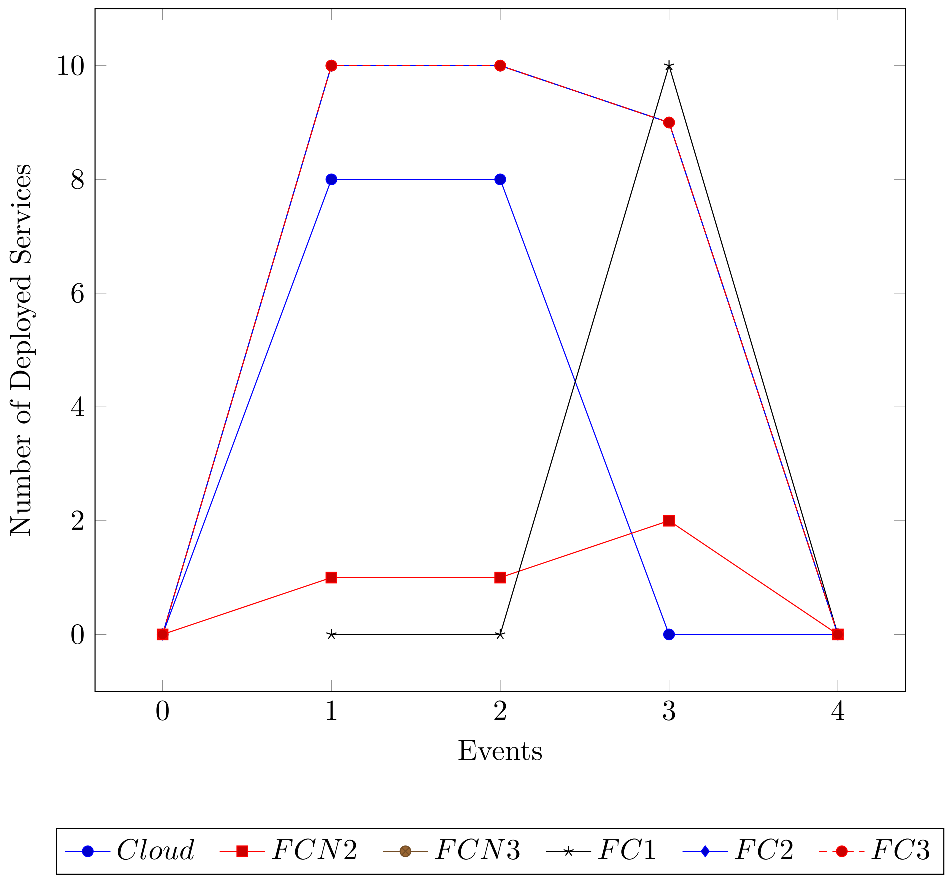


Figure 6.4: Device Accedence over Time Events and Number of Deployed Services

Scenario III: Overload

The third evaluation scenario investigates the condition when a fog control node, specifically FCN$_2$, is overloaded due to the deployed services plus the communication handling of subjacent child nodes. The setup for this scenario is completely similar to the scenario setup of Scenario I. Hence, all devices need to be up and running and the application *A1* is required to be successfully deployed. The expected result the overload mechanism should provide is the migration of enough services of the overloaded device until the device is not overloaded anymore. Since the overload is manually simulated by an endless resource intensive task, the mechanism is expected to migrate services until FCN$_2$ has no deployed services anymore. The six events visualized in Figure 6.5 show the course of the mechanisms handling the device overload. Like in the previous two scenarios, the first event visualizes the initial application request of *A1*. Event 2 is the point when the overload of FCN$_2$ is determined by the system followed by the overload handling and replanning in Events 3, 4, and 5. Between Event 4 and 5, the remaining services are migrated from the overloaded device to a suitable substitute. Event 6, again, is just the

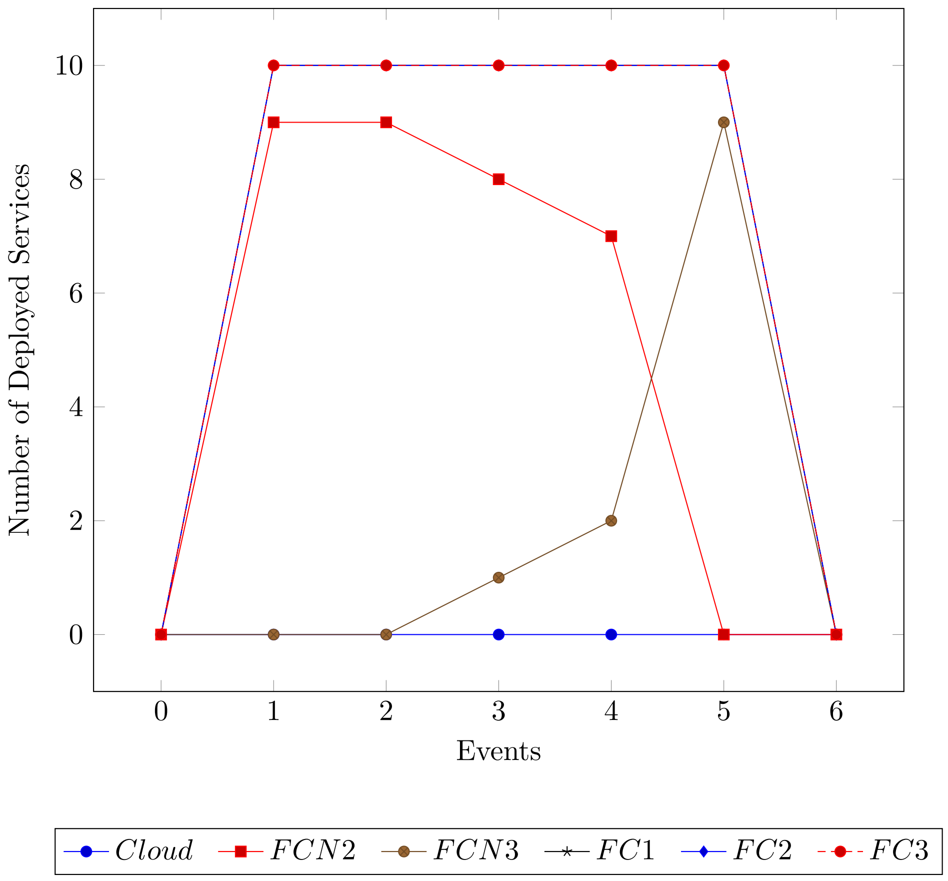
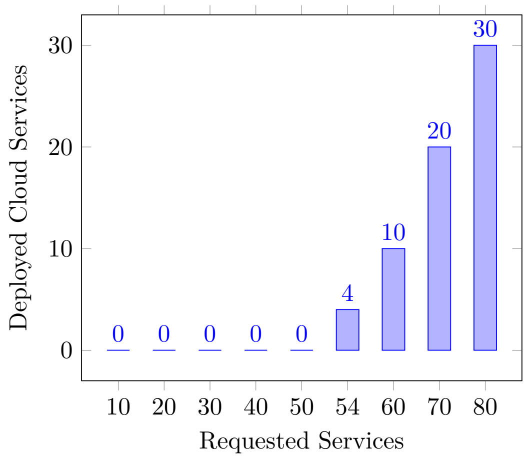


