Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Suitability of Serverless Computing Approaches

Christopher Völker

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Michael Wurster, M.Sc. Dipl.-Inf. Klaus Brenner
Commenced:	October 4, 2017
Completed:	April 4, 2018

Kurzfassung

Die zunehmende Bedeutung von Cloud Computing geht einher mit dem Bedarf eines leistungsabhängigen Workload-Managment von Services. Services müssen in der Lage sein in Abhängigkeit der aktuellen Arbeitslast zu skalieren und sollten einfach zu ersetzen sein ohne andere Services einer Anwendung zu beeinflussen. Beispielsweise muss eine monolithische Anwendung als ganzes deployed werden um mehr Resourcen für einen einzelnen Service bereit zu stellen. Zusätzlich könnten fehlerhafte Services die gesamte Anwendung beeinflussen und daher auch die Verfügbarkeit der anderen Services beeinträchtigen. Dies entspringt der Tatsache, dass Komponenten in solch einer monolithischen Anwendung stark voneinander abhängen weshalb sie sich nur schwer skalieren lassen und nicht flexibel sind. Folglich sollten die Komponenten einer Anwendung so unabhängig wie möglich voneinander sein um einfaches Skalieren zu ermöglichen und fehlertoleranter zu sein. Daher sind Monolithen nicht in der Lage diesen Anforderungen gerecht zu werden. Dieser Umstand bringt den Bedarf einer flexibleren Architektur auf. Diese Probleme werden von der Microservice Architektur addressiert. Microservices streben eine unabhängigere flexiblere Anwendungsarchitektur an indem Services so klein und unabhängig wie möglich gehalten werden. Dieser Ansatz ist von essentieller Bedeutung für Serverless Computing. Serverless Computing ist ein Teil von Cloud Computing und basiert auf der Microservice Architektur. Der Schwerpunkt liegt hierbei auf dem Bereitstellen von resourcen auf eine fein granulare und flexible Art und Weise um die Vorteile von Cloud Computing besser ausnutzen zu können. Um die Eignung des Serverless Computing Ansatzes zu evaluieren wird eine Fallstudie durchgeführt. Diese Fallstudie ist dafür ausgelegt Unterschiede der Kosten, Leistungsfähigkeit und Antwortzeiten zwischen Cloud Anwendungen und Serverless Anwendungen zu untersuchen. Sie zielt darauf ab, mögliche Anwendungsfälle für Serverless Computing zu erkennen. Die Ergebnisse dieser Fallstudie werden zusammengefasst und könnten als Basis für weitere Untersuchungen von Serverless Computing Ansätzen dienen.

Abstract

The increasing importance of cloud computing is accompanied by a need for performance sensitive workload management of software services. Services are required to scale to the current workload and it should be easy to replace them without affecting other services within an application. Monolithic applications, for instance, require deploying the whole application in order to provide more resources for a single service. In addition, failing services may influence the whole application and might therefore affect the availability of other services adversely. This is due to the dependence of components within such a monolithic application which prevents it from efficient scaling and flexibility. As a result the components of an application should be as independent as possible in order to be able to scale and provide more failure tolerance. Therefore, monoliths are not able to fulfill these requirements. This fact raises the need for a more flexible architecture. These problems are addressed by the microservice architecture. Microservices aim at a more independent and flexible application architecture by keeping services as small and independent as possible. This approach is fundamental to serverless computing which is described in this thesis. Serverless computing is part of the cloud computing paradigm and is based on the microservice architecture. It focuses on providing resources in a fine granular and flexible manner in order to further utilize the benefits of cloud computing. In order to evaluate the suitability of the serverless approach, a case study is conducted. This case study is designed to evaluate cost, performance and response time differences between applications deployed in the cloud and those realized as serverless applications. It aims at identifying suitable use cases for serverless computing approaches. The findings of this case study are summarized and may serve as basis for further research on serverless computing approaches.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

API Application Programming Interface. 22

AWS Amazon Web Services. vii

CLI Command Line Interface. 19

DDoS attack Distributed Denial of Service attack. 17

EC2 Elastic Compute Cloud. vii

ELB Elastic Load Balancer. 11

FaaS Function as a Service. 1

HTTP HyperText Transfer Protocol. 7

IaaS Infrastructure as a Service. 12

IAM Identity and Access Management. 37

JSON Javascript Object Notation. 1

NIST National Institute of Standards and Technology. 11

OS Operating System. 1

PaaS Platform as a Service. 13

RDS Relational Database Service. 41

REST Representational State Transfer. 7

S3 Simple Storage Service. 21

SaaS Software as a Service. 14

SES Simple Email Service. 42

SOA Service Oriented Architecture. 7

VM Virtual Machine. 1

1 Introduction

In the last decades accessing content over the internet has become more and more important for customers. Computing as a service has evolved to be a way of accessing content of any kind over the web. Using services over the wire also enables assembling applications by using different services [PTDL07]. Customers have come to rely on such web services which help them to do all kind of things and take them for granted. In 2008 using services in this fashion was already a vision and referred to as "computing as the 5th utility" or "utility computing" [BYV+08][AFG+09]. This has become reality for quite some time now and services nowadays are used in an on demand manner from anywhere the user wants to. Consequently, these services experience an increasing amount of users and there are many offers of similar services by different providers.
The increasing popularity of such services raises the need for ensuring high availability by providing enough resources in order to be able to satisfy all user requests. Of course the application's availability strongly depends on the server it is hosted on which raises the need for constant and high availability of such a server [PTDL07]. In order to achieve high availability applications are deployed on multiple different servers in a redundant manner to avoid single points of failures.
Providing redundancy of servers is costly since machines have to be acquired, maintained and running whether they are used or not. Of course providers are trying to minimize their operational costs in order to be as profitable as possible [ZW+13]. This raises the question of how to efficiently utilize servers since running one application on a single server is not very efficient. Various amounts of workload result in overloading or idle running of servers. This might result in paying for unused resources or even loosing money since the server is not able to handle all customer requests. As a result, there is the need of scalable applications and infrastructure which has been one of the crucial aspects for businesses providing web services [GV06]. But scaling physical servers is a time-consuming process since the corresponding hardware has to be acquired, set up and integrated into the infrastructure.
For many years, applications have been deployed on Virtual Machines (VMs) which makes for a more efficient use of servers [BYV+08]. VMs are part of the virtualization method which allows running several operating systems on the same machine independently. This allows for better utilization of a single server by running multiple applications in different VMs [RR16, page 24]. Furthermore, applications deployed in different VMs are isolated from each other, thus, contributing to the availability of the server since applications cannot affect other ones in different VMs [Ley09]. Storing the state of virtual machines in an image file, for instance using the open virtualization format (OVF), enables a quick start without manual configuration. This image format is interoperable, portable, efficient and extensive facilitating the distribution of virtual appliances [RR16, page 186-187]. Consequently, they

Figure 1.1: Relative interest for "serverless computing" in 2016 and 2017 (source: Google Trends)

can be instantiated in an on-demand manner on multiple servers while allowing them to be started and stopped in short time. Therefore, resources can be added or removed dynamically depending on the current workload. Of course, it is more efficient to add resources on-demand as well as to remove them if they are not needed anymore.
Virtual machines are a core aspect of cloud computing. A Cloud Provider provides resources in an on-demand manner by enabling the customer to add and remove resources in this manner and only bill them for what they are actually using [CLN12]. As a result enterprises do not have to operate hardware for resource provisioning themselves but can dynamically allocate computing resources from a cloud provider. Thus, operating physical hardware is sourced out to the cloud provider. Consequently, more costs can be saved since the need for employees for operating hardware decreases [AFG10].
The paradigm of cloud computing has become increasingly popular. It provides on-demand network access to a shared pool of resources which can be rapidly and dynamically provisioned and released [MG11]. This results in decreasing cost for operating web applications as well as increasing flexibility [CLN12]. This enables putting more effort in developing and deploying applications. However, large applications containing all the business logic are rather difficult to develop, maintain and scale. This is due to the fact that such monolithic applications become more complex with the amount of different services. Furthermore, such a design prevents efficient scaling because only the application as a whole can be multiplied, while scaling particular parts of the application is not possible [VGC+15]. As a result, splitting the monolithic application into small services reduces complexity and ensures maintainability [GV06]. Creating small and particularly independent services might increase an application's scalability and is part of a microservice architecture.
The microservice architecture offers a flexible and scalable approach for designing web applications. It aims at a far more independent architecture compared to monolithic approaches and results in loose coupling [Fow14b]. Such loosely coupled services can be maintained and deployed without influencing the availability of other services. They should provide interfaces and languages (e.g. XML, Javascript Object Notation (JSON)) increasing their reusability and enabling communication through such interfaces [PTDL07]. Deploying a monolithic application in order to update one particular service results in all services of this application becoming unavailable. These effects on external services do not appear with loose coupling because each service can be independently maintained.

Additionally, this allows for better utilization of the benefits of cloud computing.
In order to use an application, at least one instance has to be available even if it is not actively used at the moment because users have to be able to query an application at any time. Without any instances or any other kind of entry points available, users obviously cannot use the application. This leads to the idea of Function as a Service (FaaS) which is an event-driven approach of cloud computing [MB17]. It enables invoking a specific function upon a certain event or after a certain action. FaaS is a stateless approach which allows executing code using a predetermined runtime [WS16]. It is built on the microservice architecture and aims at realizing short living functions. Additionally, it is designed to be fault-tolerant and auto-scaling, whereas these aspects are ensured by the provider rather than the developer [BCC+17]. In order to reduce resource consumption, allocated resources required for function execution are released upon its termination. As a result, there are no resources allocated which the customer is billed for unless a function is invoked. Events triggering a function result in providing the necessary resources, executing the function and free the resources upon returning its result. This allows a more granular pay per use model and the customer is not billed for idle time of functions. This is known as serverless computing and aims at further reducing operational costs while increasing flexibility.
In contrast to VMs, functions are executed in dedicated and isolated containers. Containers do not contain a Guest Operating System (OS) but use the host OS reducing the required amount of resource and storage for starting up and being executed. They only provide the required runtime for executing the application within the container. For this reason, boot-time, time of generating and distributing container images are short and faster than VMs [SHM+14]. This enables a more efficient provisioning of resources and especially helps at better utilizing the resources of the underlying infrastructure. Serverless computing makes use of these aspects by separating application logic into small functions and executing them in containers. But this separation increases the complexity of applications due to a high amount of functions. However, reduced operational costs, fault-tolerance, auto-scaling and especially high flexibility outweigh this complexity issue. Serverless computing is becoming more and more popular and seems to be on a trajectory of increased growth and adoption [Low16]. This is also suggested by the increasing interest for “serverless computing” on Google Trends (see figure 1.1). The interest is hereby calculated in relation to the highest value within the given time period. For instance, an interest of 50 indicates half the interest at that time compared to the highest point in the diagram. This trend shows the increasing interest in serverless computing within the last two years.
Serverless computing is still under active development and there are multiple project aiming at the extension of existing serverless platforms [MB17]. At this time, there are many ideas on how to make use of the serverless computing approach. However, appropriate patterns for using serverless computing are still emerging as it might not fit for every requirement [Tho17].

1.1 Scope of work

In this thesis the term “serverless computing” is introduced and classified into the service model definitions of cloud computing. A definition and discussion of benefits and drawbacks of this approach are described as well. Additionally, providers of serverless offerings are introduced and compared to each other. The main aspects this comparison focuses on are billing models, supported programming languages and provider native monitoring options. This also includes comparing provider specific approaches to each other which might interfere with hybrid cloud deployment. Furthermore a case study is conducted in order to compare the differences between deploying applications on instances and realizing them as serverless functions. Finally, the results of the case study are evaluated with respect to cost, scalability and performance differences. The case study as well as its evaluation aim at disclosing scenarios which the serverless computing approach can or cannot be used for. Of course, this includes elaborating specific aspects which should be considered prior to making decisions about adopting the serverless computing approach.

1.2 Outline

The remainder of this thesis is structured as follows:

Chapter 2 – Fundamentals and Related Work: This chapter describes the fundamental concepts which led to the idea of serverless computing and presents other work which is related to these concepts. It also describes the serverless computing paradigm and illustrates the main aspects of serverless computing stating different characteristics for cross provider comparison. This also includes a brief description of issues these concepts were designed to overcome as well as issues they may present. Additionally, management possibilities such as debugging, monitoring and so on are discussed as well.

Chapter 3 – Research Methodology: The next chapter is about the research approach and introduces serverless providers as well as serverless frameworks. It aims at giving a rough overview of available providers and the differences between their offers. Furthermore, some frameworks supporting the development and deployment of serverless applications are described. This chapter represents the prerequisite for the case study and, therefore, also introduces some test and monitoring components for serverless computing.

Chapter 4 – Case study: This chapter describes the setup of the case study which includes the design of the test application, its deployment and the test and monitoring components. The study is designed to compare the cost, performance and scalability of serverless and monolithic approaches. Furthermore, it summarizes the results and evaluates them.

Chapter 5 – Summary and Future Work The last chapter provides a summary of the findings of this thesis. In addition, it gives an outlook to future work in the area of serverless computing.

2 Fundamentals and Related Work

Serverless computing is built on different paradigms and is designed to overcome several issues with respect to operational aspects. For example, monolithic application and the usage of plain servers running single applications presented issues in scalability and maintainability in today's requirements for web applications. This led to a different architectural approach of providing infrastructure and developing applications. Whereas cloud computing aims at better utilization of the underlying infrastructure, microservices are designed as maintainable and easy to understand services. These two concepts raised the idea of serverless computing aiming at further reduction of operational costs while increasing flexibility and scalability. This chapter introduces these fundamental concepts and presents other work which is related to them.

2.1 Monolithic Applications

Monolithic applications lack the ability of dynamic scaling or efficiently adding new features [GV06]. Monolithic applications usually consist of many different functionalities which many developers in different teams are working on [VGC+15]. One failing service in such an application might affect the whole application. Furthermore the possibly increasing amount of services within such an application increases it's complexity. This inherent complexity makes it more difficult to perform changes, thus, becoming more time-consuming to maintain and extend [VGC+15]. Rapidly providing updates and new features to an application as well as scaling is nowadays a crucial requirement for competitiveness. But the monolithic design impedes these aspects due to it's complexity and the requirement of deploying the whole application for scaling single services. Figure 2.1 shows the monolithic architecture and it's scaling. These issues are addressed by the microservice architecture which aims for flexible, easy to maintain applications whereas each service of the application can be managed independently.

2.2 Microservices Architecture

The microservice architecture addresses some issues raised by monolithic applications and strives for simple and easy to understand applications. It aims at distributing functionalities into small services which can be developed, tested and deployed independently [**2015_MicroserviceComparisonVillamizar**]. Compared to a monolithic application functionalities are separated into independent small services. Hereby, the services are built

around business capabilities whereas they take a broad-stack implementation of software for that business area, including user-interface, persistent storage, and any external collaborations [Fow14a]. This facilitates deploying and scaling each service independently as illustrated in figure 2.1.
While monolithic applications are built as single, autonomous units, microservices should be designed for a very small amount of tasks (a specific business capability). However, they still might depend on data of other services. Therefore they need to be able to communicate with other services in a lightweight manner [BHJ15]. In order to maintain their independence, microservices should establish this kind of communication as neutral as possible with respect to the data model. Usually the most popular architectural style for service communication - called Representational State Transfer (REST) - is used to establish such a neutral communication [VGC+15]. The Service Oriented Architecture (SOA) may also use vendor specific messaging for communication which microservices are not designed to do [XWQ16].
Within microservices the governance is decentralized, whereas it is centralized in SOA [Fow14a][XWQ16]. This decentralized approach enables to independently deploy single functionalities without impacting other ones. Furthermore, it allows implementing each service in a different programming language or with a different kind of data storage [BHJ15]. The choice of which language or technology to use is made based on informed decisions which allows to realize services in an environment fitting them best. This facilitates making decisions, based on the environment the service is supposed to be operated since the decision can be made for each particular business capability. For monolithic applications on the other hand, there is only one decision to be made which has to consider all capabilities of the application. Since they find themselves in the same environment and therefore must comply with its restrictions which might be difficult for some services of within that monolithic application. This is called decentralized governance
Another requirement of microservices is their design for failures. Services might become unavailable and other services relying on them must respond to that gracefully [Fow14a]. This is rather complex to handle due to their distributivity which represents a disadvantage to the monolithic design.