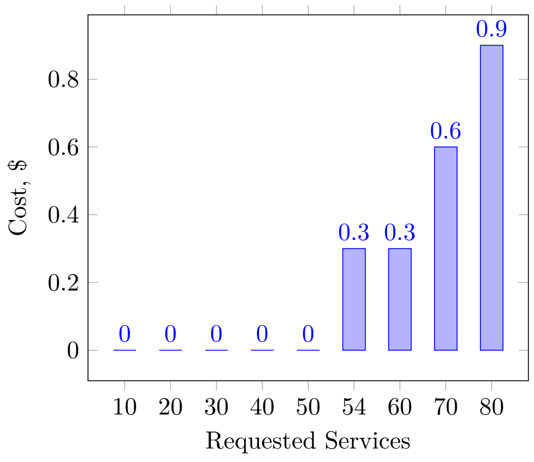
Figure 6.5: Device Overload over Time Events and Number of Deployed Services

planned termination of the running application. Since fog control nodes are responsible for the communication and service data propagation of the subjacent fog devices, it is obvious that deployed services can overload such devices. In this case, the affected FCN_2 is the fog control node with the most child nodes and deploys three services. After a specific amount of time, in Event 2, the fog control node issues the overload event at the root fog control node FCN_1 which starts to migrate services from FCN_2 to FCN_3. Since the overload on FCN_2 is simulated manually, the migration does not stop. Consequently, the affected fog control node is overloaded until all services have been migrated to another suitable device, resulting in three migrated services from FCN_2 to FCN_3. In case the overloaded device does not have any services to migrate anymore, the mechanism prints out a warning and the framework continues with its normal operations.

This evaluation scenario depicted in Figure 6.5 illustrates the overloading policy implemented in the framework. The result shows that the framework is able to handle not only events like device failures and device accedence, but also more specific and demand-dependent events like the overload of a device. This mechanism enables early preventive overload detection and improves service enactment in the fog landscape.



(a) Cloud Services per Requested Services



(b) Cloud Cost per Requested Services

Figure 6.6: Cost Assessment Results

Scenario IV: Cost Assessment

A crucial and omnipresent problem in systems using cloud resources is cost minimization. In order to minimize the cost, a lot of conditions need be taken into account, e.g., BTU, service placement constraints. In this work, we reduce the use of the cloud whenever it is possible and aim to maximize the fog resource utilization and thereby save cost by releasing unneccessarry cloud resources. Furthermore, we concentrate on service placement and fast resource releasing, yet do not consider the full exploit of a BTU. Hence, if an application is finished before other services are requested, the VM is stopped no matter if the BTU is fully used or not. Nevertheless, the consideration of the BTU utilization has to be a part of resource provisioning approaches to be researched in future work.

In this concrete cost assessment scenario, we present the cost benefit by primarily using the fog resources. Since the fog resources are assumed to be already available network devices, the cost for the fog resources is neglected. Additionally, this scenario is only valid for services that can be deployed both in the cloud and the fog environment. For this, the application *A2* is used. This application was varied in terms of the amount of task requests sent. Figure 6.6a visualizes the tested amount of requested services with the deployed services in the cloud. The plot shows that 50 services can be deployed in the fog landscape without the necessity of leasing any cloud resources. The increase above 50 services demands the start of the first cloud VM. Due to the fact that the maximal amount of service containers is limited to 10, every eleventh service above 40 requires the start of a VM in the cloud environment, e.g., 51st, 61st.

The resulting cost of the cloud resource leasing is depicted in Figure 6.6b. As it has already been defined in the scenario description, we assume the BTU to be one hour with cost of 0.30$. Consequently, the resulting cost match the amount of VMs required to deploy the requested services in the cloud. So the cost of 0.30$ for 54 and 60 services result from the single necessary VM, whereas 70 services require two, and 80 services require three VMs to deploy the according amount of services. The results of the experiment were tested with application *A2* consisting of 10 to 80 services of type *t3*. This type embodies a service to be deployed in the cloud as well as in the fog landscape. The maximum amount of 80 services was chosen due to the fact that we are limited by three floating IPs in the OpenStack cloud environment. Therefore, only three VMs can be started, resulting in 30 service deployment possibilities in the cloud in addition to the 50 service deployment possibilities of the five fog devices FCN_2, FCN_3, FC_1, FC_2, and FC_3.

The presented cost assessment provides the results of up to 80 deployed services and the resulting cost. In a nutshell, with the provided evaluation setup every eleventh service request above 40 services requires the start of an additional VM in the cloud. Hence, the first 50 services are deployed in the fog, resulting in no additional cost, whereas the 51st service request requires the start of the first VM in the cloud. Consequently, the developed framework helps to reduce the cost by primarily using fog resources free of charge, before leasing costly cloud resources.

Scenario V: Deployment Time Assessment

Beside functionality and cost evaluation, the service deployment time is an essential metric to take into consideration when evaluating the fog computing framework. The service deployment time defined in Section 6.3.1 consists of the VM startup time, Docker Image download (pull) time, and the starting time of the Docker Container. Obviously, the VM startup time does not exist in the fog landscape thanks to the container virtualization technology. Nevertheless, the assessment also recorded the container startup times, enabling further analysis. In addition to the startup times, the VMs come with another disadvantage. Newly-started VMs do not have previously stored or cached data, e.g., previously used Docker Images, and therefore need to download them every time again. Fog devices, on the other hand, download these Docker Images once and then reuse them in future service deployments. Again, to empower further analysis, the Docker Image download times of the VMs were recored and are presented in the course of this scenario.

The statistically interpreted data is presented in Table 6.1 and Table 6.2 followed by diverse plots visualizing the outcome of the assessment. The computed statistical evaluation data used to analyze the deployment time in the contrasting environments include the minimum, first quartile, average, median (second quartile), third quartile, maximum deployment time, and the standard deviation σ. The data result from five similar application executions of the application *A3* performed in the full evaluation setup. Additionally, the data is refined by calculating the deployment times per request to give an overview on the times relative to the deployed requests. Table 6.2 depicts the same statistical information for the cloud VM startup times and Docker Image download times for further analysis.

In Figure 6.7, the deployment times of the fog and cloud environment are compared. The y-axis on the left-hand side is used for fog results and the y-axis on the right-hand

Table 6.1: Deployment Time Assessment Data

Metric	Amount
Deployed Cloud Services	15
Deployed Fog Services	15
Total Deployed Services	30

Metric	Min	Q1	Avg	Median	Q3	Max	σ
Fog Deployment Time	27.56	28.09	28.87	28.52	28.99	31.20	1.26
Cloud Deployment Time	257.22	258.01	262.09	262.50	265.68	267.05	3.95
Total Deployment Time	285.57	286.21	290.96	290.59	295.57	296.88	4.65
Fog Deployment Time per Request	1.84	1.87	1.92	1.90	1.93	2.08	0.08
Cloud Deployment Time per Request	17.15	17.20	17.47	17.50	17.71	17.80	0.26
Total Deployment Time per Request	9.52	9.54	9.70	9.69	9.85	9.90	0.15

Table 6.2: Additional VM Startup and Image Download Times

Metric	Min	Q1	Avg	Median	Q3	Max	σ
VM Startup Time	36.29	37.52	39.69	38.22	40.36	46.41	3.35
Image Download Time	64.32	65.68	67.05	66.61	67.96	71.00	1.95

side for cloud results. In this plot one can see the considerable difference between fog and cloud deployment times. The maximum fog deployment time is at about 31 seconds, whereas the maximum cloud deployment time is about 267 seconds. The maximum cloud deployment time is approximately nine times higher than the fog deployment time. Obviously, the VM startup and Docker Image download take a huge responsibility in this case (see Figure 6.8b). Due to the fact that the fog devices do not have to download the Docker Images as they have already been cached, the result can seem mis-interpreted. Nevertheless, one can subtract the maximum Docker Image download time from the maximum startup time, resulting in 31 seconds in the fog and 196 seconds in the cloud. The remaining cloud deployment time of 196 seconds is about seven times the fog deployment time. This considerable difference gets illustrated in Figure 6.8a which shows the total deployment time in comparison to the cloud deployment time.

Summing up this scenario one can conclude that the deployment times in the cloud are a lot higher than the deployment times in the fog landscape. Of course, one needs to keep record of the VM startup times and the Docker Image download times, and that fact can also be considered as a disadvantage of the cloud environment. The result of this scenario manifests the clear benefits of the IoT service deployment in the fog landscape compared to the cloud.

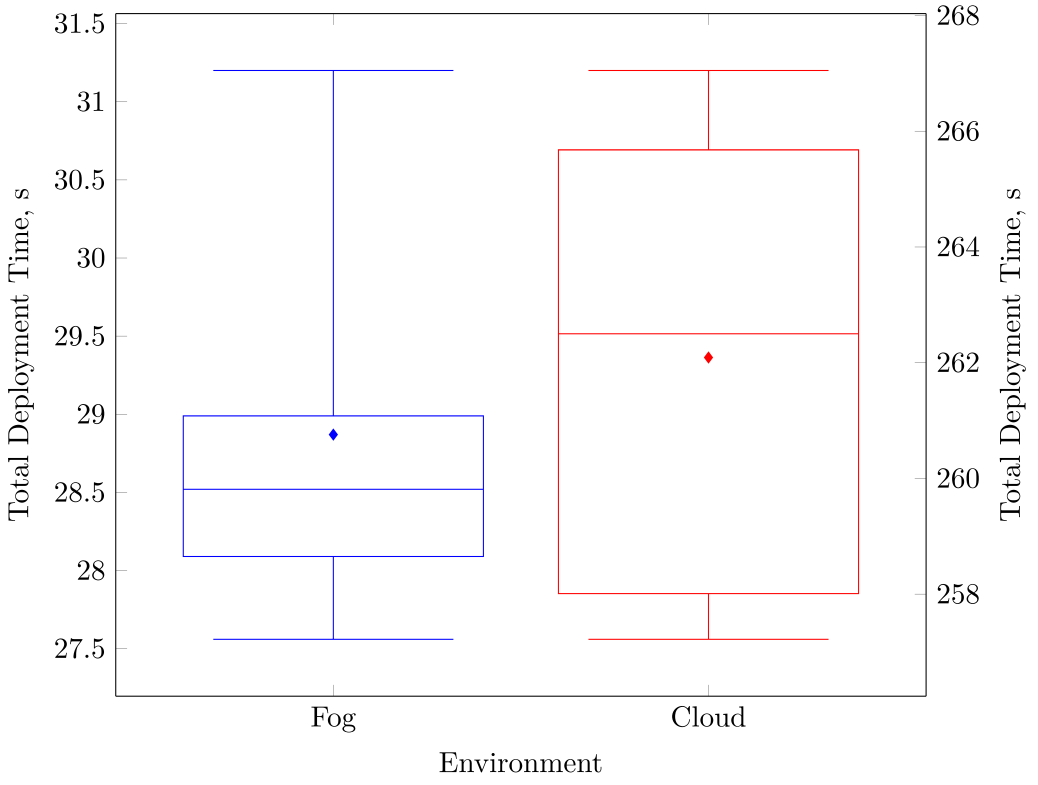
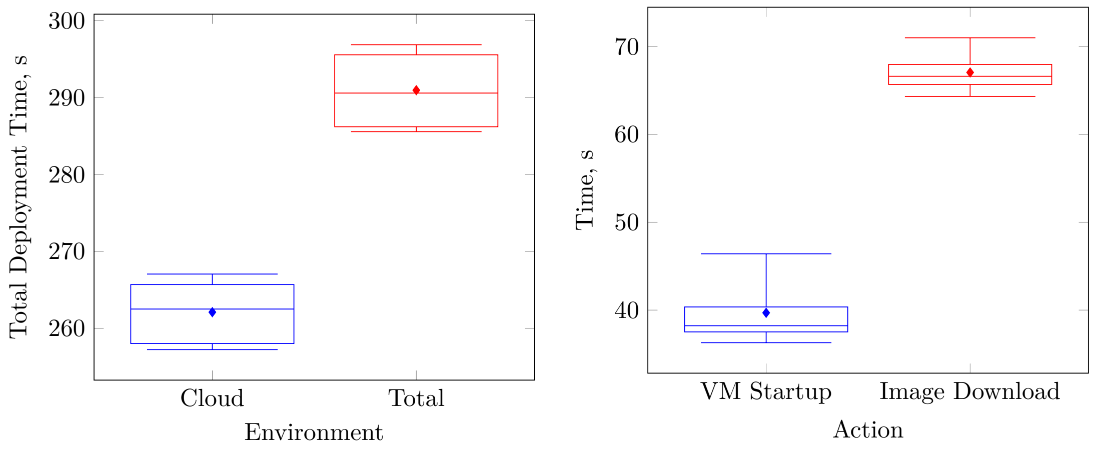
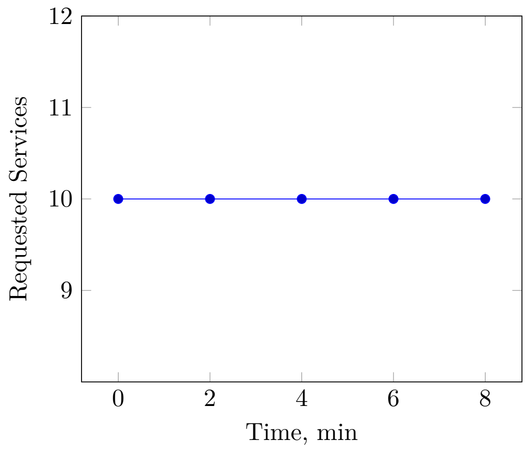


Figure 6.7: Boxplot of the Total Fog and Cloud Deployment Times with 15 Services each