In order to compare the cost of the microservice approach to the monolithic one, Villamizar et al conducted a case study which is described below. The application within this case study is deployed on AWS EC2 and on AWS Lambda. AWS EC2 is an offering of Amazon which enables the customer to provision VMs. The customer is hereby in complete control of the configuration of these VMs as they behave like a single server “owned” by the customer. AWS Lambda on the other hand is a more restrictive service. In allows the customer to deploy code which will be run by the service upon invocation. Lambda functions can be invoked by different events such as a HyperText Transfer Protocol (HTTP) request. In contrast to EC2 the customer is not able to configure runtime environments or other requirements. However, the customer does not have to worry about providing enough resources since this is taken care of by AWS Lambda.

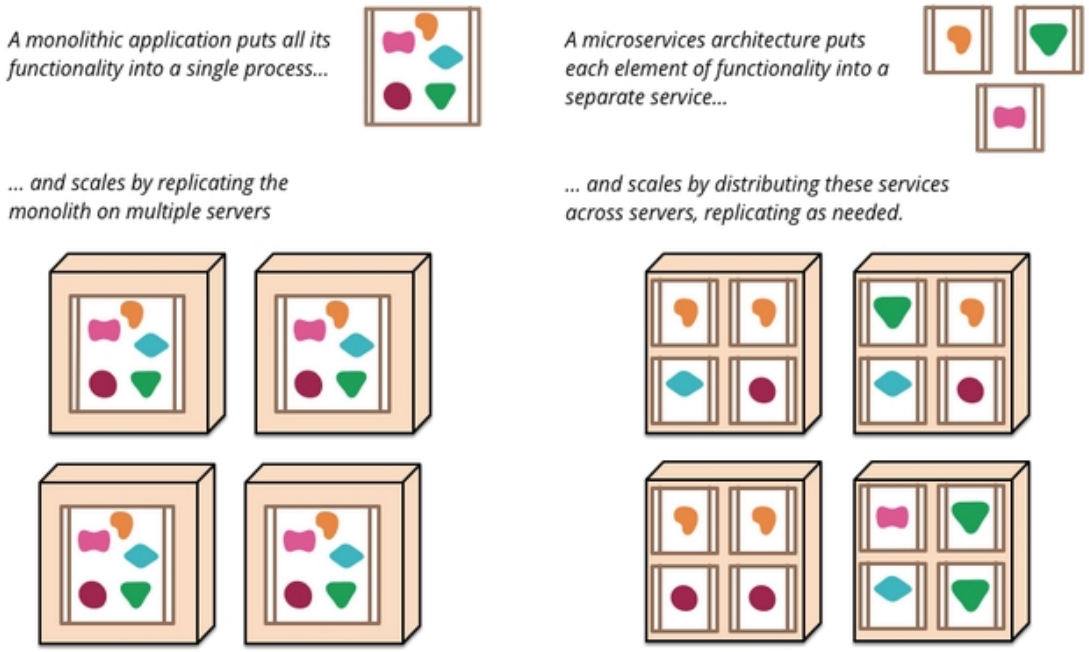

Figure 2.1: Monoliths and Microservices (source: [Fow14a])

2.2.1 Infrastructure Cost Comparison

In [VGO+16] Villamizar et al conducted a case study comparing the costs and response times of monoliths with those of microservices [VGO+16]. This case study is based on an application consisting of two services. The first service implements a CPU intensive algorithm. The second service is implemented using a separate data store and returning data from it. For the monolithic application which contains both services the usual response times were 3000 milliseconds for service 1 (S1) and 300 milliseconds for service 2 (S2). The general set-up of the different approaches is shown in table 2.1 [VGO+16]. In order to being able to receive comparable results all three approaches shown in table 2.1 where deployed on AWS.
The monolithic approach as well as the microservice approach of this case study were deployed on AWS EC2. The microservice was additionally deployed on AWS Lambda. Of

Application	Developed (using)	Deployed on
Monolith	Play-Framework	AWS EC2
Microservice	Play-Framework	AWS EC2
AWS Lambda	Node.js	AWS Lambda

Table 2.1: Technical details of the different approaches

course in the microservice approach, both services (S1 and S2) are developed independently and therefore can be deployed and operated separately, whether they are deployed in the cloud or on AWS Lambda does not matter.

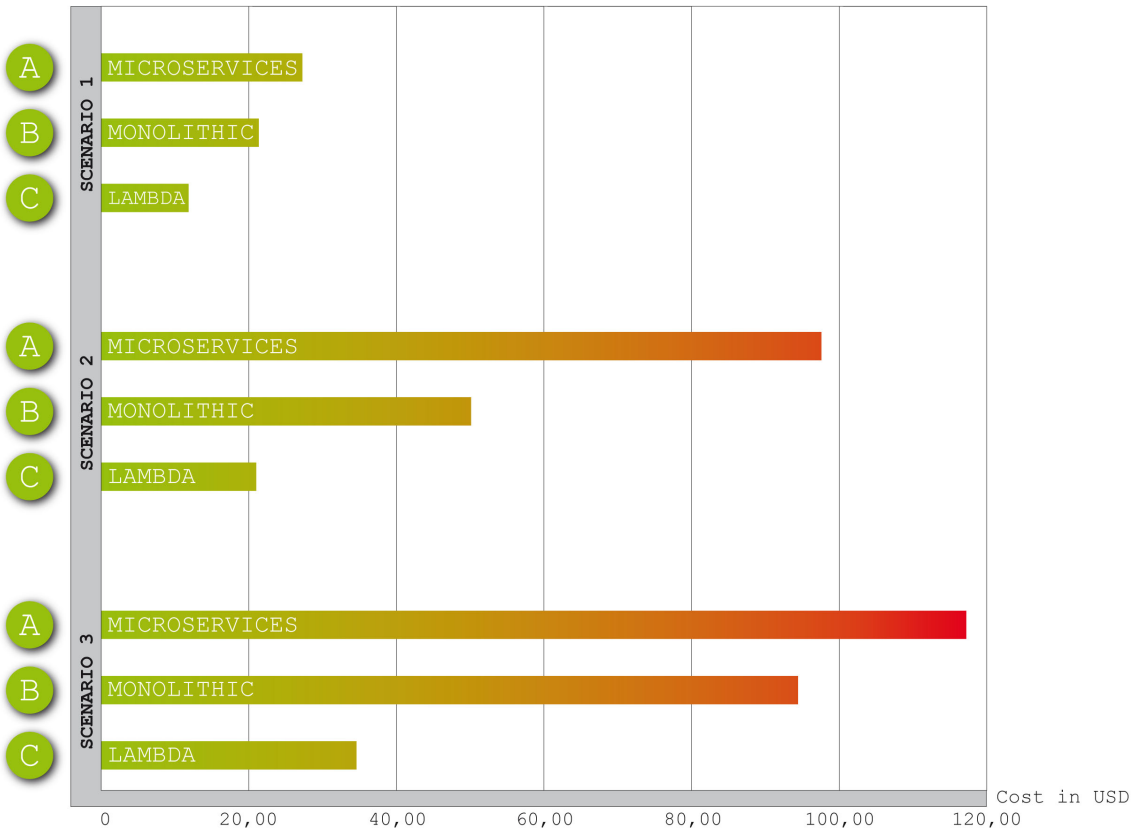

Figure 2.2: Cost comparison of the three architectures per million of requests; (adapted from [VGO+16])

There were three scenarios for which the results were determined. They only refer to the percentage of requests the corresponding service needed to handle. In the first scenario the service S1 had a workload of 20% while S1 had 80% where as in the third scenario it was the other way around. The second scenario had an equal workload on both services. However, this case study is not designed to determine the differences in scalability with respect to changing workload. They rather focused on comparing response times and costs during workload peaks. Therefore, they tested the amount of users the monolithic application deployed on AWS EC2 could handle and stress tested the other architectures with the same number of users. Consequently, this case study aimed at determining the required resources and cost needed for the microservice to match the amount of requests of the monolithic application. The case study in this thesis, however, focuses on the differences with respect to changing workloads while also leveraging scaling capabilities of AWS EC2.

Costs The resulting costs for the three test architectures are shown in figure 2.2. This figure shows that the microservices architecture is able to reduce the operating cost by 9% to 13% [VGO+16]. Even more reduction is achieved using AWS Lambda which yields

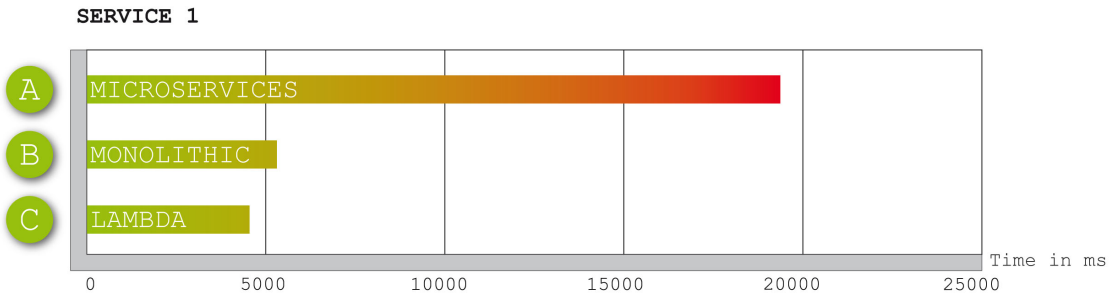

Figure 2.3: Average response time for S1 during peak periods; (adapted from [VGO+16])

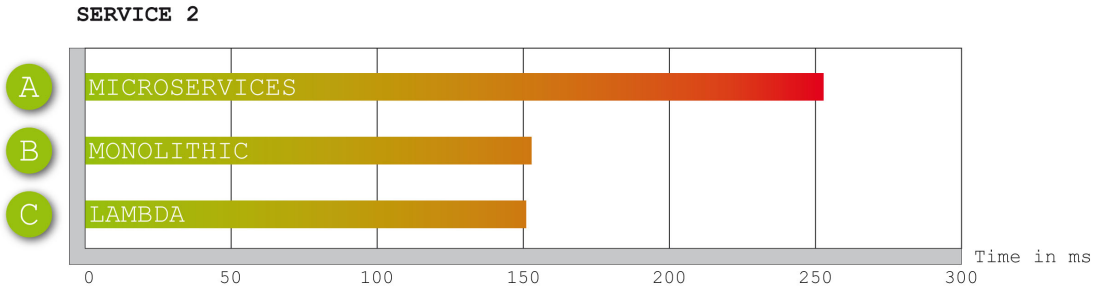

Figure 2.4: Average response time for S2 during peak periods; (adapted from [VGO+16])

between 51% and 70% depending on which architecture it is compared to [VGO+16]. This improvement applies to all three scenarios.

Response time For reasons of simplicity it is not distinguished between the three scenarios in this case since there was almost no difference concerning the response time between the scenarios. As shown in figure 2.3 (S1) and 2.4 (S2), the response times of the three architectures were similar except for the microservices architecture which was deployed in the cloud. This is due to the fact that all requests for both services first have to pass the same gateway which distributes the request to the corresponding service [VGO+16]. As a result, this gateway is becoming the bottleneck because it is not built for distributing high amounts of requests. Because the gateway requires additional time for handling and passing requests to each service, the response times for the microservice approach were higher. The monolithic application circumvents this by using an Elastic Load Balancer (ELB) which is designed to distribute high amounts of requests with minimum delay and scales according to the workload. Since AWS Lambda has separate components for invoking each service, it was not affected by this problem. Although the response times are worse for the microservice architecture in the cloud, the advantages of cost reduction and agility in independently deploying the services makes it a more beneficial approach than the monolithic one. The AWS Lambda architecture shows even more benefits since it greatly reduces the costs while keeping the response times rather stable.

2.3 Cloud Computing

The paradigm of cloud computing is a model for providing on-demand network access to a shared pool of resources [MG11]. Such resources may be storage, computing or networking capacity among others. The main aspect of this model is the ability to rapidly provide and release resources in an on-demand manner while keeping the interaction with the service provider to a minimum. The National Institute of Standards and Technology (NIST) has defined essential characteristics, services models and deployment models for enabling broad comparisons of cloud services [MG11].

2.3.1 Essential Characteristics

The NIST has defined five characteristic which are essential for cloud computing offerings. The first characteristic is the provisioning of computing capabilities without human interaction with each service provider [MG11]. Furthermore they are available over the network and can be accessed by many different kinds of devices [MG11]. Provided resources are pooled to serve multiple customers using a multi-tenant model whereas resources are dynamically assigned and reassigned according to consumer demand [MG11]. This also includes rapid elasticity allowing to scale rapidly outward and inward by enabling the customer to elastically provision and release resources [MG11]. These resources are charged based on the actual use, whereas cloud systems providing these resources automatically control and optimize them by leveraging a metering capability at some level of abstraction appropriate to the type of service [MG11]. In summary the five essential characteristics for cloud computing consist of an “On-demand self-service”, “Broad network access”, “Resource pooling”, “Rapid elasticity” and “Measured service”.

2.3.2 Service Models

There are different service models of cloud offerings each of which provides different capabilities for the customer. Each of these service models shifts some responsibilities for providing web applications between provider and customer (See figure 2.5). The three main models are briefly described below.

Infrastructure as a Service (IaaS) The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications [MG11].

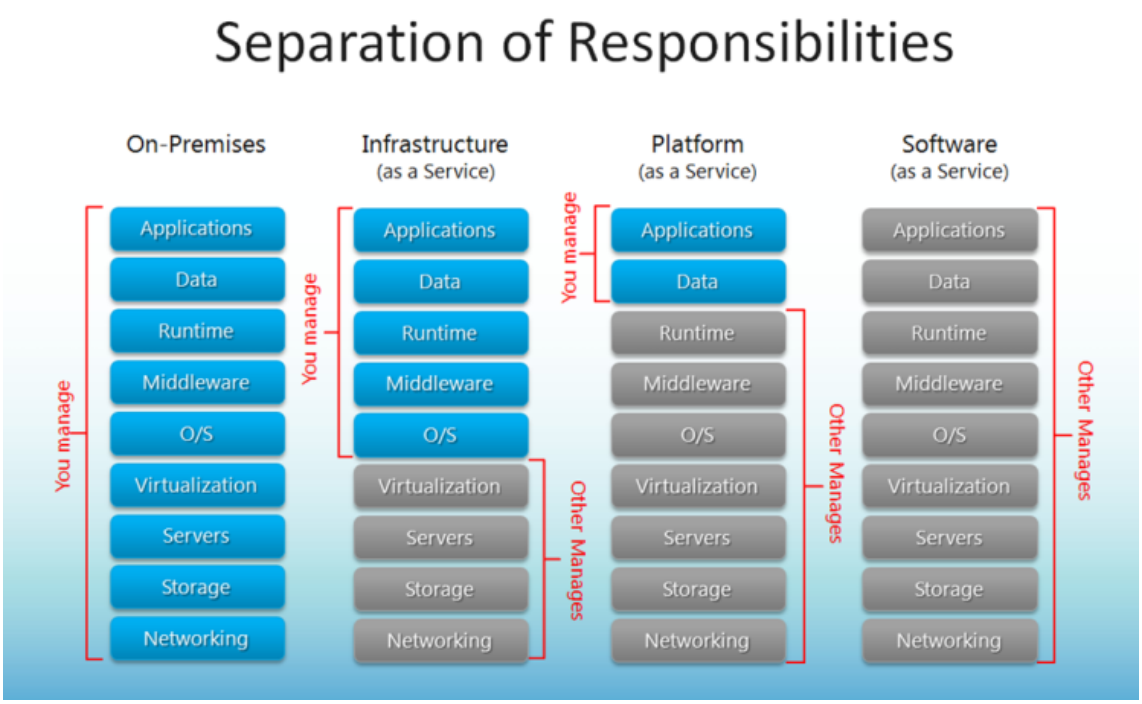

Figure 2.5: Separation of Responsibilities[1]

Platform as a Service (PaaS) The consumer is able to deploy onto the cloud applications created using programming languages, libraries, services, and tools supported by the provider[2]. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment [MG11].