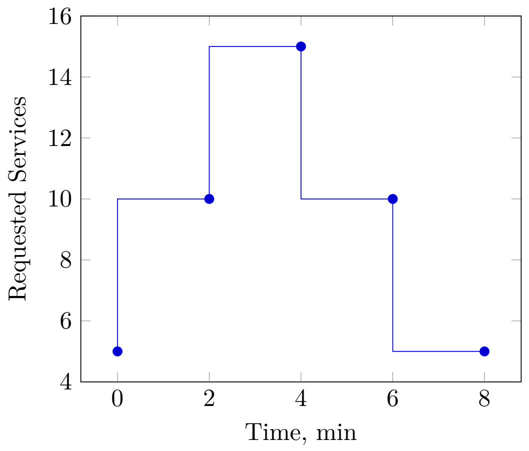


(a) Cloud and Total Deployment Times (b) VM Startup and Image Download Times

Figure 6.8: Deployment Time Assessment Results



(a) Constant Input Function



(b) Pyramid Input Function

Figure 6.9: Input Functions

Scenario VI: Varying Service Deployment

The last scenario emphasizes on the effects of application request variation in the fog computing framework over time. Hence, this experiment defines two diverse input functions visualized in Figure 6.9a and Figure 6.9b. These plots illustrate input functions of application *A4*. Every mark in the plots represents an application request with a service amount specified on the y-axis. Hence, for example in Figure 6.9a, every two minutes an application with 10 services is issued.

The results of this experiment were assessed by the script presented in Listing 6.1 that automatically sends an application request, sleeps the specified 2 minutes, and continues sending the next defined application requests. This specific script in the listing below is requesting the pyramid input function as can be investigated by the changing amount of requested services of service type *t3*.

In this script, another HTTP command line client called *HTTPie*[2] is used. The bash keyword of this tool is *http* and it enables the convenient sending of HTTP requests. In this case we send a POST request to a testing endpoint. The endpoint takes the following URL path arguments: (i) amount of task requests of type *t1*, (ii) amount of task requests of type *t2*, (iii) amount of task requests of type *t3*, (iv) amount of task requests of type *t4*, and (v) the application duration in minutes. Additionally, a *–timeout* argument is set to one second, in order to control the exact time of requesting the next application. As a result, the sleep duration between the application requests is not 120 seconds as expected, but 119 seconds because the missing second is consumed by the connection timeout of the HTTP request. The optional descriptive *date* command is used to show the starting time of every request making sure the process works as intended.

[2]`https://httpie.org/`

Listing 6.1: Fog Control Node Run Script

```bash
#!/usr/bin/env bash
# reasoner/test/{countT1}/{countT2}/{countT3}/{countT4}/{min}
date
http post http://192.168.1.105:8080/reasoner/test/2/0/3/0/5
   --timeout=1
sleep 119
date
http post http://192.168.1.105:8080/reasoner/test/2/0/8/0/2
   --timeout=1
sleep 119
date
http post http://192.168.1.105:8080/reasoner/test/2/0/13/0/5
      --timeout=1
sleep 119
date
http post http://192.168.1.105:8080/reasoner/test/2/0/8/0/1
   --timeout=1
sleep 119
date
http post http://192.168.1.105:8080/reasoner/test/2/0/3/0/1
   --timeout=1
```

Out of reasons of simplicity and relevance the deployment and stopping times were measured and mathematically rounded in order to use them in the figures. The measured and rounded deployment time per request is 2 seconds (rounded from 1.92 seconds), whereas the stopping time results in 0.5 seconds (rounded from 0.47 seconds). In the following graphs the red line depicts the input function, i.e., the requested services, and the blue line visualizes the number of deployed and running services. The service startup times are visible in the short delay between requesting the services and the visible rise of the function in the resulting diagram. The stopping time is visible at the end of Figure 6.10 where two applications finish at the same time.

Figure 6.10 shows the results of the experiment using the constant input function of Figure 6.9a drawn in red. In this case, all issued applications consist of two task requests of service type *t1* and eight of service type *t3*. The durations range from 1 to 5 minutes. At the beginning of the experiment, the *A4* application with 5 minutes duration, i.e., A4/5, is issued, followed by A4/2 at minute two, i.e., 2:00, A4/5 at 4:00, A4/1 at 6:00, and A4/1 at 08:00. The second line, drawn in blue, depicts the deployed services over time. At 0:00, A4/5 is requested and after the startup time of about 20 seconds, the 10 requested services are up and running. At 2:00, A4/2 is issued without extraordinary events. The first interesting point is at about 4:20. What happens there is that the input function requests A4/5 to start 10 more services, while a short period after the application request A4/2 is finished and shuts down, leaving 20 deployed services. At about 5:20, the first A4/5 is stopped as well. In minute 6:00, the first A4/1 starts and

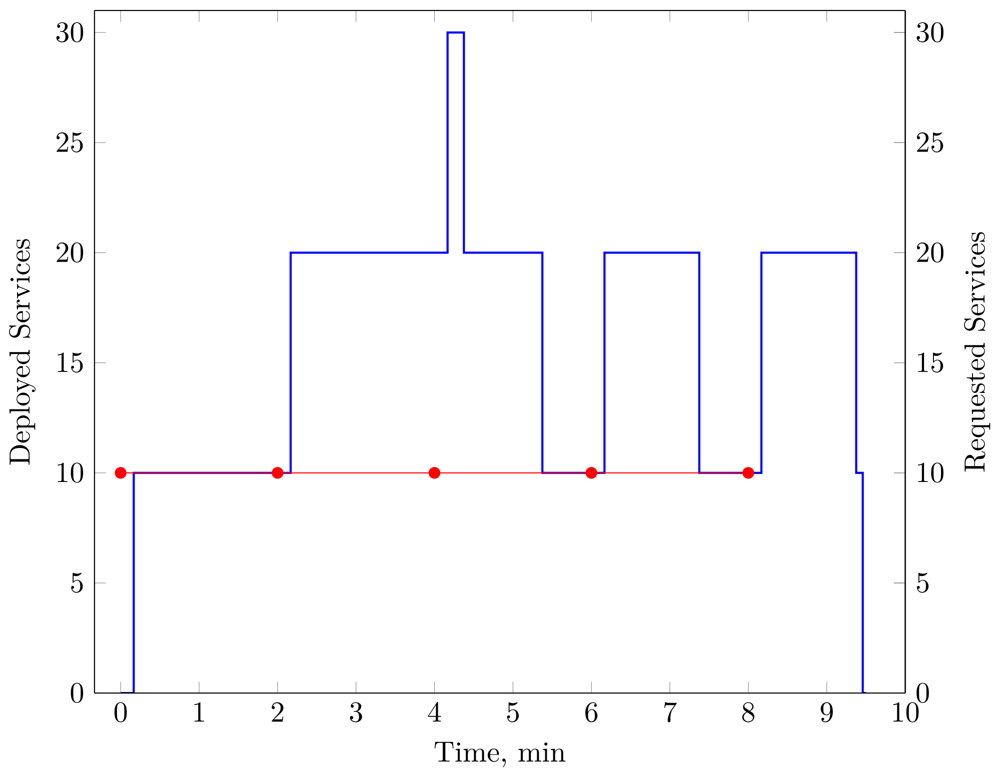


Figure 6.10: Service Deployment over Time with a Constant Input Function

finishes at around 7:20. The last request in minute 8:00 starts another A4/1 that finishes at around 09:20 at the same time as the second request of A4/5. The visible step at around 09:25 shows the stopping of A4/1 and A4/5.

Regarding the pyramid input function assessment presented in Figure 6.11, the experiment setup is the same as for the constant arrival scenario. The only difference is the varying number of task requests. The amount of task requests with service type *t1* remains at two per application, whereas the task request amount of service type *t3* fills up to the needed amount of task requests to achieve the total amounts of 5, 10, and 15 task requests per application.

Investigating the results of the experiment, the focus is placed on the essential points in this second experiment of this evaluation scenario. In the first two minutes A4/5 and A4/2 are requested and deployed without noteworthy events. Shortly after 4:00, the simultaneous starting of A4/5 and stopping of A4/2 results in a compensation phase.

The results of the last evaluation scenario present the dynamics of the framework and the possibility of handling several diverse applications with different durations and service types. Even simultaneous starting and stopping of services or whole applications does not affect the successful operation of the framework.

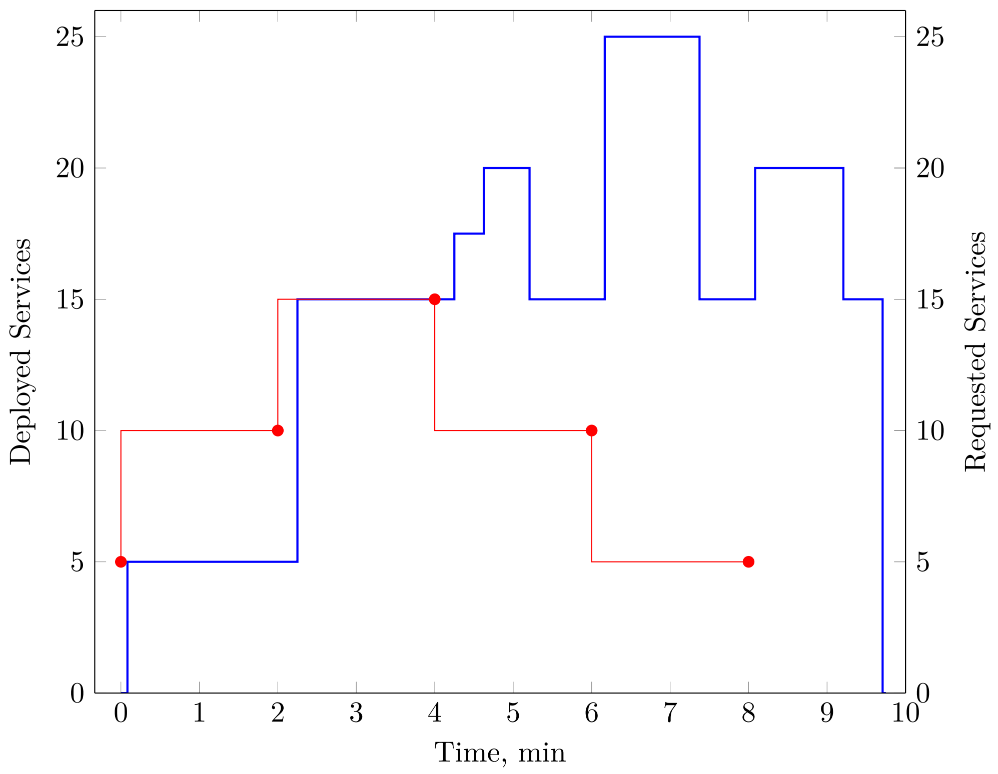


Figure 6.11: Service Deployment over Time with a Pyramid Input Function

In this chapter, the overall functionality of the developed fog computing framework was evaluated. The first three evaluation scenarios were aimed to evaluate the functionality of the framework and to analyze the essential functions and events the fog computing framework provides and handles. The other three evaluation scenarios, i.e., cost assessment, deployment time assessment, and the analysis of varying service deployment according to constant and pyramid input functions, presented the considerable benefits this framework accrues. To conclude, this chapter includes the answer for the last research question presented in Section 1.2, “How does the implemented fog computing framework improve the execution of IoT services compared to the execution in the cloud?”. The answer to this question is expressed by the obtained results of the corresponding experiments.

CHAPTER 7

Conclusion and Future Work

7.1 Conclusion

The IoT is an expanding technology trend that promises a substantial economic and scientific value for industry and academia in the upcoming years. A novel computing paradigm to support and unleash the full extent of the IoT is fog computing. Fog computing itself was introduced in the last couple of years, meaning, the corresponding research is still in its very beginning. Even though some theoretical aspects of fog computing have already been introduced, there is a lack of concrete implementation solutions that fulfill the full stack of volatile IoT requirements, e.g., topology dynamics and real-time service execution.

For this reason, in the course of this thesis, a comprehensive background and related work analysis was conducted before starting to design and implement a dynamic, extensible, and scalable real-world fog computing framework. The reviewed background technologies included cloud computing, the IoT, fog computing, virtualization technologies, and resource provisioning in cloud-based environments. The related work analysis revealed the fact that in the area of fog computing frameworks only few contributions have been made so far. To be more precise, this fact unfolds the necessity of further research in specific areas of fog computing frameworks and of supporting mechanisms for IoT service executions.

Consequently, the requirements of a potential fog computing framework were analyzed by investigating IoT use cases and two already existing IoT frameworks. The first investigated IoT framework is the work of Vögler et al. [63], who introduced a scalable large-scale IoT framework focusing on the use case of smart cities. The second work by Kim and Lee [38] presented an open source IoT framework that provides the means to develop and execute IoT services for all kinds of stakeholders, i.e., device and software developers, service providers, platform operators, and service users. With the ideas of these frameworks and the best practices of general distributed system designs [28], the architectural design of the dynamic, extensible, and scalable fog computing framework was constructed and technical decisions were made. During the design phase, the components of the framework were developed loosely coupled to facilitate the replaceability and extensibility of the framework. These non-functional requirements empower the researchers to further improve the developed framework.

Within the design phase not only the overall architecture of the framework but as well the technological design decisions have been made. A technology empowering the overall service deployment is Docker. Docker is a container virtualization technology used to deploy services in light-weight containers. The major benefits of Docker in comparison to common VMs is the omission of long VM start up times and the convenient storage and building of Docker Images used to deploy the containers. Another crucial technology decision is the Java Micro Framework to be used as microservice development basis, namely Spring Boot. Spring Boot is a slimmed Java framework making it possible to develop light-weight Java applications to be used as microservices. With this technology at hand, the huge Spring community, rich set of libraries, and third party integrations, there are endless possibilities for further implementation and improvements of developed components.

After constructing a holistic design, including identification of functional and technical requirements, the implementation phase took place. During this phase, the emphasis was put on light-weight technologies, loosely-coupled components, and on creating a stable and fault-tolerant distributed system. The extensible modules were implemented using Java interfaces, enabling a convenient substitution by implementing the specified interface methods. Stability was achieved by the use of stable and well-tested third party libraries making a thread-safe operation execution possible.