[1] http://robertgrainer.com/uploads/images/2014/AzureServiceOverview.png

[2] This capability does not necessarily preclude the use of compatible programming languages, libraries, services, and tools from other sources.

Software as a Service (SaaS) The capability provided to the consumer is to use the provider's applications running on the cloud infrastructure[3]. The applications are accessible from various client devices through either a thin client interface, such as a web browser, or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings [MG11].

2.4 Serverless Computing

This section explains how the term "serverless computing" fits into all of this. The idea of serverless computing is building on the paradigm of cloud computing. It is another cloud offering which, unlike other offerings, follows an event driven approach. While these events may be triggered by sources outside of the corresponding cloud, they can be triggered from internal events as well [MB17]. This event-driven approach enables a different, more independent, way of composing different services with each other. Different events trigger different functions whereas each event is designed to trigger a specific function. Therefore, this approach enables coordinating the compositon of microservices rather than having the client or a dedicated middleware managing it [BCC+17]. More specifically, the user does not have to worry about specific components which might be needed for running an application or service. For instance, he is not required to provide a runtime or to take care of keeping the platform up-to-date. Of course there are servers, and there is some kind of infrastructure and runtime required for running the code. But with serverless computing the provisioning of resources, the required runtime as well as automatic scaling and fault-tolerance are all outsourced to the provider. The user simply has to write the code, deploy it to the platform and define event triggers for invoking it.
Unlike other cloud offerings, serverless functions do not allocate resources until they are actually needed. On the other hand, they remove allocated resources if they are not required anymore. Consequently, the function "scales to zero" at which point all resources for that particular function execution have been removed. This means that the allocation of resources for function execution usually has to be done upon invocation. In order to execute a function, the user pays the penalty of getting serverless code to run which is also referred to as "cold start" [BCC+17]. This does lead to slightly longer response times since the function execution cannot start immediately but has to be delayed until the resources have been allocated. But performing a cold start in order to provision a function on-demand only takes a few milliseconds and, therefore, is faster than provisioning a VM [SMM18].

[3] A cloud infrastructure is the collection of hardware and software that enables the five essential characteristics of cloud computing. The cloud infrastructure can be viewed as containing both a physical layer and an abstraction layer. The physical layer consists of the hardware resources that are necessary to support the cloud services being provided, and typically includes server, storage and network components. The abstraction layer consists of the software deployed across the physical layer, which manifests the essential cloud characteristics. Conceptually the abstraction layer sits above the physical layer.

Because of this, serverless computing might be a good fit for background tasks which are not required to provide some kind of response to a customer. For instance, a function may be triggered by an event such as a new image which was uploaded to the storage. This function now might do something with that image (e.g. compute a thumbnail) and store the result somewhere else. Such a task would not impact the user experience because the cold start problem in this case does not influence response times. However, invoking a function and expecting some kind of response is not excluded. It has to be evaluated if the response time, which includes the time for resource provisioning and execution, is adequate for that particular use case. For example, is it adequate to wait for about 20 ms on resource provisioning if the function executes within 50 ms? This strongly depends on each particular use case a function is considered to be used for. Furthermore the interval of invocations has to be considered as well because a function which is not invoked regularly might be more efficient with respect to the cost. This is because the customer does not have to pay for a function if it is not invoked which is one of the main differences compared to other cloud offerings. In comparison, using an instance of a VM has to be paid per hour whether the instance is busy or not. Serverless functions, on the other hand, usually have to be paid per invocation and execution time while there is nothing to be billed for if they are not invoked.
The approach of serverless computing strives for simplifying the programming model for developing applications. It forces developers to split up large applications into separate and independent functions which increases scalability [MB17]. Consequently, serverless computing fits into the microservices approach which aims at composing small services rather than large applications [Eiv17]. This results in small and easy to understand functions, which not only increase scalability but their maintainability significantly as well. These aspects of serverless computing are precisely what the microservice architecture approach is designed to do [MB17]. But this kind of separation does not only come with positive effects. For instance, it raises the problem of having to manage a large amount of functions [MB17].
Apart from requiring the customer to create microservices, which are to be deployed on a serverless platform, this approach relieves them from other previous responsibilities. The IaaS and PaaS approach requires the customer to maintain their resources, enable them to scale and provide required run-times, libraries and so on. The programming model of serverless computing removes these operational concerns shifting the responsiblity to the provider [BCC+17]. This means that the customer can focus on writing code without having to worry about the underlying platform and infrastructure. There are of, course, restrictions concerning the run-times supported by the different providers which developers have to comply with. Serverless platforms, therefore, can be seen as refinement of PaaS which may additionally complement other artifact types which are deployed on PaaS or underlying IaaS environments [SMM18] In contrast to PaaS, the developer is not bound to use pre-packaged applications but can write arbitrary code [BCC+17].
Considering the NIST definitions of cloud computing (see 2), FaaS would probably be ranked after PaaS. This is due to the fact that PaaS provides more control over the underlying infrastructure than serverless does. Serverless plattforms are using a shared infrastructure which the developer has absolutely no control over while PaaS is using custom infrastructure

[BCC+17]. Developers of course have full control over their code similar to PaaS. However, SaaS offerings do not grant this kind of control over application code as the complete software stack is managed by the provider. As a result, serverless functions may be ranked between PaaS and SaaS.
Serverless computing is still an active part of development and there are multiple projects aiming at the extension of existing serverless platforms [MB17].

2.4.1 Definition of Serverless

Serverless computing is part of the cloud computing paradigm. Providing a definition for serverless computing can be difficult since many parts of such a definition may overlap with PaaS or SaaS definitions [BCC+17]. The term serverless may seem to imply that there is no need for servers anymore. However, it actually refers to shifting the management of servers and other resources needed for code execution to the provider rather than the developer [BCC+17].
Serverless computing is also often referred to as Function as a Service (FaaS) since it uses functions as its deployment unit [BCC+17]. The design of such functions is strongly attached to the microservice approach. Due to this design, the developer is forced to create loosly coupled code since serverless functions have to be stateless [WS16]. A function does not save any state reached during its execution and basically starts from scratch upon invocation. As a result, serverless functions are required to use external components such as databases in order to simulate state behavior. This also leads to more flexibility concerning the design of their function because stateless functions are easier to be reused than stateful components. The only aspects which are predetermined by the provider are supported run-times which can be used to write and eventually execute code.
FaaS follows an event-driven approach which avoids resource consumption if functions currently are not used. Consequently, they consist of code blocks which are invoked upon certain events while they do not cause any costs beforehand [Eiv17]. While code deployed on PaaS may strongly depend on the platform, FaaS loosens this kind of coupling. It gives the developer control over the code and its dependencies without being restricted by the platform [BCC+17]. Furthermore, developers may expect the execution of their deployed code to be fault-tolerant as well as to be auto-scaling [BCC+17]. This of course has to be ensured by the provider rather than the developer.
Consequently, FaaS is an event-driven approach which avoids having to manage servers or virtual resources. Serverless functions are stateless and scale corresponding to the amount of invocations without requiring any manual configuration or intervention. Furthermore, availability and fault tolerance are built-in features and do not have to be ensured by developers. Billing for using such a function is strictly restricted to actual use which avoids paying for idle resources. As a result, generated costs are more transparent since they are composed of actual use of functions.

2.4.2 Characteristics of Serverless Platforms

There are a few characteristics which should be considered when choosing a serverless platform. They enable the developer to distinguish the different providers and help compare them to each other.

cost Typically, the user pays only for running functions whereas the exact billing mostly depends on execution time and used resources such as bandwidth or memory. Although the cost per execution can be predetermined, this cost model does not necessarily reduce the total cost compared to other cloud offerings. For instance, the cost efficiency of FaaS strongly depends on the amount of invocations.

Performance Scalability of functions is taken care of by the provider. However, the amount of concurrent requests to a specific function may vary for different providers. Consequently, this has to be considered for ensuring that the chosen provider is able to deal with high workloads.

Composability Functions may depend on other functions in order to properly fulfill their purpose. Functions are usually designed to be stateless and are responsible for a specific task. But this task might depend on work another function did prior to the task's execution. For instance, persisting data requires previous handling and preparation of the data. The persistence itself might be isolated in a function which can be invoked by others. Therefore, the composability capabilities of functions might be relevant and may vary between providers.

Programming language Every piece of code is written in a specific language. Whereas most of the functionalities of such a language provided most of the time are not restricted to a specific version of that language, there might be some that are. Therefore, the languages supported by a provider need to be considered when having to choose between different providers.

Monitoring and Debugging Functions are short lived entities which makes it more difficult for any health or metrical data. This raises the question of how to monitor such functions. Tracking the amount of invocations and corresponding execution times is usually provided by the provider. Tracing errors in order to identify problems depends on the monitoring capabilities (such as log files, invocation count, etc) provider be the provider.

2.4.3 Serverless Security Aspects

Using serverless computing, the developer is not responsible for ensuring the security of servers or VMs. It is not necessary to maintain security patches since this is done by the provider. But this, of course, requires the user to trust the corresponding provider to maintain and secure their infrastructure as well as keeping it up to date [Boy17]. Most providers enable restricting access to functions through permissions such as users,roles and policies. Each of them can be assigned a key or other security credentials in order to be able to properly distinguish them. This kind of access management allows for a fine grained access policy whereas each function can be assigned its own security entity. Meaning that, while one function is assigned to only accept 10000 requests per second, another one may only respond to 5000 requests per second. Furthermore, such access management can usually not only be applied to function invocation. For example, it is possible to assign specific rights to certain users such as editing function code or to allow access to other services for that provider. A gateway can be allowed to invoke functions whereas it would be more secure to explicitly limit this access to specific functions rather than all of them. However, it is much easier for the developer to establish security permissions on gateway level while setting access restrictions on function level would be more secure [Boy17]. If a function has to do 47 things, it of course requires all the permissions for doing so [Joh17]. But using access management properly may significantly increase security since a function can be permitted to only access resources it actually needs. A function which requires many permissions, for instance, provides a much larger attack area if it accidentally was deployed with exploitable code [Joh17]. As a result, each function, each gateway and any other component should only be assigned the permissions they actually need. Therefore, it is better to keep a function as well as the corresponding permission set as small as possible which serverless computing (similar to microservice) strives for anyway. Access management may be a powerful way of restricting access of any kind to functions and services they depend on. However, the granularity for such access management might vary for different providers.

Attacking serverless platforms is more difficult compared to virtual machines. The platform does not allow to open ports, run multiple applications and most importantly it is not available all the time [WS16]. A specific function is hard to get a hold of since it is only available for its execution time which is very small.

In order to reduce the impact of such attacks, function execution can be limited by the developer by using roles, users or policies for its execution. However, functions can still be invoked in a manner similar to a Distributed Denial of Service attack (DDoS attack). This may strongly impact the cost for running the service and may enable another kind of DDoS attack [Boy17]. Although there is technically no server to bring down, the service might be invoked as many times as necessary in order for it to deny any further requests because of its security policy. This is rather similar to a DDoS attack and serverless functions are consequently, not implicitly, protected against DDoS attack.

Naturally FaaS does not prevent security issues which originate within the code. For example the serverless computing approach does not natively protect against malicious user input (e.g. SQL Injection) [Joh17]. Developers are still responsible for avoiding such flaws within their code since serverless platforms usually do not provide protection on the application layer [Boy17].

2.4.4 Managing Serverless Functions

Similar to any other application, serverless functions are required to be managed with respect to testing, debugging, monitoring, controlling and deploying them. Deploying serverless functions requires to upload the code as well as its dependencies such as external libraries. This can be done packaging these parts and uploading them to the serverless platform. It is also possible to upload each part separately on a per file basis, but this is not very efficient. However, there are some tools allowing an automated deployment of function code and all required dependencies. One example for such a tool is the open-source serverless framework. It provides a Command Line Interface (CLI) for building serverless architecture and supports various providers which the user can choose to deploy on [ser18]. In addition, it can be integrated into jenkins which allows to establish automated deployment of serverless functions. This framework as well as other ones are described in more detail in section 3.
Functions and their execution can be controlled through access management of the corresponding provider. It is possible to assign different kinds of access policies to each function and any other service for that provider. This, of course, also includes access for editing or invoking functions as well as access rights for other services which functions might be required to use. The access management is described in the security section above (see section 2.4.3).
In order to ensure the behavior of functions, they have to be tested like any other application. Testing serverless functions can be done in a similar manner to testing microservices. Within the microservice architecture, it is possible to test each service in a highly isolated manner since the system consists of independent and autonomous services [SRT15]. However, testing the overall functionality can be very difficult because testing from one end to another involves many parts, which might have reasons to fail [Fow14c]. Although there are not many approaches for testing serverless applications, some approaches for testing microservices (e.g. as proposed by Martin Fowler [Fow14c]) may be adapted for the serverless approach.
Debugging functions is a more challenging task in a serverless environment. Since there are no servers developers can access, and functions are running for shorter amounts of time, it becomes harder to identify problems and bottlenecks [BCC+17]. For instance, it is rather difficult to reproduce the exact invocation sequence of a single request, because such a sequence may be dynamic and depend on input data, the origin of the request, and so on. As a result, most providers write information about them in log files, which can be accessed by the developer. Information logged into such a file may be customized by the developer and enriched with some kind of identifier in order to correlate function executions with each other. However, with the increasing amount of function invocations it

becomes increasingly difficult to evaluate all logs to identify problems. Since it is usually not supported by providers to debug deployed functions, this has to be done locally. In order to locally test functions Amazon recently launched a beta of a tool allowing the user to locally build and test serverless applications [Ama17]. Apart from AWS SAM, there are some tools for running and debugging functions locally [Cui17]. For instance, the serverless framework allows to locally invoke functions whereas it does not support debugging. To be able to debug function execution, another tool such as Visual Studio Code can be combined with the serverless framework. Visual Studio Code allows to run external programs within its debugging configuration. In case of Node.js a configuration running the serverless framework can be created whereas the user can pass the arguments required for the framework to invoke the function [Cui17]. But there are functions developed in other languages as well. The ability to debug functions depends on the language as well as available tools for them. This was also stated in [YCCI16] which presented an approach for using serverless functions for a chatbot.

building a chatbot with serverless computing In [YCCI16] a serverless architecture and a prototype of a chatbot is presented. The serverless platform which the prototype was deployed on was IBM Openwhisk. This paper also researches current challenges with respect to developing serverless functions. One example is programming and debugging functions which are composed to realize a certain flow. This was found to be rather challenging due to a lack of appropriate tools enabling an easy way for debugging short living functions. Furthermore, some concerns were expressed concerning the monitoring capabilities which, similar to the operational tasks of the underlying architecture, are the responsibility of the provider. The provider hereby has to ensure that the monitoring infrastructure scales to potentially billions of functions. These concerns about the monitoring capabilities are further analyzed within this thesis. One primary aspect of the case study in this thesis is considering monitoring approaches which address the problem of short living functions. It aims to present a scalable, provider independent approach for monitoring serverless functions.
Another problem was identified for cross provider communication since the quality of service for different providers may differ. However, the prototype turned out to be inherently scalable, although it was not explicitly designed for performance because the focus of this prototype was on an extensible serverless architecture.

Ensuring proper behavior and execution of functions requires constant monitoring of applications. Monitoring solutions often come with some kind of dashboard which summarizes the status of all components of the monitored platform. This summary might contain additional information such as workload, response times or other elements with critical impact on the application [RR16, page 45-46]. Serverless computing may require slightly different approaches for monitoring since there is no application which can be queried for application state data. Of course basic monitoring features concerning invocation amount and execution time usually are offered by the provider. This also includes storing execution logs which functions can write into during execution. Furthermore, some providers allow

the developer to push custom metrics gathered by their function to the providers monitoring environment. For example, Amazon supports this for its service AWS CloudWatch and provides interfaces for sending metrics. However, the customer may be charged additionally for using these services and accessing metrics. In order to avoid spending the money which could be saved with serverless approaches on monitoring, it might be more efficient to use cheaper solutions for monitoring. A common solution for monitoring applications is the open-source toolkit “Prometheus” which is described below.