Lastly, the developed framework was evaluated according to six evaluation scenarios, including the analysis and assessment of the functionality and involved mechanisms, cost, deployment times, and the behavior of the fog computing framework in the dynamic volatile fog landscape. In more detail, in Scenario I the device failure mechanism was evaluated by running a predefined application and simulating a device failure by stopping the main application of the device. The result showed the successful migration of all services deployed on the failing device to suitable devices in the fog and cloud environment. Scenario II analyzed the device accedence event by running a similar application and adding a new fog cell to the system. As a result, all cloud services and several services running on fog devices were migrated to the newly acceded device. In Scenario III, a device overload was simulated and the overload policy was analyzed. The outcome was that the mechanism successfully and piece by piece migrated the services from the overloaded device until it was not overloaded anymore. Scenario IV assessed the service deployment cost and showed that with the fog computing framework 50 services can be deployed in the specified evaluation setup without the necessity of leasing costly cloud resources. Scenario V analyzed the service deployment times of 15 services both in the cloud and in the fog environment. The conducted values show a maximum deployment time of 31.20 seconds in the fog landscape and 267.05 seconds in the cloud environment. Thus, the fog deployment times are about nine times lower than the deployment times in the cloud. In the last scenario, Scenario VI, the behavior of the fog computing framework with a constant and pyramid input function was investigated. The conducted experiment shows the essential dynamics of the framework with the possibility to simultaneously start and stop several services and quickly react to the volatile service demand. As a result, the evaluation revealed the various benefits in the IoT service execution using the fog landscape.

In conclusion, this thesis provides a detailed functional and technical view on the fog computing framework and its components. The contribution of this work serves as a methodological basis and an evaluation toolset for further research in the area of fog computing.

7.2 Future Work

Since fog computing is a recent and emerging research field, there are many open research questions and promising research areas which are challenging for the future work. In this work, a prototype of the fog computing framework was designed, implemented and evaluated. Apart from the functionality of this prototype, there are lots of approaches, boundary conditions, special cases, and general improvements which require attention from the research community. The listing below presents some possible future improvements of the fog computing framework and gives an insight into according approaches to be applied.

Advanced Policies: This work provides a real-world fog computing framework and aims to satisfy the most important fog computing requirements, i.e, effective event processing in the case of device accedence, device failure, and overload. These events require sophisticated event handling policies. Since this work was focused on the development of the framework itself, specifically on the setup, communication, service deployment, and other mechanisms in the fog landscape, the implemented policies can be reconsidered and enhanced in future contributions. To be more specific, the *Resource Provisioning*, *Device Accedence*, *Device Failure*, and *Overload* policies can be extended or substituted by new policies. The architecture of the implemented prototype allows development and easy integration of new policies by using provided APIs.

Fault Tolerance Mechanisms: Another promising research challenge is to further improve fault tolerance and failure handling mechanisms. Specific ideas of improvement are the following: First, the creation of a *timeout handler* responsible for retrying requests several times before emitting a failure event. This would improve the fault tolerance in case a device is down for just a short period of time or is overloaded and can not answer in the defined connection timeout. Second, attention can be drawn to further pairing mechanism improvements. Currently, the pairing is performed in a way that at start-up of the main application, the device requests a responsible parent from the cloud-fog middleware. The device uses its fallback parent saved in its properties in case when the cloud-fog middleware is either not available, or does not respond, or the requested parent returned from the cloud-fog middleware does not respond. Though, assuming the parent fails during its operation, the child device is disconnected from the topology since it is not able to re-connect to another parent. Consequently, a re-pairing and additional parent searching mechanism in the fog landscape is another research challenge.

Fog Landscape Device Rearrangements: The device rearrangement in the fog landscape is another topic to be considered in the future work. After a specific amount of devices in the fog landscape fail, it is likely that the resulting new topology constellation is not optimal in terms of latency, bandwidth, network hops, location mapping, and connection preservation. Therefore, the rearrangement of fog devices to build a new effective device topology is a noteworthy aspect for future work. Such a rearrangement problem can be solved either locally within each fog colony, or globally, by providing the cloud-fog middleware with according optimization mechanisms.

Cloud Cost Consideration: The currently implemented cloud VM handling operates in a way that every time a VM does not contain any services, it is terminated. Due to the clearly defined cost of the leased VMs, the termination before a BTU is fully exploited leads to a waste of cloud resources and money. Thus, in future work, cost optimization methods have to be introduced into resource provisioning mechanisms.

Automated Device Discovery: In addition to the dynamic requesting of the responsible parent from the current cloud-fog middleware, the automated discovery of new devices, e.g., by scanning the network, is a promising research area. The automated device discovery would replace the manual pairing mechanism and improve the stability and fault tolerance of the fog computing framework.

List of Figures

List of Tables

List of Algorithms

Acronyms

AP Access Point. 70

API Application Programming Interface. 21

BTU Billing Time Unit. 82

DC Data Center. 7

DI Dependency Injection. 49

ETSI European Telecommunications Standards Institute. 19

HTTP Hypertext Transfer Protocol. 49

IaaS Infrastructure-as-a-Service. 10

IoT Internet of Things. 1

JAR Java Archive. 67

JPA Java Persistence API. 49

JSON JavaScript Object Notation. 16

LAN Local Area Network. 27

MCC Mobile Cloud Computing. 19

MEC Mobile Edge Computing. 19

MSA Micro Services Architecture. 16

NFC Near Field Communication. 8

NFV Network Function Virtualization. 26

P2P	Peer-to-Peer. 9
PaaS	Platform-as-a-Service. 10
QoS	Quality of Service. 12
RAN	Radio Access Network. 19
REST	Representational State Transfer. 16
RFID	Radio-Frequency Identification. 8
SaaS	Software-as-a-Service. 10
SDN	Software-Defined Networking. 26
SLA	Service Level Agreement. 12
UML	Unified Modelling Language. 41
URI	Uniform Resource Identifier. 50
VM	Virtual Machine. 10
VMM	Virtual Machine Monitor. 14
WAN	Wide Area Network. 27
WiFi	Wireless Connection Technologie. 8
WSN	Wireless Sensor Network. 9
XML	Extensible Markup Language. 16

Bibliography

[1] Mohammad Aazam and Eui-Nam Huh. Dynamic Resource Provisioning through Fog Micro Datacenter. In *Pervasive Computing and Communication Workshops*, PerCom'15, pages 105–110, Sydney, NSW, Australia, March 2015. IEEE.

[2] Mohammad Aazam and Eui-Nam Huh. Fog Computing Micro Datacenter Based Dynamic Resource Estimation and Pricing Model for IoT. In *29th IEEE International Conference on Advanced Information Networking and Applications*, AINA'15, pages 687–694. IEEE, March 2015.

[3] Mohammad Aazam, Marc St-Hilaire, Chung-Horng Lung, and Ioannis Lambadaris. MeFoRE: QoE based Resource Estimation at Fog to enhance QoS in IoT. In *23rd IEEE International Conference on Telecommunications*, ICT'16, pages 1–5, Thessalonica, Greece, May 2016. IEEE.

[4] Arif Ahmed and Ejaz Ahmed. A Survey on Mobile Edge Computing. In *10th IEEE International Conference on Intelligent Systems and Control*, ISCO'16, pages 1–8, Coimbatore, Tamilnadu, India, January 2016. IEEE.

[5] Shahid Ahmed. The Six Forces Driving the Internet of Things - Digital Revolution Summit, PricewaterhouseCoopers AG. `https://www.pwc.com/gx/en/technology/pdf/six-forces-driving-iot.pdf`, 2016. Accessed: 2016-09-29.

[6] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, April 2010.

[7] Kevin Ashton. That Internet of Things Thing. *RFiD Journal*, 22(7):97–114, June 2009.

[8] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A Survey. *Computer Networks*, 54(15):2787–2805, October 2010.

[9] Amazon Web Services (AWS). Fallstudien und Kundenerfolge mit der AWS Cloud - Warum Kunden sich fuer AWS entscheiden. `https://aws.amazon.com/de/solutions/case-studies/`, 2016. Accessed: 2016-09-29.

[10] Alessandro Bassi and Geir Horn. Internet of Things in 2020: A Roadmap for the Future. *European Commission: Information Society and Media*, 2008.

[11] Christian Baun, Marcel Kunze, and Thomas Ludwig. Servervirtualisierung. *Informatik-Spektrum*, 32(3):197–205, June 2009.

[12] Christian Baun, Marcel Kunze, Jens Nimis, and Stefan Tai. *Cloud Computing: Web-Based Dynamic IT Services.* Springer Publishing Company, Incorporated, 1st edition, 2011.

[13] Till Michael Beck, Sebastian Feld, Claudia Linnhoff-Popien, and Uwe Pützschler. Mobile Edge Computing. *Informatik-Spektrum*, 39(2):108–114, April 2016.

[14] Till Michael Beck, Martin Werner, Sebastian Feld, and Thomas Schimper. Mobile Edge Computing: A Taxonomy. In *6th International Conference on Advances in Future Internet*, AFIN'14, pages 48–54, Lisbon, Portugal, November 2014. International Academy, Research, and Industry Association.

[15] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. *Fog Computing: A Platform for Internet of Things and Analytics*, volume 546 of *Studies in Computational Intelligence*, pages 169–186. Springer International Publishing, Cham, Switzerland, March 2014.

[16] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog Computing and Its Role in the Internet of Things. In *1st ACM SIGCOMM Workshop on Mobile Cloud Computing*, SIGCOMM'12, pages 13–16, Helsinki, Finland, August 2012. ACM.

[17] Alessio Botta, Walter de Donato, Valerio Persico, and Antonio Pescape. Integration of Cloud Computing and Internet of Things: A Survey. *Future Generation Computer Systems*, 56:684 – 700, 2016.

[18] Joseph Bradley, Joel Barbier, and Doug Handler. Embracing the Internet of Everything to Capture your Share of $14.4 trillion. *White Paper, Cisco*, February 2013.

[19] Betsy Burton and David Willis. Gartner Mega Trends 2016. `https://www.gartner.com/doc/3407820`, 2016. Accessed: 2016-09-20.

[20] Sivadon Chaisiri, Bu-Sung Lee, and Dusit Niyato. Optimization of Resource Provisioning Cost in Cloud Computing. *IEEE Transactions on Services Computing*, 5(2):164–177, April 2012.

[21] Michael Chui, Markus Löffler, and Roger Roberts. The Internet of Things. *McKinsey Quarterly*, 2(2010):1–9, March 2010.

[22] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using Metrics to Evaluate Software System Maintainability. *Computer*, 27(8):44–49, August 1994.

[23] National Intelligence Council. Six Technologies with Potential Impacts on US Interests Out to 2025. *Disruptive Civil Technologies 2008*, April 2008.

[24] Amir Vahid Dastjerdi and Rajkumar Buyya. Fog Computing: Helping the Internet of Things Realize Its Potential. *Computer*, 49(8):112–116, August 2016.

[25] Amir Vahid Dastjerdi, Harshit Gupta, Rodrigo N. Calheiros, Soumya K. Ghosh, and Rajkumar Buyya. *Fog Computing: Principles, Architectures, and Applications*, chapter 4, pages 61–75. Elsevier, Burlington, MA, USA, January 2016.

[26] T. Hoang Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A Survey of Mobile Cloud Computing: Architecture, Applications, and Approaches. *Wireless Communications and Mobile Computing*, 13(18):1587–1611, February 2013.

[27] European Telecommunications Standards Institute (ETSI). Mobile Edge Computing - ETSI. `http://www.etsi.org/technologies-clusters/technologies/mobile-edge-computing`, 2016. Accessed: 2016-10-15.

[28] Mohamed E. Fayad and Douglas C. Schmidt. Lessons Learned Building Reusable OO Frameworks for Distributed Software. *Communications ACM*, 40(10):85–87, October 1997.

[29] Niroshinie Fernando, Seng W. Loke, and Wenny Rahayu. Mobile Cloud Computing. *Future Generation Computer Systems*, 29(1):84–106, January 2013.

[30] Martin Glinz. On Non-Functional Requirements. In *15th IEEE International Requirements Engineering Conference*, RE'07, pages 21–26, New Dehli, India, October 2007. IEEE.

[31] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions. *Future Generation Computer Systems*, 29(7):1645 – 1660, September 2013.

[32] Coda Hale, Yammer Inc., and Dropwizard Team. Dropwizard User Manual. `http://www.dropwizard.io/1.0.2/docs/manual`, 2016. Accessed: 2016-11-05.

[33] Bryan Hayes. Cloud computing. *Communications of the ACM*, 51(7), July 2008.

[34] Philipp Hoenisch, Stefan Schulte, Schahram Dustdar, and Srikumar Venugopal. Self-Adaptive Resource Allocation for Elastic Process Execution. In *6th IEEE International Conference on Cloud Computing*, CLOUD'13, pages 220–227, Washington, DC, USA, June 2013. IEEE.

[35] 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 *2nd ACM SIGCOMM Workshop on Mobile Cloud Computing*, SIGCOMM'13, pages 15–20, Hong Kong, China, August 2013. ACM.

[36] David S. Johnson. *Near-Optimal Bin Packing Algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[37] Evangelia Kalyvianaki. *Resource Provisioning for Virtualized Server Applications*. PhD thesis, University of Cambridge, 2009.

[38] Jaeho Kim and Jang-Won Lee. OpenIoT: An Open Service Framework for the Internet of Things. In *2014 IEEE World Forum on Internet of Things*, WF-IoT'14, pages 89–93, Seoul, Korea, March 2014. IEEE.

[39] John Leslie King. Centralized Versus Decentralized Computing: Organizational Considerations and Management Options. *ACM Computing Surveys*, 15(4):319–349, December 1983.

[40] Rob Kowalczyk. Introduction to Intel® Architecture - The Basics. *White Paper, Introduction to Intel*, 2014.