Prometheus An open-source solution for monitoring applications such as Prometheus might reduce the cost for enable monitoring while providing gathered metrics centrally and possibly even aggregated [Aut17]. Prometheus is a systems monitoring and alerting toolkit which uses a time-series database for storing metrics [BB16]. Prometheus requires applications to expose their gathered metrics as http endpoint from which it will scrape them. Short living jobs such as batch jobs or serverless functions are not available for being scraped by Prometheus. Therefore, Prometheus offers a Push Gateway which such jobs can send their data to prior to their termination. Prometheus will than scrape these metrics from the gateway. This approach seems to be a good fit for gathering metrics from functions on a serverless platform. Especially the fact that Prometheus is an open-source solution may help at further reducing costs for monitoring or rather avoid to increase them.

2.5 Hybrid Cloud Deployment

Choosing a cloud provider is not always a simple task. There are many aspects which have to be considered and compared between different providers. One of the main reasons for considering a deployment of one application across multiple providers might be to increase the availability of that application. Deploying an application across multiple providers is also called hybrid cloud deployment. Companies such as Amazon, Google and Microsoft advertise their services as highly available. For instance, Amazon Simple Storage Service (S3) is designed to be available 99.99% within one year [Ama18]. This raises expectations among users for other services (not necessarily provided by Amazon), which is not easy to accomplish [AFG10]. Although other companies might not be able to ensure, this they may charge a fraction for using their services compared to Amazon, Google or Microsoft. It may be a good idea to provide essential services of an application on a provider who is promising high availability while deploying other services on other providers. For instance, services which do not need to be executed in real-time and are more flexible concerning the actual time they are executed at may be hosted on another provider. This might be more cost efficient because the essential components are highly available whereas the other ones are not as costly concerning their execution. In [McG17] and [MB17] McGrath et al evaluates the serverless computing approach. They study serverless implementation considerations and aim at providing a baseline for comparing existing platforms to each other.

Type	Price/GB
Outbound Data (Egress)	$0.12
Outbound Data per month	5GB Free
Inbound Data (Ingress)	Free
Outbound Data to Google APIs in the same region	Free

Figure 2.6: Networking Cost for Google Cloud Functions (source: [Goo18])

serverless computing: applications, design, implementation, and performance In [McG17] as well as [MB17] applications which make use of serverless platforms are discussed and described. Additionally, it is explored how existing applications can be adapted in order to enable them to run in a serverless environment. Another part of this thesis is the design of a performance-oriented serverless computing platform which is realized in .NET and deployed on Microsoft Azure. In order to evaluate the execution performance of this approach, different metrics are introduced allowing to compare this approach to other platforms. Having created a serverless platform prototype for better comparability and easier testing of performance, the conclusion regarding serverless computing was promising. The evaluation of the prototype has shown that serverless computing indeed improves different aspects such as performance and cost utilization. Furthermore the effort for operating the prototype was significantly less compared to previous ways of operating applications in the cloud. The comparison aimed at studying serverless implementation considerations and provide a baseline for existing platform comparison [McG17]. However, it was not designed to compare serverless platforms to other cloud offerings such as AWS EC2. Particularly it was not intended to compare an application deployed on an IaaS platform to an application deployed on the serverless platform. This kind of comparison is part of the case study conducted in this thesis.

Cloud computing Application Programming Interfaces (APIs) as well as those of serverless platforms are not yet standardized [AFG10]. Cloud providers usually deploy proprietary APIs to access and connect their offered services. This might result in a vendor lock-in which might require a substantial effort during migration to a different provider. This impedes a hybrid cloud deployment because developers have to implement against different APIs in order to realize some kind of communication between components. Therefore distributing the deployment of an application among different cloud providers is not efficient since every proprietary API implementation has to be adapted accordingly [AFG10]. Furthermore, providers impede hybrid cloud deployment by applying additional charges for data leaving their service environment. Figure 2.6 shows the current networking cost for Google Cloud Functions. It stands out, that Google explicitly charges for outbound data. These charges apply for transferring data to another region within the Google cloud platform and especially for sending data to services outside of the platform. Realizing applications across multiple providers therefore seems rather restricted because it comes with additional cost and additional effort for complying with different APIs.

2.6 Benefits and Drawbacks of Serverless Computing

Serverless computing ranked between PaaS and SaaS offerings addresses several topics which so far required managing resources in addition to writing code. This architecture comes with different trade-offs with respect to cost, control and flexibility whereas not all of them are beneficial for both - developers and providers.
Developers benefit from being able to concentrate on developing cost efficient code without having to worry about infrastructural aspects. They can focus on developing the business logic instead of having to worry about provisioning and managing servers, VMs, etc. In contrast to IaaS, the provider is now responsible for taking care of scalability and fault tolerance which raises the effort for providing a FaaS offering [BCC+17]. Of course these aspects (e.g. small latency, scalability and elasticity) lead to additional efforts for providers [BCC+17]. However, this could also be interpreted as advantage for the provider. From the provider perspective these aspects result in increased control over the software stack. Although they have to put more effort in ensuring certain aspects, they do not have to worry about many side effects within the software stack. Providers control the full software stack up until the provisioning and execution of code. Furthermore, they define the supported programming languages which they have to optimize their platform for. This also enables an easier and more transparent provisioning of patches and optimizations [BCC+17].
On the other hand applications might need specific versions of a run-time environment. This could impede developing functions which may need certain versions of run-time environments. Consequently such a constraint might be considered a drawback for application development [BCC+17].
Customizing the software stack and particularly customizing APIs may reduce the providers efforts for establishing and maintaining such a platform. But for developers this leads to a strong dependence on these proprietary APIs and consequently to the providers ecosystem. This kind of dependence may result in a vendor lock-in which strongly impedes the portability of the application with respect to deploying it to another provider.

3 Research Methodology

Recently, multiple providers of serverless platforms have emerged. One of the first ones to introduce serverless functions was Amazon announcing AWS Lambda at AWS re:Invent, Amazon Web Services in November 2014 [inc14]. Today, there are several serverless computing systems as well as some frameworks, which faciliate developing and deploying functions on these platforms, on the market . The amount of such offers is still increasing as the recent announcement of Pivotal, to also include serverless functions in its Cloud Foundry platform, shows[Han17].
The increasing amount of serverless providers raises the question of when and how serverless computing can be used. It is still an active part of development and there are multiple projects working on further use cases for applying this approach [MB17]. It is not quite clear in which areas serverless computing is a more beneficial approach compared to the previous ones. In this thesis, a case study is conducted for researching differences between deploying applications on IaaS and FaaS products. It focuses on their scaling capability, on differences in cost and on impacts on response times with respect to sudden changes in workload. In order to analyze these aspects, the different approaches have to be monitored. Provider specific monitoring solutions usually are not for free. Since the cost of both approaches are compared as well, the case study is also designed to compare different monitoring approaches. Furthermore, it strives for comparing provider specific solutions to independent ones. For that reason, the open-source toolkit Prometheus is used for the provider independent monitoring which was introduced in section 2.4.4. Additionally, the Serverless Framework is made use of for deploying on a serverless platform. This framework was chosen because it supports a broad variety of providers to deploy functions on.
This chapter creates the basis for the case study which is described in the following chapter. In this chapter, selected providers and their offered features are introduced in order to be able to compare certain aspects provided by them. Furthermore, some of the available frameworks for working with serverless platforms will be described. Considering the different frameworks, a distinction between frameworks, which help to set up an own serverless platform and those which support the user in using them, is made. In order to provide an overview for underlying technologies used in serverless platform frameworks they are briefly introduced. However, a comparison of these platform frameworks is omitted in the case study. In addition to the components taking part in the comparison of the case study, some tools for performing the tests and evaluating and presenting the results are required as well. The load and performance testing tool Gatling is chosen for running tests on the test applications. Furthermore, the open-source tool Grafana is used for analyzing and presenting metrics gathered by Prometheus. Both tools are briefly described later in this chapter.

3.1 Serverless Providers

In this section providers offering ready to use solutions of serverless functions are introduced. These solutions can be used without any kind of setup of a platform or similar components. All the providers introduced below apply the pay per use model for serverless functions which avoids paying for idle resources. Furthermore, each of them offer an event-driven service providing all features which were defined to be expected from serverless platforms (see Chapter 2.4). This also includes automatic scaling, built-in fault tolerance as well as automated administration of the underlying infrastructure. Since Amazon, Microsoft and Google are the most popular providers of Cloud offers the comparison will be focused on them. Table 3.1 shows a summary of the provided serverless features for those three providers.

Provider	supported run-times	Event triggers	Access management	Dependency management
Amazon	Java, Node.js, C#, Python	S3 bucket, DynamoDB, HTTPS Endpoint Scheduled events And many more [1]	Users, Roles, Policies API keys	NPM
Microsoft	C# F#, Node.js	Azure Cosmos DB, Azure Blob Storage, Azure Queue Storage, HTTPS Endpoint, Timer-based invocation, GitHub Webhook	Users, Groups, Roles	NPM NuGet
Google	Node.js	Google Firebase, Google Stackdriver Logging, Google Cloud Pub/Sub, Google Cloud Storage, HTTPS Endpoint	Users, Role API keys	NPM

Table 3.1: Serverless features provided by Amazon, Microsoft and Google

[1] https://docs.aws.amazon.com/lambda/latest/dg/invoking-lambda-function.html

3.1.1 AWS Lambda

Within the product range of AWS, Amazon offers a serverless compute service called AWS Lambda. While features provided by AWS Lambda are summarized in table 3.1 the costs for using using functions are shown in table 3.2. The duration is calculated from the time the code execution begins until it returns or otherwise terminates [Ama18]. The duration is hereby rounded up to the nearest 100ms. Similar to EC2, the user is charged for data transfer related to the execution of the function. This correlates to the current charge of the EC2 data transfer rate and also applies to Lambda. The costs for transferring data are also shown in table 3.2 whereas the inbound transfer is depending on the availability zones and AWS region. The costs for outgoing data transfer are stacked and depend on the amount of bytes. For example, having transferred more than a certain amount of data out of AWS, the next 10 TB are becoming more expensive.
Amazon offers a free tier which provides 1,000,000 invocations and 400,000 GB-s of execution time per month free of charge. Note, that this does not cover data transfer from or to other AWS services [Ama18].
AWS Lambda supports several different languages for executing code which is shown in

Memory (MB)	$ / 100ms	$ / invocation	$ / GB-s (out | in)
256	$0.000000417	$0.0000002	$0.05 - $0.09 | $0.00 - $0.01
512	$0.000000834	$0.0000002	$0.05 - $0.09 | $0.00 - $0.01
1024	$0.000001667	$0.0000002	$0.05 - $0.09 | $0.00 - $0.01
1536	$0.000002501	$0.0000002	$0.05 - $0.09 | $0.00 - $0.01
2048	$0.000003334	$0.0000002	$0.05 - $0.09 | $0.00 - $0.01

Table 3.2: Cost table for AWS Lambda

table 3.1. AWS Lambda sends and expects events in JSON format. Lambda functions can be invoked in response to events within various offerings within the range of AWS which also can be found in table 3.1.
The scheduling of invocations can be set up using the schedule event capability of AWS CloudWatch [Ama18]. But Lambda functions can also be invoked from outside of AWS by defining a corresponding trigger. This can be achieved by configuring an API Gateway as trigger for the Lambda function. Having created an API Gateway, it is possible to restrict access using an API key. The need for an API key can be set for each different HTTP method, whereas multiple API keys can be used in order to realize different access control for different users. These keys are mapped to so called “usage plans” which the user has to configure in order to be able use the key for a gateway. Such a usage plan determines the amount of requests accepted by the gateway which is why at least one usage plan has to be defined. The usage plan realizes management of requests by using a token bucket which can only hold a certain amount of tokens. For each request one token is consumed which results in a maximum number of requests per second equal to the size of the bucket. Each second a certain amount of tokens is put into the bucket which enables to set an average amount of requests the gateway is able to handle each second. Furthermore such an usage plan can limit the number of requests a user can make to the API each day, week or month.

Having created the usage plan it can be added to a corresponding API as well as a stage which it should be used for. This way the API could be used for distinguishing which plan to use for a request. The user can associate the same plan with different APIs or stages which of course will also associate the API keys within the corresponding plans to the API.

3.1.2 Microsoft Azure Functions

Following the trend of serverless computing Microsoft introduced its serverless offering called Azure Functions. This offering also is realized as event-driven FaaS approach. Naturally, it enables the user to trigger functions in response to events within the Microsoft Azure environment. Furthermore, Microsoft offers to trigger functions according to a specific schedule based on timers or upon events from a webhook. The functions runtime is open-source and made available on GitHub [Mic18].
Similar to AWS Lambda, Microsoft Azure functions provides 1 million requests and 400,000 gigabyte-seconds of compute time for free each month [Mic18]. Other than that the customer is charged for what he is actually using following the pay per use model. The prices corresponding to some of the available configurations are shown in table 3.3.

Languages supported by Microsoft Azure Funtions as well as available dependency

Memory (MB)	$ / 100ms	$ / invocation	$ / GB-s (out | in)
256	$0.0000004	$0.0000002	$0.087 - $ 0.05 | $ 0.00
512	$0.0000008	$0.0000002	$0.087 - $ 0.05 | $ 0.00
1024	$0.0000016	$0.0000002	$0.087 - $ 0.05 | $ 0.00
1536	$0.0000024	$0.0000002	$0.087 - $ 0.05 | $ 0.00

Table 3.3: Free execution time depending on allocated memory

managers are shown in table 3.1. Furthermore, Microsoft provides a preview for Java support within runtime 2.x which will be available in the future. It seems that F# will not be supported in version 2.x anymore [Mic18]. Within runtime 1.x Microsoft seems to be experimenting with further languages such as Python, PHP and TypeScript although there are no previews available at this time [Mic18].
Event triggers for Microsoft Azure Functions are shown in table 3.1. Unlike other providers it is possible to trigger functions in response to specific events occurring within GitHub (e.g. new comments on certain issues). Microsoft provides access management by allowing to assign roles to users (or user groups). Additionally Microsoft allows to restrict invoking function via HTTPS by requiring an API key. Unlike the AWS API Gateway it is not possible to limit the requests using a key to a certain amount in a specific time period.

3.1.3 Google Cloud Functions

As part of its Cloud Platform, Google offers Cloud Functions as event-driven serverless compute service. However, this offering is still in beta state [Goo18]. Other than Microsoft or Amazon, Google Cloud Functions support GitHub or Bitbucket as source repository for deploying functions [Goo18].
The costs for function execution time can be found in table 3.4. Unlike the previous providers Google Cloud Functions are not only billed for allocated memory and the amount of invocations. They charge the customer additionally for CPU usage in terms of GHz per second. In contrast to the allocated memory, CPU usage is not fixed for specific functions. The actual allocation of CPU clock cycles may vary across function invocation [Goo18]. Therefore the CPU usage is only an approximation which Google provides in order to estimate the cost for function execution [Goo18]. Otherwise Google charges the customer for data transfer out of the function to somewhere outside of the Google Cloud Platform. Google offers similar to Amazon and Microsoft a free tier which provides a certain amount of function invocations, compute time and bandwidth usage without charge. Corresponding to their price model this free tier also includes free compute time. Within the free tier the customer is not charged for the first 2 million invocations. Additionally the first 5 GB of data transfer from the function to non-Google services is for free. Furthermore 400,000 GB-seconds as well as 200,000 GHz-seconds of compute time are included within this tier.
At this point Google only offers a Node.js runtime and NPM as dependency manager (see

Memory (MB)	CPU	$ / 100ms	$ / invocation	$ / GB-s (out | in)
256	400 MHz	0.000000463	$0.0000004	$0.12 | $ 0.00
512	800 MHz	0.000000925	$0.0000004	$0.12 | $ 0.00
1024	1,400 MHz	0.000001650	$0.0000004	$0.12 | $ 0.00
2048	2,400 MHz	0.000002900	$0.0000004	$0.12 | $ 0.00

Table 3.4: Cost table for Google Cloud Functions

table 3.1). Triggering functions within Google Cloud Platform is currently restricted to five different event sources which are also shown in table 3.1. Similar to AWS, Google allows to restrict access using users and roles. Although Google allows to secure function invocation over HTTPS using an API key it does not seem to be possible to define different amounts of requests for each key. Using an API key allows to invoke a function without restricting the amount of requests. However, each key can be restricted to be valid for specific type of caller such as an iOS or Android application or certain website domains [Goo18].

3.1.4 Other Providers

Apart from Amazon, Microsoft and Google there are also other providers of serverless platforms. IBM, for instance, offers Cloud Functions which are based on the open-source serverless platform framework OpenWhisk which is hosted on IBMs Bluemix platform [IBM18]. IBM Cloud Functions provide a runtime for Node.js, Python, PHP and Swift whereas NPM is supported as dependency manager. Exemplary events for invoking functions are changes in database systems, internet of things sensors or commits within a GitHub repository. IBM Cloud Functions support triggering function via HTTPS endpoints as well as enabling the user to use his functions from other applications and platforms.
Another recently announced serverless platforms is the Pivotal Function Service which will be integrated into Cloud Foundry 2.0 [Han17]. This service is realized using the open-source project "riff" and will run on top of Kubernetes [Piv18a]. riff is a serverless platform framework which will be introduced in section 3.2. Pivotal Function Service is planned to support Node.js, Shell and of course Spring/Java for creating functions. Pivotal does not yet provide any information about dependency management. However, it can be assumed that for the Spring/Java runtime the dependencies can be managed similar as they are managed in Spring projects. Triggering events is categorized into three different categories. The first category is about Web Events which contains events such as Webhook handlers and APIs to backend data services. Chat integrations as well as integration into continuous integration/delivery systems is planned to be available too. The second category contains different kinds of event-based integration cases which consist of scheduled tasks, file processing and security scanning. Furthermore, it will be possible to customize authorization through an API Gateway. The last category contains events in conjunction with large-scale streaming data. It is planned to provide event integration for bulk processing, internet of things streams and log ingestion.
While other providers offer only one serverless platform, Auth0 provides two FaaS platforms for building serverless apps. The first one is Auth0 Webtask which is a free to use serverless computing platform. It is intended to enable developers to familiarize themselves with creating serverless application. As free to use platform it comes with a soft limit of one request per second [Aut18c]. Webtask supports executing functions written in Node.js which is the only runtime provided natively whereas NPM can be used for dependency management [Aut18a]. In order to create functions, Webtask provides three programming models for Node.js which represent three kinds of functions. They can be written as simple functions requiring only one argument as callback for indicating its completion [Aut18b]. The second model enables to provide a function context containing objects such as query parameters, secrets or other kind of data. The last model represents a function with full HTTP control in order to properly handle and respond to HTTP requests. Furthermore, it is possible to create custom programming models and requires a custom adaption layer. However, there are several Webtask compilers enabling the user to write code in a domain specific language which can be used within a Webtask script using such a compiler [Aut18b]. These compilers are able to compile code written in C#, T-SQL and Express as well as CoffeeScript and Typescript [Aut18b]. At this time Webtask only supports invocation of functions via HTTP request [Aut18b]. Because Webtask is a platform for getting to know

serverless functions it is not likely that further triggers will be provided.
Auth0 offers another service called "Auth0 Extend" which is designed as a highly scalable, production-ready serverless computing solution. Auth0 Extend is very similar to Webtask since it is in fact the technology underlying the Auth0 Extend product [Aut18d]. As a result it comes with the same features with respect to supported programming language and models.
Another provider for FaaS is Spotinst. Unlike other serverless providers Spotinst does not only provide FaaS based on its own infrastructure. Spotinst functions enable the customer to choose to use computational resources from the largest cloud providers [Spo18]. This of course results in a variety of geological locations the user can choose to host functions in. By leveraging different Cloud providers compute and network capacities, Spotinst Functions are able to offer their service at better rates per compute unit [Spo18]. It also improves the availability of the service because their is no dependence to a single provider. Furthermore, this reduces the chance of a single vendor lock-in since it is possible to choose multiple cloud provider for hosting functions. However, one might argue that this could result in a vendor lock-in to Spotinst Functions instead. Spotinst offers advanced analytics by exposing everything as a time-series metric which can be filtered and viewed for each individual version of the function [Spo18]. Among supported run-times Spotinst Functions lists Node.js, Java,Python, Ruby and Go. Furthermore, it is planned to include support for shipping containers as functions which allows to specify a docker image in order to bundle a function [Spo18]. Spotinst Functions focuses on deploying HTTP endpoints in order to trigger functions. As a result it can be used for creating web applications as well as for handling requests from mobile devices, internet of things devices or other APIs [Spo18]. Furthermore, it is possible to create scheduled functions for timer-based processing. The user can define a time schedule according to which the function will be executed in a periodical manner. Spotinst Functions can also be executed in response to triggers such as changes in data, shifts in system state, queues or actions by users [Spo18].

3.1.5 Cost comparison

This section offers a comparison between Amazon, Microsoft and Google regarding impact on the costs. For receiving comparable results equally for all providers there are two different scenarios for comparing them. The first one omits the usage of network bandwidth in order to be able to compare the cost for execution time and bandwidth usage separately. Both scenarios are based on an average execution time of 2 seconds per invocation, 3 million invocations and functions allocating 1024 megabytes of memory. The second scenario is also looking at cost for network usage where as each request additionally requires 500 kilobyte of network bandwidth. This refers to transferring data out of the cloud environment of the corresponding provider.
Each scenario requires 6 million seconds of execution time due to 3 million function invocations executing for 2 seconds. Of course the cost for the 3 million invocations themselves have to be considered as well. In the second scenario each request requires 500 kilobytes of bandwidth resulting in a total consumption of 1500 gigabytes. In this comparison the free tiers of all providers are omitted because their savings do not influence

the result significantly. In case of Google, for instance, the customer would be charged for 1495GB instead of 1500GB which would hardly alter the result.
The cost for the first scenario is shown in table 3.5.

Provider	$ per requests	$ per second	cost requests	cost execution	total cost
Amazon	$0.0000002	$0.00000417	$0.40	$25.02	$25.42
Microsoft	$0.0000002	$0.000004	$0.60	$24.00	$24.60
Google	$0.0000004	$0.00000463	$0.40	$27.78	$28.18

Table 3.5: Cost Comparison for Scenario 1

As this calculation shows the differences between all providers are rather small for scenario 1 because the cost for function executions are pretty similar.
For the second scenario the used bandwidth produces additional cost. The additional cost for the bandwidth usage as well as the total cost for scenario 2 are shown in table 3.6. It

Provider	$ per GB	cost GB	$ cost scenario 1	total cost
Amazon	0.09	135	$25.42	$160.42
Microsoft	0.087	130.5	$24.60	$154.60
Google	0.12	180	$28.18	$208.18

Table 3.6: Cost Comparison for Scenario 2

becomes obvious that the differences of the cost for using bandwidth significantly vary between the providers. While Amazon and Microsoft are not that different, Google charges the customer much more for using bandwidth.
This comparison shows that the cost for Amazon and Microsoft are not that different. For deciding which provider might be a better fit, it would consequently depend on other provided features and services which are used in combination with their serverless offering.

3.2 Serverless Platform Frameworks

Serverless platform frameworks enable the user to setup own platforms for running serverless functions. These platform frameworks are designed to deal with the underlying infrastructure avoiding the need for manual management. Because deploying applications in a private cloud may be obligatory for some companies some platform frameworks for setting up a serverless platform are described below.

3.2.1 Apache Openwhisk

OpenWhisk is an open-source implementation of a distributed, event-driven compute service [Ope18]. It is designed to run on proprietary, on-premises hardware, PaaS or

IaaS cloud offers. Similar to ready to use serverless offers, OpenWhisk runs application logic in response to some kind of event. Apache OpenWhisk supports to run functions written in Java, Python, Node.js, Swift, Go and PHP. Additionally it is possible to write functions as custom executables packed as Docker containers. In order to invoke a function different external triggers can be defined and linked to certain functions. For linking triggers to functions they are associated with rules. These rules allow to either invoke a single function in response to multiple triggers or have one trigger execute multiple functions. Furthermore, it is supported to invoke functions using the corresponding CLI or iOS SDK which are communication with OpenWhisk via REST API. For setting up the OpenWhisk CLI, executables for MacOS, Linux and Windows are provided.

3.2.2 Kubeless

Another serverless platform framework is Kubeless which runs on top of Kubernetes [Kub18]. It is open-source and comes with support for Python, Node.js, Ruby and PHP and also allows to integrate custom run-times. Hereby each of these run-times are encapsulated in a container image while the reference to such an image is injected in the Kubeless controller. In order to extend Kubeless to support a custom runtime, the user has to provide a corresponding image which has to be referenced in the "configmap". Furthermore it is required to provide information on how to install dependencies for this particular runtime as well as it has to be specified which environment variables are needed.
Functions can be exposed as HTTP endpoint or be invoked in a regular manner according to a certain timer. Triggering functions, using the Kafka messaging system, is supported as well [Kub18]. Having setup Kubeless, functions can be deployed using the provided CLI which is compliant to the AWS Lambda CLI [Kub18]. Kubeless provides monitoring using Prometheus, whereas metrics for function calls and function latency are exposed by default. In order to facilitate deploying functions, there is a plugin for the serverless framework allowing to easily deploy functions in an automated manner.

3.2.3 Fission

Similar to Kubeless, Fission is a serverless platform framework built on Kubernetes. It also is an open-source project published under the Apache License and maintained by Platform9 [Fis18]. Native support of programming languages is provided for Python, Node.js, Go, C# and PHP. However, the container based approached allows to integrate support for custom languages. One of its key features is the configurable pool of containers which is maintained in order to reduce cold-start latencies [Fis18]. Fission allows functions to be triggered via HTTP or webhooks for integrating it with third-party services. For instance, it is possible to trigger functions using certain messages within Slack or GitHub. Furthermore, Fission can be triggered by certain events using subscriptions to Kubernetes watches whereas the specified function is executed when the watched set of resources changed.

3.2.4 Spring Cloud Functions

In 2017 Mark Fischer introduced the Spring Cloud Functions project extending the Spring platform with serverless features [Fis17]. The main goals of Spring Cloud Functions is to support a uniform programming model across serverless providers and the ability to run standalone in a PaaS environment [Fis17]. It strives for integrating features of Spring Boot (e.g. auto-configuration, dependency injection) in other serverless offers. Spring provides adapters for other serverless providers such as AWS and Apache Openwhisk. At this time, the provided adapters are limited to providers supporting the Java runtime for executing functions [Fis17]. Another goal is to allow to run the same code either as web endpoint, a stream process, a task or maybe even all three of them [Piv18b]. It provides wrappers for beans allowing to expose them as HTTP endpoint or stream listener/publisher. This is realized using the core interface “Function”, “Consumer” and “Supplier” defined in Java. Having implemented these interfaces they can be registered as beans. However, Spring Cloud Functions does, unlike the previously introduced platform frameworks, not enable the user to set up an actual serverless platform. It aims at facilitating developing serverless functions while promoting the implementation of business logic via functions [Fis17].

3.2.5 Iron functions

IronFunctions is another open-source serverless computing platform. It aims at a better utilization of infrastructure resources and is written in Go [Iro18]. IronFunctions uses containers as base building block for realizing functions. Therefore, it allows to use various different programming languages which, similar to other platform frameworks, may be shipped as containers for IronFunctions to use. While in its early stages, IronFunctions required Docker for packaging and distributing containers via the Docker registries, it by now supports Kubernetes and Mesosphere as well [Iro18]. This platform framework is designed to trigger functions in response to HTTP requests whereas the corresponding input for the function has be to contained in the request body. The user receives the response directly from the function which avoids to deploy a gateway sitting between the user and the function. IronFunctions provides support for the AWS Lambda format [Iro18]. This allows to import functions directly from Lambda in order to use them wherever IronFunctions comes to use. As a result, it provides the possibility to reuse existing functions by importing them from AWS Lambda rather than developing them.

3.2.6 Fn project

Another serverless platform framework based on Docker containers is the open-source “fn project”. Fn provides support for Java, Go, Ruby, Python, PHP and Node.js (including Lambda functions) out of the box [Fnp18]. Due to its container based approach it also allows to extend the amount of supported languages similar to the previous platform frameworks. Created functions can be triggered via an HTTP endpoint. At this time, Fn supports to monitor the platform by gathering traces using Zipkin or Jaeger both of which

are container based monitoring approaches. Furthermore, a Prometheus metrics endpoint is offered out of the box [Fnp18].

3.2.7 Openfaas

OpenFaaS is an open-source serverless platform framework allowing to built a serverless platform on Docker and Kubernetes. OpenFaas provides an API Gateway for triggering functions, whereas it collects Cloud Native metrics through Prometheus [Ell18]. It provides function templates for Node.js, Python an Go and further allows to use Docker files as function templates. This serverless platform can be easily set up by deploying it on a Kubernetes cluster or as Docker container. Furthermore, it provides a CLI for creating functions templates and defining them using the YAML format [Ell18].

3.2.8 Riff

The open-source serverless platform framework Riff is designed to run on Kubernetes. Functions within Riff are packaged and containers and at this time invokers for Shell, Node.js, Python and Java are provided [Ell18]. Furthermore, it allows to create custom invoker images whereas this is not documented yet. In order to send and receive events, Riff provides a component connecting functions with event brokers [Ell18]. This component (called “sidecar”) supports to send and receive events over different protocols. At this time, these protocols are HTTP, gRPC and named pipes, however, further protocols are planned to be supported as well as to increase the components abstractions in order to make the event broker pluggable [Ell18]. The main objective of this project is to provide support for event stream processing avoiding to limit functions to handle single request at a time. As mentioned previously in this thesis, Riff is used for realizing the Pivotal Function Service and integrating serverless computing into Cloud Foundry 2.0 (see section 3.1.4).

3.3 Serverless Frameworks

There are several different serverless frameworks supporting the user at developing and deploying serverless functions on corresponding platforms. For instance, they provide automated packaging and deploying of functions and take care about acquiring necessary dependencies. Some of these frameworks even support local development of functions which is another reason for introducing some of them below.

3.3.1 Serverless Framework

The serverless framework is a toolkit for deploying and operating serverless architectures. It enables to “generically” create functions. These functions can be deployed at a provider of the users choice [Ser18]. Currently Serverless allows to deploy functions and even

required components to AWS Lambda, Microsoft Azure Functions, Google Cloud Functions and IBM OpenWhisk. Furthermore it is possible to deploy functions to Kubeless, Spotinst and Webtask.
Serverless provides a CLI for handling functions. It organizes functions, events and resources in services whereas each services requires a YAML file which defines how each component is configured. Within the CLI the user may deploy single functions or choose to deploy all functions in a specific service. Additionally, it allows to locally invoke functions while streaming all logs for a certain function in a separate tab of the console [Ser18].
Defining functions and corresponding events for a certain provider can be done in the serverless.yml file. Here the user can define many properties which are applied during the deployment.

```
provider:
  name: aws
  runtime: nodejs6.10
  profile: your-provider-profile-with-credentials
  memorySize: 256
  stage: dev
  region: eu-central-1
```

Listing 3.1: Defining Service Properties in the Serverless Framework

Listing 3.1 shows the definition of a service within Serverless. Here the provider of choice is AWS where each function defined within this service is set to 256 MB of memory and will be deployed to the dev stage in the eu-central-1 region. All properties defined for the service are applied to its functions unless they are overwritten within the function definition itself. Serverless allows to declare functions using different run-times, handlers and other properties all of which can be separately defined. An example for three rather different functions is shown in listing 3.2. Here the first function defines an HTTP endpoint using the path “welcome” and is configured with the properties defined for the service (see listing 3.1). The second function realizes a file upload and therefore is chosen to be configured with 1024 MB of memory. Furthermore, it uses a different runtime (Java) and of course is required to support the HTTP POST method instead of GET. The definition of the HTTP endpoint as event trigger for function two is semantically equivalent to the one of the first function. A separation of the endpoint definition as shown for function two is required if further properties such as request templates have to be defined as well. The third function uses python as its runtime and is allocated with 256MB of memory. Other than the previous services this function defines AWS S3 as its event trigger whereas it is defined to be triggered upon object creation within bucket “photos”.

```
functions:
  welcome:
    welcome: handler.welcome
    events:
      - http: GET welcome
  doupload:
    handler: handler.doupload
    runtime: java
    memorySize: 1024
```

```
  events:
    - http:
        method: POST
        path: doupload
thumbnail:
  handler: handler.thumbnail
  runtime: python2.7
  memorySize: 512
  events:
    - s3:
      bucket: photos
      event: s3:ObjectCreated:*
```

Listing 3.2: Defining Functions in the Serverless Framework

Deploying the service defined above results in three functions being created with the configured properties. Furthermore, Serverless takes care about creating the required resources (e.g. the bucket in S3 or the API Gateway) and assigning them as triggers to the corresponding function. This is done by making use of AWS CloudFormation. However, in order to successfully deploying all components and services Serverless is required to be allowed to alter the corresponding resources within the provider. Therefore the user has to pass a profile for the service (see listing 3.1) which has to be configured with the corresponding rights for creating resources. In the example above these rights would include creating an S3 bucket, an API Gateway and the corresponding Lambda functions. Of course access to use and create CloudFormation templates is required as well. Similar to overwriting properties for certain functions, it is possible to assign different Identity and Access Management (IAM) roles to them as well. This allows to assign only the rights, a function actually needs rather than assigning all which are required for the deployment of the service.
This is only a small snippet of the capabilities the Serverless Framework provides. It allows the user to define more complex functions and services, whereas each of them can be separately defined. Although it supports deploying to different providers it is not as simple to change the target provider. This is due to different function signatures of each provider and is a limitation which even the Serverless Framework cannot circumvent. Nevertheless, it really facilitates developing, testing and deploying serverless functions. In addition, since Serverless is a NPM module it can be easily integrated in Continuous Integration/Delivery as long as NPM is supported.