[41] Jay Lee, Behrad Bagheri, and Hung-An Kao. A Cyber-Physical Systems Architecture for Industry 4.0-Based Manufacturing Systems. *Manufacturing Letters*, 3(1):18–23, January 2015.

[42] Zongqing Lu, Jing Zhao, Yibo Wu, and Guohong Cao. Task Allocation for Mobile Cloud Computing in Heterogeneous Wireless Networks. In *24th IEEE International Conference on Computer Communication and Networks*, ICCCN'15, pages 1–9, Las Vegas, NV USA, August 2015. IEEE.

[43] Tom H. Luan, Longxiang Gao, Zhi Li, Yang Xiang, and Limin Sun. Fog Computing: Focusing on Mobile Users at the Edge. *Computing Research Repository*, abs/1502.01815(1), March 2015.

[44] Peter M. Mell and Timothy Grance. The NIST Definition of Cloud Computing. Technical report, National Institute of Standards and Technology, Gaithersburg, MD, USA, September 2011.

[45] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 1st edition, 2015.

[46] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful Web Services vs. 'Big' Web Services: Making the Right Architectural Decision. In *17th ACM International Conference on World Wide Web*, WWW'08, pages 805–814, New York, NY, USA, April 2008. ACM.

[47] Theodore S. Rappaport, Shu Sun, Rimma Mayzus, Hang Zhao, Yaniv Azar, Kevin Wang, George N. Wong, Jocelyn K. Schulz, Mathew Samimi, and Felix Gutierrez. Millimeter Wave Mobile Communications for 5G Cellular: It Will Work! *IEEE Access*, 1:335–349, May 2013.

[48] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, Inc., 2008.

[49] Dirk Riehle. *Framework Design.* PhD thesis, Technische Wissenschaften ETH Zürich, 2000.

[50] Rodrigo Roman, Javier Lopez, and Masahiro Mambo. Mobile Edge Computing, Fog et al.: A Survey and Analysis of Security Threats and Challenges. *Computing Research Repository*, abs/1602.00484, November 2016.

[51] Franz Rothlauf. *Design of Modern Heuristics: Principles and Application.* Springer Publishing Company, Incorporated, 1st edition, 2011.

[52] Zohreh Sanaei, Saeid Abolfazli, Abdullah Gani, and Rajkumar Buyya. Heterogeneity in Mobile Cloud Computing: Taxonomy and Open Challenges. *IEEE Communications Surveys and Tutorials*, 16(1):369–392, January 2014.

[53] Subhadeep Sarkar and Sudip Misra. Theoretical Modelling of Fog Computing: a green Computing Paradigm to Support IoT Applications. *IET Networks*, 5(2):23–29, March 2016.

[54] Enrique Saurez, Kirak Hong, Dave Lillethun, Umakishore Ramachandran, and Beate Ottenwälder. Incremental Deployment and Migration of Geo-distributed Situation Awareness Applications in the Fog. In *10th ACM International Conference on Distributed and Event-based Systems*, DEBS'16, pages 258–269, Irvine, CA, USA, June 2016. ACM.

[55] Sukhpal Singh and Inderveer Chana. QoS-Aware Autonomic Resource Management in Cloud Computing: A Systematic Review. *ACM Computing Surveys*, 48(3):42:1–42:46, December 2015.

[56] Olena Skarlat, Stefan Schulte, Michael Borkowski, and Philipp Leitner. Resource Provisioning for IoT Services in the Fog. In *9th IEEE International Conference on Service Oriented Computing and Applications*, SOCA'16, pages 32–39, Macau, China, November 2016. IEEE.

[57] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors. *SIGOPS Operating Systems Review*, 41(3):275–287, March 2007.

[58] Victor Sower, Kenneth Green, Pamela Zelbst, and Morgan Thomas. U.S. Manufacturers Report Greater RFID Usage - RFID Journal. `http://www.rfidjournal.com/articles/view?9589/3`, 2016. Accessed: 2016-10-02.

[59] Krishna Srinivasan. Dropwizard vs. Spring Boot: A Comparison by JavaBeat. `http://javabeat.net/spring-boot-vs-dropwizard/`, April 2016. Accessed: 2016-11-05.

[60] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *General Track: 2001 USENIX Annual Technical Conference*, USENIX'01, pages 1–14, Berkeley, CA, USA, June 2001. USENIX Association.

[61] 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, October 2014.

[62] Ovidiu Vermesan and Peter Friess. *Internet of Things: From Research and Innovation to Market Deployment.* River Publishers Aalborg, 2014.

[63] Michael Vögler, Johannes M. Schleicher, Christian Inzinger, and Schahram Dustdar. A Scalable Framework for Provisioning Large-Scale IoT Deployments. *ACM Transactions on Internet Technology*, 16(2):11:1–11:20, March 2016.

[64] Mike J. Walker, Betsy Burton, and Michele Cantara. Gartner Hype Cycle 2016. `https://www.gartner.com/doc/3383817`, 2016. Accessed: 2016-09-20.

[65] Phillip Webb, Dave Syer, Josh Long, Stephane Nicoll, Rob Winch, Andy Wilkinson, Marcel Overdijk, Christian Dupuis, and Sebastien Deleuze. Spring Boot Reference Guide. `http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/`, 2016. Accessed: 2016-11-05.

[66] Aaron Weiss. Computing in the Clouds. *netWorker - Cloud computing*, 11(4):16–25, December 2007.

[67] Evan Welbourne, Leilani Battle, Garret Cole, Kyla Gould, Kyle Rector, Samuel Raymer, Magdalena Balazinska, and Gaetano Borriello. Building the Internet of Things Using RFID: The RFID Ecosystem Experience. *IEEE Internet Computing*, 13(3):48–55, May 2009.

[68] Charles P. Wright and Erez Zadok. UnionFS: Bringing Filesystems Together. *Linux Journal*, 2004(128):1–8, December 2004.

[69] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. Fog Computing: Platform and Applications. In *Hot Topics in Web Systems and Technologies, 2015 3rd IEEE Workshop*, HotWeb'15, pages 73–78, Washington, DC, USA, November 2015. IEEE.

[70] Shanhe Yi, Cheng Li, and Qun Li. A Survey of Fog Computing: Concepts, Applications and Issues. In *2015 Workshop on Mobile Big Data*, Mobidata'15, pages 37–42, Hangzhou, China, June 2015. ACM.

[71] Zhi-Hui Zhan, Xiao-Fang Liu, Yue-Jiao Gong, Jun Zhang, Henry Shu-Hung Chung, and Yun Li. Cloud Computing Resource Scheduling and a Survey of Its Evolutionary Approaches. *ACM Computing Surveys*, 47(4):63:1–63:33, July 2015.

[72] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud Computing: State-Of-the-Art and Research Challenges. *Journal of Internet Services and Applications*, 1(1):7–18, April 2010.

[73] Jiang Zhu, Douglas S. Chan, Mythili S. Prabhu, Preethi Natarajan, Hao Hu, and Flavio Bonomi. Improving Web Sites Performance Using Edge Servers in Fog Computing Architecture. In *2013 IEEE 7th International Symposium on*, SOSE'13, pages 320–323. IEEE, March 2013.