3.3.2 Amazon Chalice

Chalice is a Python framework for AWS Lambda which enables the user to create and deploy applications using the Amazon API Gateway and AWS Lambda [Sar16]. This serverless framework provides a CLI for creating, deploying and managing applications. Furthermore it enables the user to automatically generate IAM policies which may be used for creating and deploying functions. Unlike the Serverless Framework, Chalice only allows to create functions triggered via the API Gateway or in a scheduled manner.
Chalice is a framework which is designed for actually building functions using the python

runtime and is not limited to managing serverless functions. However, using this framework it is only possible to deploy functions to AWS whereas its capability of assigning the different triggers within AWS is rather limited.

3.3.3 Claudia.js

Other than Chalice, Claudia.js is a deployment tool for Node.js projects [Adz16]. It allows to package and deploy functions written in Node.js to AWS Lambda. Hereby it enables to define different event triggers for each function such as an API Gateway, S3 and many more [Adz16]. Claudia.js enables the user to run automated tests locally whereas required services for function execution may be simulated. For instance, in case of a HTTP request it is possible to create an event object in order to simulate a specific HTTP request including required parameters and the corresponding method for being used in the test. Claudia.js can facilitate managing Node.js functions on AWS Lambda lifting the responsibility of packaging, configuring and deploying functions and corresponding resources.

3.3.4 Zappa

Zappa is an open-source framework supporting the user at deploying serverless functions written in Python [Zap]. It supports various triggers for executing Lambda functions. The user can define function execution in response to various AWS events such as changes in S3, DynamoDB entries, Kinesis streams or SNS messages [Zap]. Similar to the previous frameworks managing and deploying functions can be done via the Zappa CLI. However, this framework is limited to be used with AWS and the Python runtime.

3.4 Gatling Load and Performance Testing

Gatling is an open-source tool allowing the user to perform requests to web based applications. It is able to simulate a large number of users with complex behaviors [Gat18d].
First of all the user can define different HTTP requests to a specific url. For each request particular query parameters and headers can be defined which will be included in the request. Furthermore, it is possible to check the response for certain values for instance by retrieving them from HTML tags searching for them by id. These values can be saved in parameters for later use in subsequent requests [Gat18a].
Having created the requests they can be integrated in scenarios which upon execution perform these requests one after the other. Between each consequent request it is possible to define a certain amount of time to wait until the next request is performed which eases the simulation of user behaviour [Gat18b].
In order to be able to perform the actual load and performance tests the different scenarios have to be added to the execution setup. Hereby, it has to be defined in which way and how many users are simulated in a certain time period. There are various kinds of ways to

Figure 3.1: Exemplary dashboard unifying data from different sources (source: [Gra18])

simulate user behavior, for instance, a specific amount of users can be injected at a linear ramp over a given duration [Gat18c]. Injecting a certain amount of users at once or at a constant per second rate is supported as well.
After performing the tests Gatling creates a report containing aggregated data for all the requests[Gat18d]. This report is provided as HTML page whereas the collected data is wrapped into different graphs in order to analyze the test results.

3.5 Grafana - Analytics and Monitoring

Grafana is an open-source tool for analyzing time series data. It allows to visualize time series data in various different ways. It natively supports retrieving data from many different database such as Prometheus, ElasticSearch, InfluxDB, CloudWatch and many more [Gra18]. This enables to mix data from different sources and display them centrally in a single dashboard (see figure 3.1). Grafana makes it easy for the user to gather data from different monitoring sources which might be used for monitoring applications. This data can be visualized as “raw data” or displayed in an aggregated manner providing various amounts of different visualization techniques.

4 Case study

This case study aims at comparing the scalability, response times and costs of applications deployed on IaaS and FaaS platforms. Since both approaches are deployed on the same provider, the cost for using network bandwidth are omitted because the same rates for network consumption apply.
For evaluating the scalability and the response times automatic scaling is set up on the corresponding platform if it is not provided already (e.g. AWS EC2). For that reason the first scenario will be deployed on AWS EC2 which requires setting up auto-scaling policies. The second scenario is deployed on AWS Lambda which does provide auto-scaling without any manual interaction.
For developing the application deployed on EC2 it was chosen to use Spring Boot. This choice is based on the fact that Spring Boot enables to develop web applications from scratch with minimal effort. For the serverless application the language of choice was Node.js, since it is supported by all serverless providers. As a result the test application is developed as Spring Boot (Java) as well as Node.js application.
In order to avoid querying the application deployed on EC2 and Lambda from within the AWS environment the tools for testing and monitoring the test applications are deployed on BWCloud [BWC18]. The main intention of this approach is to avoid receiving falsified response times due to requests within Amazons own network.

4.1 Test Application

The test application is a user-based web application offering several services for its users to use. Figure 4.1 shows a high level overview of the test application. In order to use the services of the application a user has either to login or register.
For registering the user has to enter his information (name, address, ...) which will be saved upon submission using AWS Relational Database Service (RDS). Each registration has to be confirmed by entering a code which is sent to the user via email. After successfully entering the generated code as well as its successful validation the registration is complete and the users account is activated. For simplicity reasons a successful login is stored within a cookie containing the users first name, last name and birthday. The cookie is only checked for its presence rather than validating the user using the database for each request. In order to login the user has to provide the corresponding login information which is validated using the database. Upon successful validation a user cookie is set and the user is forwarded to the user view. If a user chooses to logout the user cookie is reset followed by a redirect to the main page.

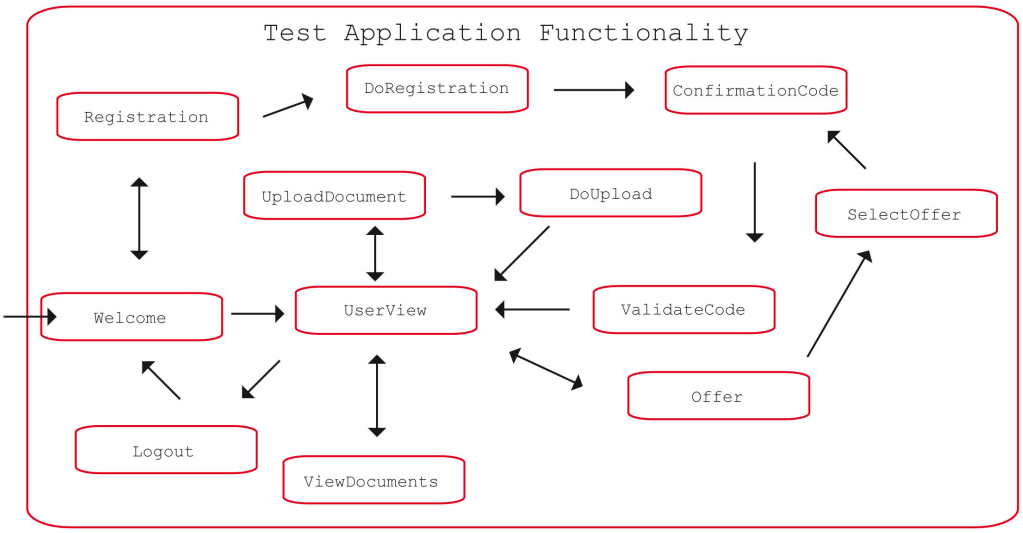

Figure 4.1: Services of the test application

Within the user view a logged in user can choose to upload a document, view all of his documents or request offers. Choosing to upload a document the user can now choose the file he would like to upload. Having chosen a file the user can submit it which results in the file being uploaded. These files are stored within AWS S3.
In order to see which files the user already uploaded he can go to the view documents page. This page retrieves a list of all files uploaded by the user and presents them within the browser.
The last service available is the request for offers. Calculating these offers is designed to take longer than any other requests which is why it takes approximately 1 second. Having created the offers the user may either choose to accept one of them or decline all of them by returning to the user view. If the user chooses an offer a confirmation code is sent to him in order to confirm his choice similar to confirming the registration.

Spring Boot Application The Spring Boot application is using Maven for managing dependencies and packaging the application. The application interacts with the database by using Hibernate which is part of Spring JPA. In order to be able to use AWS S3 and Simple Email Service (SES) the corresponding AWS SDKs are used. The Spring Boot application is deployed on EC2 using an Apache Tomcat Webserver. Furthermore, the Prometheus Node Exporter has to be installed on each instance. This is required to expose data about the underlying hardware as Prometheus endpoint and requires no further configuration. This endpoint can be scraped by Prometheus.

Node.js Application In case of the Node.js application the dependencies are managed with NPM. Using NPM the required libraries for interacting with the services in Amazon are

included in each function. The Node.js application is managed, tested and deployed using the Serverless Framework (see section 3.3.1). The framework allows to easily set required dependencies separately for each function and package them accordingly. Hereby it is also possible to include certain JavaScript files in the package of each function enabling it to reuse components which are required in different functions. Furthermore this framework creates the required API Gateway upon deployment for invoking the functions via HTTPS.

4.2 Test Monitoring

In order to keep the monitoring of the application simple it was decided to use two approaches which are compared to each other. The first approach uses AWS CloudWatch which keeps track of different metrics of AWS instances. For instance it measures CPU, memory or network usage and other hardware related metrics. However, the use of CloudWatch is not for free and the user is charged for monitoring instances, recording executions logs and other features it provides. Especially receiving information about instances in the interval of 1 minute (instead of 5) is additionally charged. Storing logs is charged on a per gigabyte rate. The second monitoring approach uses the open-source monitoring system Prometheus which was introduced in section 2.4.4. The results of these approaches will be compared to each other focusing on the effort for embedding and using them as well as their cost.
In this case study Prometheus is deployed as Docker container on an instance on BWCloud. Prometheus scrapes the currently running instances of the AWS account used for this study. Since in case of AWS Lambda there are no running instances which could be scraped an instance of the Prometheus Push Gateway is also deployed. Applications (and especially short living functions such as Lambda) can push their gathered metrics to this gateway prior to their termination. Prometheus may then scrape the Push Gateway in order to obtain these metrics.
In order to properly display the gathered metrics the open-source analytics and monitoring platform Grafana is used. Grafana enables to visualize metrics from Prometheus putting them in many different kind of graphs and other visualizations (see section 3.5). A deployment overview of each test application is shown in figure 4.2 and figure 4.3.

4.3 Test Structure

The overall structure of this case study is illustrated in figure 4.4. The overall execution time of the test is set to 60 minutes. While the Spring Boot application is deployed on EC2 instances the Node.js is uploaded to AWS Lambda using the Serverless Framework. Hereby an EC2 instance of size m4.xlarge was chosen whereas the auto-scaling is configured to use up to 3 instances. One instance provides 4 virtual CPUs as well as 16 GB of memory. Because the amount of CPU for a function in AWS Lambda is not known, two different configurations are used in two separate tests. The first one uses functions with 1024 MB of

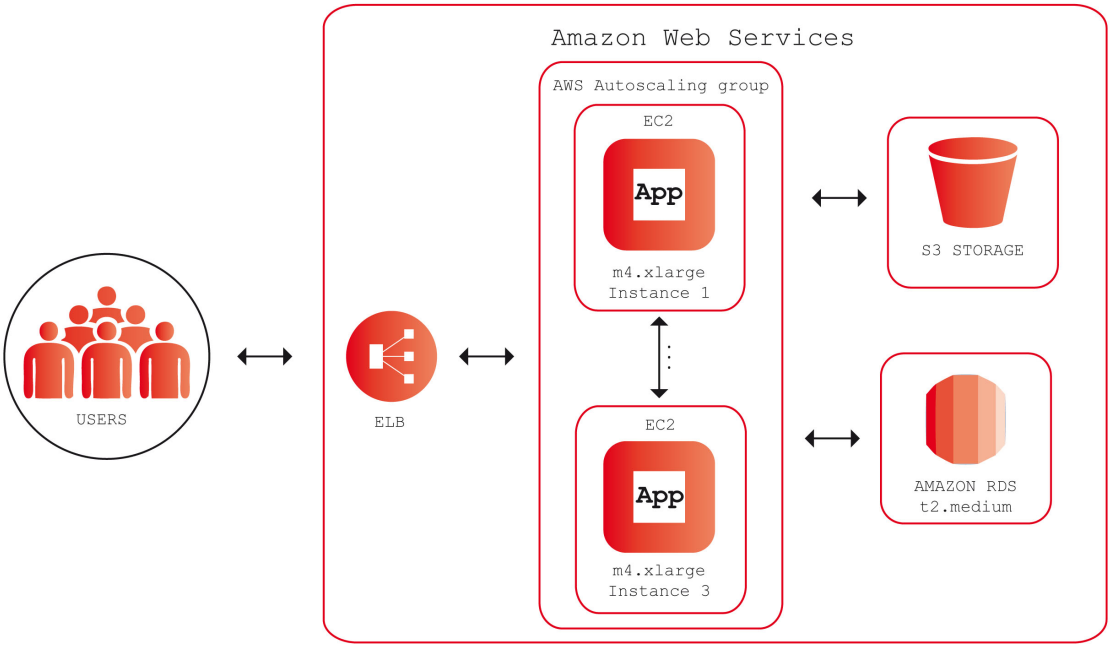

Figure 4.2: Deployment overview on AWS EC2

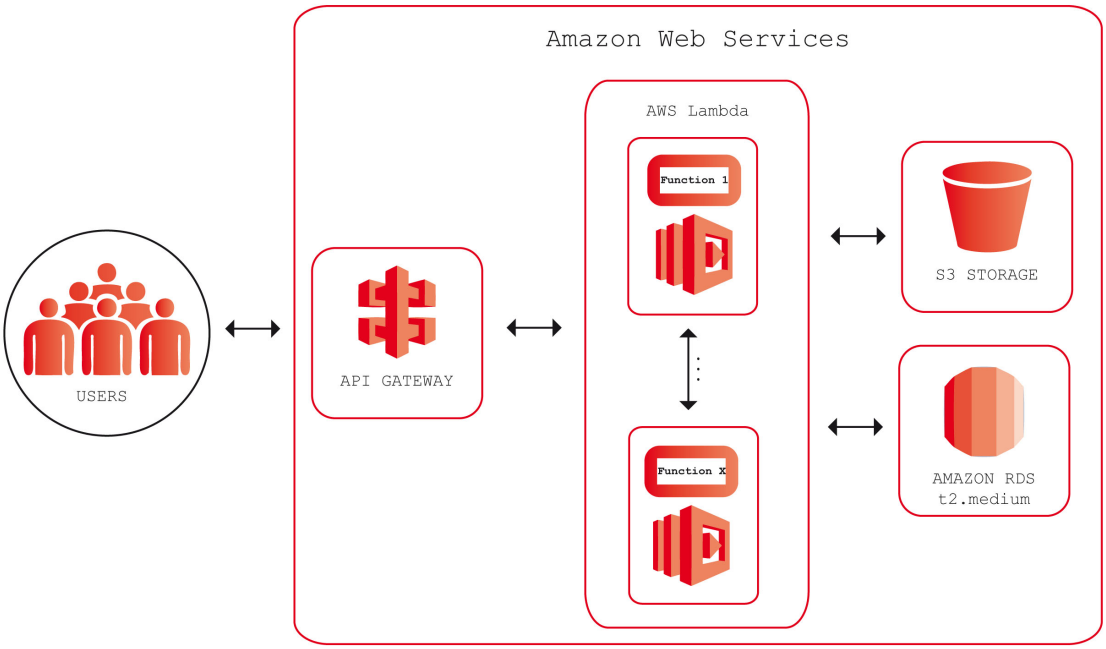

Figure 4.3: Deployment overview on AWS Lambda

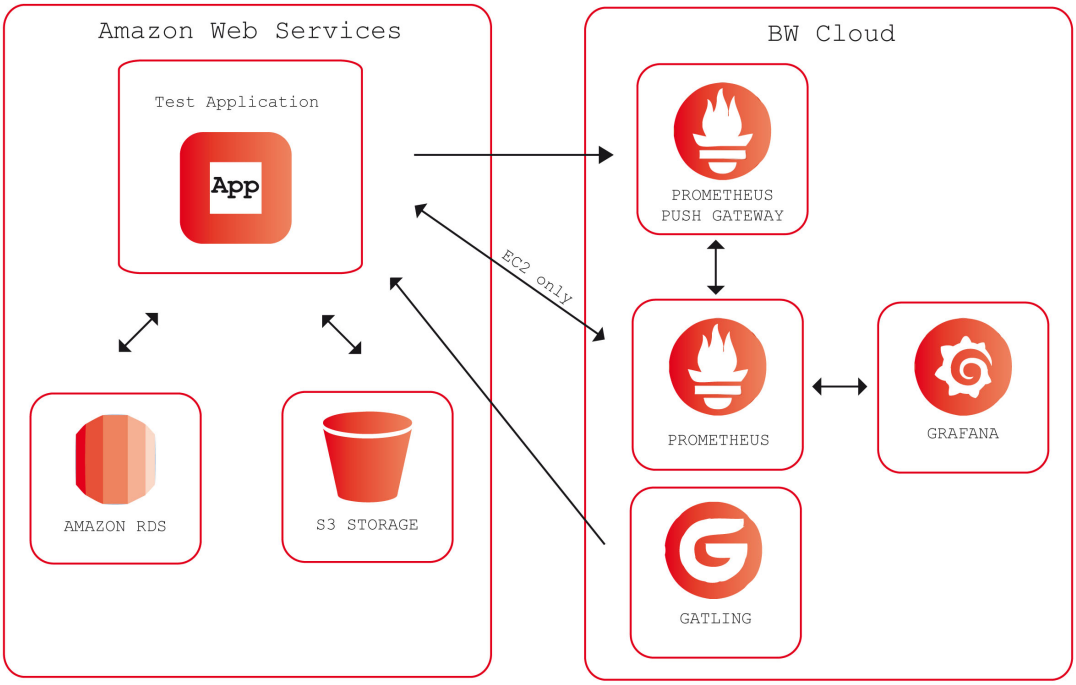

Figure 4.4: Overview over the complete test

memory while in the second one 2048 MB of memory are allocated. This aims at avoiding inaccurate response times due to lack of computing resources.
The EC2 auto-scaling was set to determine the requirement for scaling depending on the last two minutes of CPU usage. If the CPU usage was higher than 55% for two minutes another instance is added. Is it lower than 30% an instance is removed. Scaling instances takes a few minutes because an instance first has to start up followed by an healthcheck by the auto-scaling service which takes about 120 seconds. Having passed the auto-scaling healthcheck the load balancer performs healthchecks of his own prior to forwarding requests to a new instance which takes another 30 seconds. Of course, the time required for measuring data scaling decisions are based on as well as time for monitoring the impact of the scaling measure has to be accounted for as well. Therefore the test is divided into different phases and time frames as described in section 4.3.1.
Both applications are using AWS S3 for storing files and AWS RDS for handling customer related data. In order to receive meaningful results for the response times without limiting them to the database an RDS instance of size t2.medium was chosen. Although this size is not required by the test application it provides a larger pool of active connections. Since this study aims at testing the scalability of EC2 instances and functions a larger instance was chosen to ensure enough connections.
Prometheus is hosted on an instance on BW Cloud and scrapes metrics from all currently running EC2 instances. Furthermore the Prometheus Push Gateway is deployed on that same instance to which the application sends its metric data. For visualizing the gathered metrics another instance was set up on BW Cloud running Grafana. Grafana queries

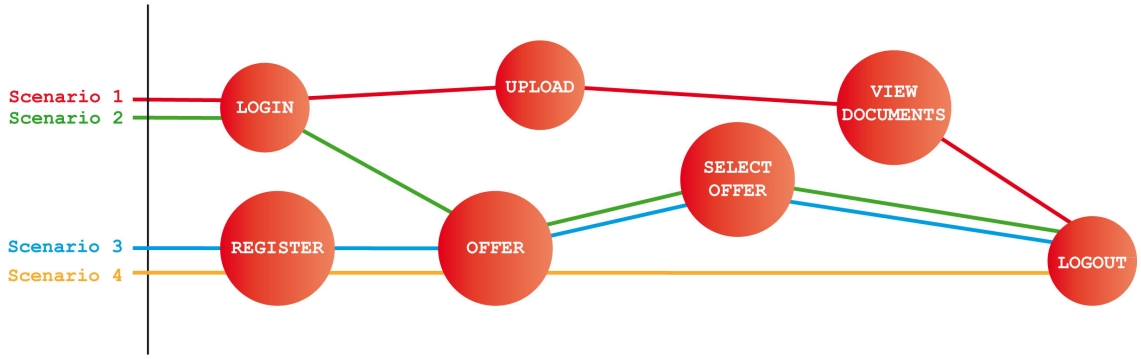

Figure 4.5: Overview of the Four Scenarios

Prometheus for certain metrics which are visualized within the Dashboards. One dashboard is used for displaying data about CPU and memory usage as well as work load for each instance. Additionally it visualizes the amount of ingoing and outgoing data traffic. Another dashboard is used for presenting the response times of servlet requests to the Spring Boot application. Measurements sent to the Push Gateway are displayed in this dashboard as well.
Performing requests to the application requires a tool for sending corresponding requests and handling response data. This is realized using Gatling, a load and performance testing tool (see section 3.4), which is deployed on another instance on BWCloud.

4.3.1 Gatling Configuration

Having created the HTTP requests the test application is able to handle, there are four different scenarios within the Gatling setup. Figure 4.5 shows an overview of the four scenarios. For simplicity reasons and to avoid forcing Gatling to parse emails containing the code, a “mastercode” is used for confirming the registration or offer requests. Nevertheless, a randomly generated code is sent via email to the user. The file size for simulating upload is 1 MB. Between each subsequent request of a simulated user it is waited 5 seconds before continuing with the next request. This is supposed to emulate the time the user requires for deciding what to do next. However, it is assumed that each user precisely knows what he intended to do and has only to decide how to achieve his goal. This also includes the time the browser needs for rendering the response page. Furthermore each of the scenarios assumes a user, who is not yet logged in and will either login in or register.
As illustrated in figure 4.5 the first scenario is about already registered users. Each user logs into the application, chooses to upload a document and reviews the success of the upload prior to logging out.

The second scenario represents already registered users requesting an offer. Having reviewed all offers the user chooses one of them and logs out after confirming his decision by entering the verification code.
Scenario 3 on the other hand assumes new users who have to register and confirm their

registration first. After successfully registering, the users in scenario 3 request offers and similar to the second scenario decide to accept one of them. As a result, the chosen offer is confirmed by entering the code and the users logs out.
The last scenario is almost equal to scenario 4, however, the user is not willing to accept any of the provided offers. As a result, the user logs out without choosing an offer.

Scenario injection setup In order to be able to compare the scalability of the two different approaches the test is split into several phases. Each phase comes with a significant change in the amount of users which will require to scale the application accordingly. However, the forth and fifth phase are not designed to force the application to scale, but for measuring the impact on response times during constant high workload periods.
Each phase will last 450 seconds and is separated into 3 time frames. The first and the last time frame of 150 seconds each is required for measuring the changes in response times prior to and after the auto scaling measures. Additionally, the first frame serves to evaluate the current work load which the decision for frame two is based on. Consequently the auto-scaling is configured to react in the second frame of 150 seconds.
Table 4.1 shows the distribution of user requests for each phase. It also states how many

Phase (750 sec)	Scenario 1 (31%)	Scenario 2 (13%)	Scenario 3 (19%)	Scenario 4 (37%)	Total (100%)
1	139	59	86	166	450
2	326	137	199	388	1050
3	628	263	385	749	2025
4	977	409	599	1165	3150
5	977	409	599	1165	3150
6	465	195	285	555	1500
7	233	97	143	277	750
8	139	59	86	166	450

Table 4.1: Users Injected in each Phase

users are injected in each scenario. The percentage of users for each scenario was arbitrary chosen. This percentage was required in order to simplify the tests by allowing to define a total amount of users while the amount for each scenario is calculated applying the corresponding percentage. The amount of total users on the other hand was determined by testing how many requests can be served before reaching a certain threshold of CPU usage. This is required in order to “enforce” scaling before the workload is too high for a certain amount of time. However, this was not easy at all since the actual CPU utilization strongly depends on the work distribution of the load balancer. AWS auto-scaling requires some time before a recently started instance can be targeted with requests (as described in section 4.3 below). Considering this as well as including time for monitoring the effects, each time frame was set to 150 seconds.

4.4 Results and Evaluation

In this section the results of the case study are described and evaluated. The metrics visualized in grafana as well as the summary of requests and responses per second created by Gatling can be found in section A.

4.4.1 Deploying with Serverless Framework

Deploying the test function with the Serverless Framework was very simple and easy. As this framework supports packaging functions with individual files, each function can be shipped with exactly the files it requirs. For instance, the services requiring the a data source (e.g. viewdocuments) can be configured to have the corresponding implementation file included. Listing 4.1 shows an exemplary definitions of the viewdocuments service.

```
viewdocuments:
  handler: viewdocuments.handler
  events:
    - http: GET viewdocuments
  package:
    include:
      - viewdocuments.js
      - viewdocuments.html
      - handleFile.js
```

Listing 4.1: Defining a Function with Custom Packages

The created services can easily be deployed using the frameworks CLI. Hereby it is possible to either deploy all functions or package and deploy single functions (see listing **??**). Furthermore the Serverless Framework allows to locally invoke functions in order to test them prior to their deployment.

```
serverless deploy
serverless deploy -f viewdocuments
serverless invoke -f viewdocuments
```

Listing 4.2: Deploying Functions with Serverless

Having set up the required credentials for a provider, it is very easy to deploy functions with this framework. The deployment takes about 30 seconds and, of course, includes the creation of all services required by the deployed functions (e.g. API Gateway). The Serverless Framework facilitates creating, testing and managing functions and is very easy to use.

4.4.2 Monitoring

In order to monitor the test application on both, AWS EC2 and AWS Lambda, Prometheus and AWS CloudWatch were used. In both approaches the user first has to create a dashboard on which the data can be visualized. As CloudWatch naturally does not have access to application specific measurement and a corresponding integration was not done, comparing

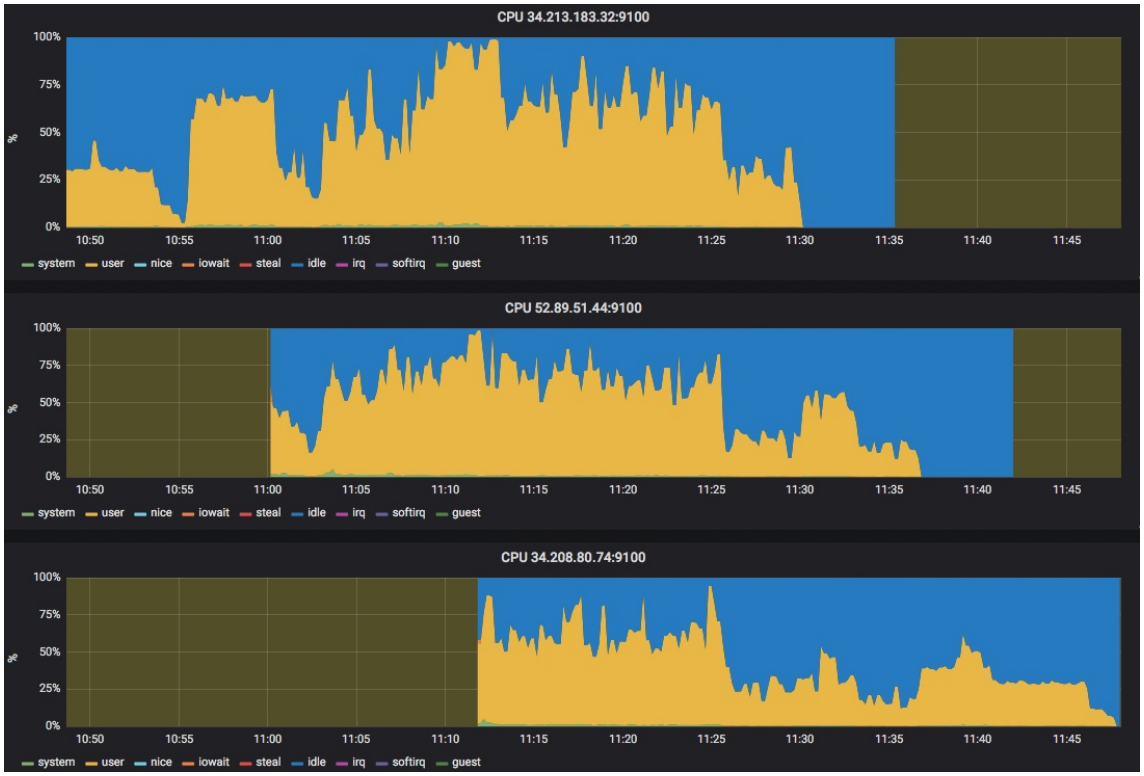

Figure 4.6: CPU usage of the EC2 instances during the test - Prometheus

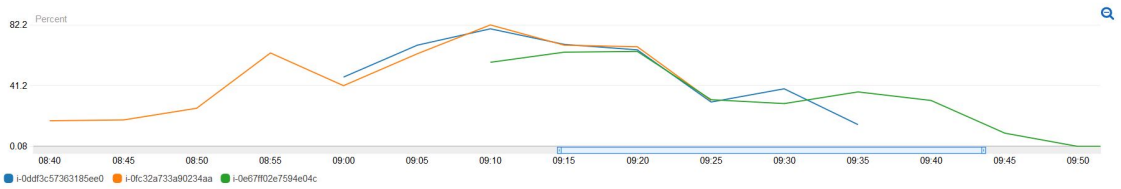

Figure 4.7: CPU usage of the EC2 instances during the test - CloudWatch

these measurements is omitted in this section.
AWS CloudWatch makes it rather difficult to search and find specific components. Although there are categories for each different service which in case of EC2 can be filtered by "Auto Scaling Group" it is not possible to retrieve single instances this way. The only way of adding single instances rather than already aggregated groups is to search for these instances by using there id (e.g. "i-043d89a9aba35d20c"). Furthermore the listed instances which can be added to the dashboard are not filtered by the time frame chosen by the user. The list seems to always contain all instances which have been active in the past which makes it almost impossible to search for them by hand. Prometheus on the other hand enables the user to filter data within the query. The functional expression language provided by Prometheus allows to include selectors in the query in order to retrieve only data the user is interested in. However, this requires that the corresponding time series data is labeled. As a result the user can decide whether to show the data in an aggregated

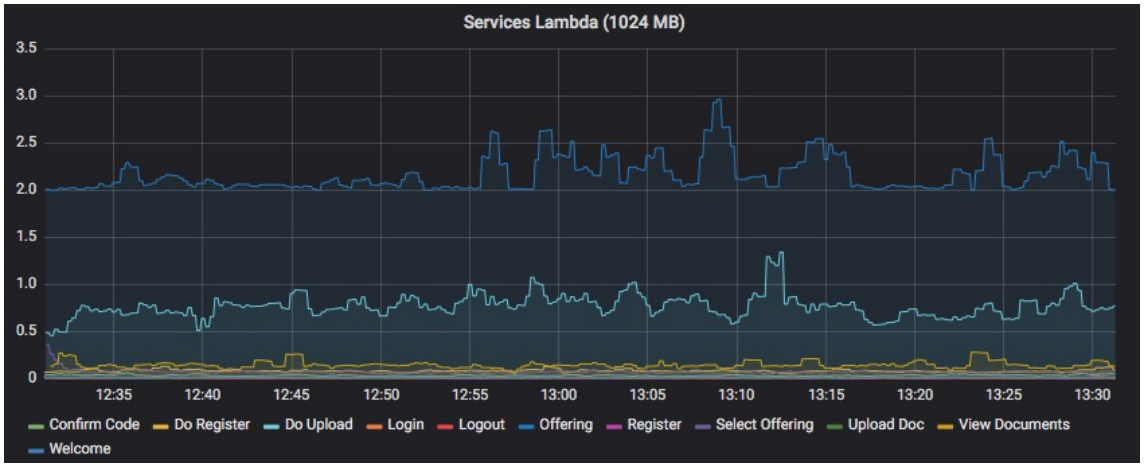

Figure 4.8: Execution duration - Prometheus

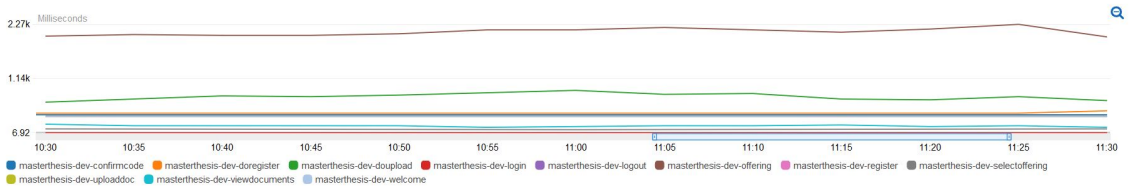

Figure 4.9: Function execution duration - CloudWatch

manner or separated by label. Figure 4.6 and 4.7 show the data gathered by Prometheus (visualized in Grafana) and CloudWatch with respect to the current CPU usage during the test. While CloudWatch only shows the current CPU usage with respect to each single instance Prometheus also distinguishes between the kind of usage. Of course, it can be filtered by specific CPU usage modes (e.g. system, user and idle only) or visualize the CPU usage within one graph instead of three. In the latter case the data is aggregated similar to CloudWatch although the visualized data is presented more coarse grained in AWS CloudWatch.
As stated earlier in this thesis, monitoring serverless functions is more difficult since there are no instances which can be queried for data. In order to be able to gather metrics about function execution and compare them to metrics gathered by CloudWatch the Prometheus Push Gateway was deployed. The results for function execution for the two different approaches does not differ significantly. Although metrics gathered by Prometheus and presented by Grafana do seem more detailed than those of CloudWatch as can be seen in figure 4.8 and 4.9 respectively. However, CloudWatch presents slightly higher execution times than Prometheus does. This is due to the fact that data received by the Prometheus Push Gateway is calculated during function execution. As a result, the function itself has been running for a few milliseconds before it can begin the measurement. Of course, the function is executing a little while longer than the actual measurement for Prometheus does. CloudWatch on the other hand has more precise information about begin and end of function execution since it has access to the platform the function is executed on.

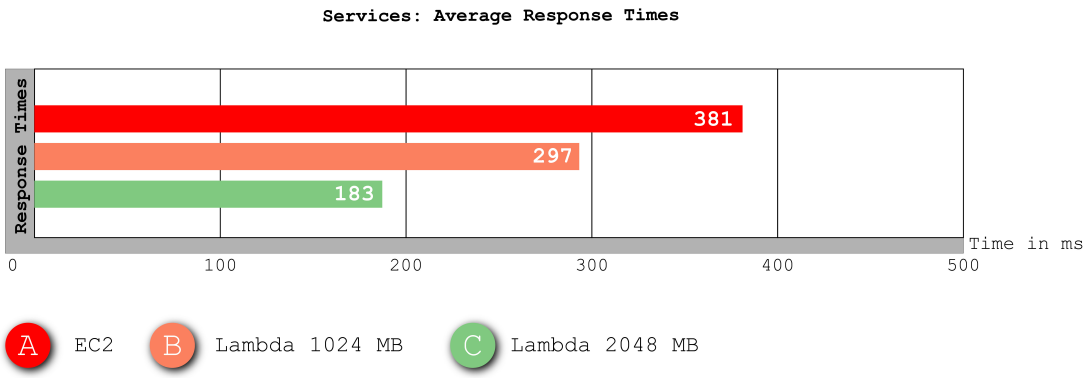

Figure 4.10: Average service response times

4.4.3 Response Times

Since AWS CloudWatch does not receive any data about the response times of the application, the evaluation of the response times is based on the measurements gathered by Prometheus. Figure 4.10 shows the average service response times for the three approaches. During the test the application deployed on EC2 presented higher response times compared to both Lambda approaches. As Lambda functions are executed similar to containers, their execution does not interfere with each other. On EC2 on the other hand the response times appear increasingly shaky as the test goes on and even rise significantly for a short period of time (see figure A.2). This peak seems to be the result of a delayed scaling measure as the CPU usage within that period was above 80% for quite some time. However, the smaller Lambda functions show differences in response times as well whereas it is less affected by workload peaks (see figure A.2). Although, this lambda functions shows higher response times for the offer service than the EC2 instance does as illustrated in figure 4.11. Comparing the response time of the offer service of EC2 with the the smaller Lambda function leads to the conclusion that it has not as much computing resources as the instance. Providing twice the amount of resources results in significantly improved response times because the compute intensive task can be finished much faster. Although the higher amount of resources affects the response times for the offer service in a positive way, other services do not benefit that much (see figure A.8)
This especially applies to services requiring access to a database or data storage. Figure 4.12 shows the average response times for the file upload. Here the EC2 instance presents the smallest whereas the 2048 MB function has slightly higher and the smaller function significantly higher response times. An instance, which is permanently running, is able to maintain and reuse connections to databases and storage services. Lambda functions on the other hand have to acquire a new connection with each invocation because their resources are removed upon termination. As acquiring new connections takes some time

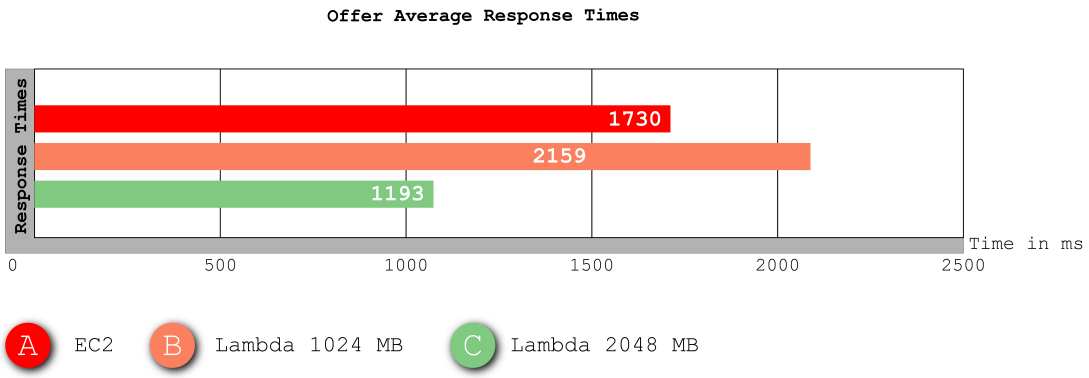

Figure 4.11: Average offer response times

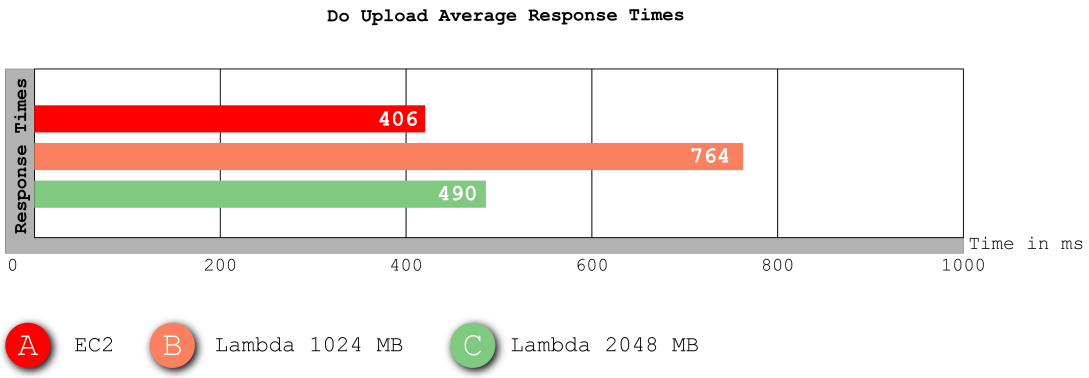

Figure 4.12: Average do upload response times

this negatively affects their response times. Of course, having more resources may mitigate this effect, but as shown in figure 4.12 the EC2 instance provides the lowest response times with respect to accessing data storage.

4.4.4 Scalability

Adapting to changing workload is essential for applications in order to satisfy all requests. During this test the workload changed rather quickly (in a matter of seconds) forcing the applications to eventually react on these changes. With quickly increasing workload it is required to increase the available resources as well in order to meet the current demand.

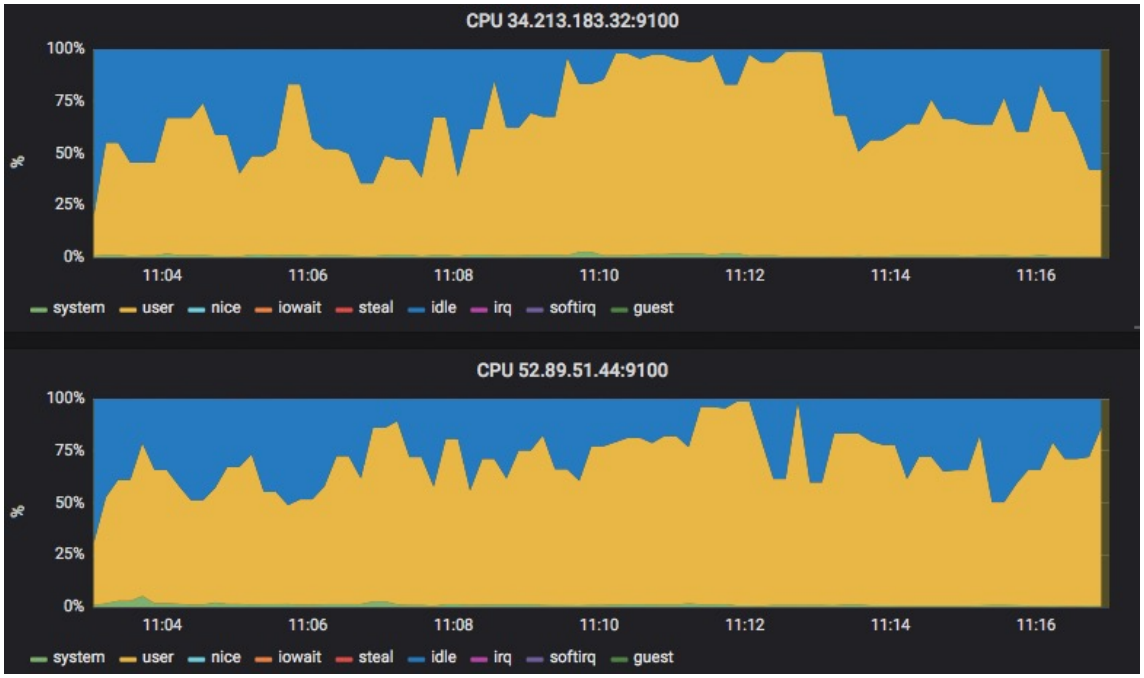

Figure 4.13: Delayed auto-scaling may lead to overload

AWS EC2 has shown, that its auto-scaling capability works reliably and is able to react on high workload. However, it takes rather long to add resources if required. Auto-scaling measures a certain time period and reacts upon consecutively reaching a certain threshold. The minimum time period is hereby 60 seconds whereas in this case study 120 were defined to be the measuring period. But after deciding to take scaling action, an instance first has to be started. Depending on the kind of instance, deployed applications and so on it might take an instance quite a while to start up. Although an instance might be started already it is not served with requests until auto-scaling as well as the designated load balancer have successfully performed their healthchecks on that instance. Figure 4.13 shows a time frame of the test run. At about 11:07 the workload significantly increased after a short period of less workload. This resulted in auto-scaling waiting another 120 seconds for gathering the required consecutive data before deciding to scale up. Meanwhile both instances were rather busy and close to being overloaded. Since starting a new instance takes a while the workload kept getting higher and the response times were getting worse (see figure A.2). On the one hand scaling up requires to define thresholds which do not yet overload the instance in order to avoid high response times or even failing instances. On the other hand these thresholds have to be chosen rather high because otherwise the auto-scaling policy might decide to scale down again because the utilization was too low after scaling. Furthermore it should be avoided to scale up and down to fast in order to save costs. However, auto-scaling meets its limits if it is required to scale up within a very short time. AWS Lambda was almost not at all influenced by these fast changes in workload because each request results in a separate container for executing functions. Although this container has to be started as well, it performs cold starts in a matter of a few milliseconds. Furthermore it does not require any manual configuration for AWS Lambda. It scales

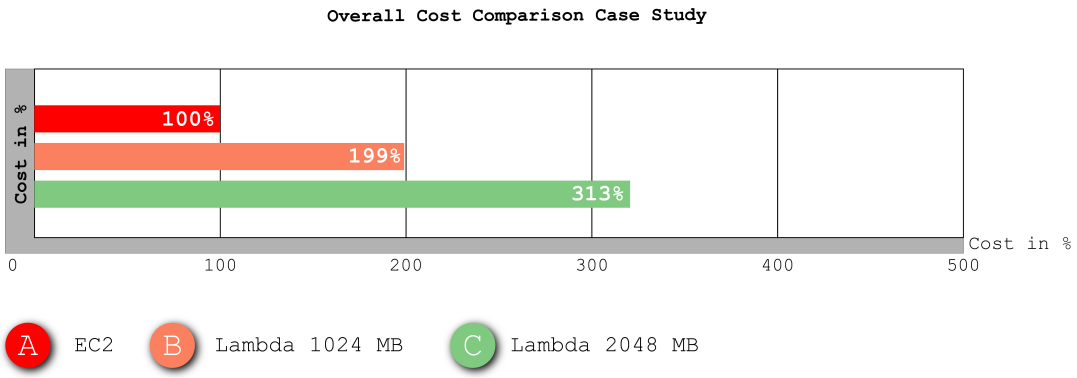

Figure 4.14: Lambda Cost Difference Compared to EC2

reliably with the workload and does not reveal significant changes in response times during workload peaks. This also applies to the API Gateway which serves as trigger for Lambda. Besides defining the accepted HTTP methods for invoking a function it does not require to be configured in order to scale.

4.4.5 Cost

The cost for the three test runs consist of different billable entities. The cost for executing the test on AWS EC2 is additionally composed of AWS CloudWatch, AWS ELB. The required database, S3 storage as well as AWS SES are omitted because these cost apply to all three tests. In addition to the cost for AWS Lambda there are charges for using API Gateway and AWS CloudWatch. The overall cost is shown in figure 4.14 where as the cost are compared to the EC2. The large difference in the cost is the result of the design of this case study. This case study was mainly designed to compare the scalability and, therefore, the function size was equally chosen for all services. Naturally, not all services require an instance with 2048 MB of memory or even 1024 MB of memory. But in order to be able to compare the impact of an increase of resources on all services, and to be able to compare them to ec2, their memory size was chosen to be equal. Figure 4.15 illustrates an estimate of the cost difference given the function size is adapted. Hereby it is assumed that all function without requiring data access (e.g. database, file storage) need 512 MB of memory. Furthermore, data access functions require 1024 MB while the compute intensive offer function requires 2048 MB of memory. Estimating the cost for the same test, the same amount of users and the same time period results in costs savings of about 1%. However, as stated above, functions are not designed to be used for every kind of service. The design of the test application as well as the amount of test users is not primarily meant for pointing out cost

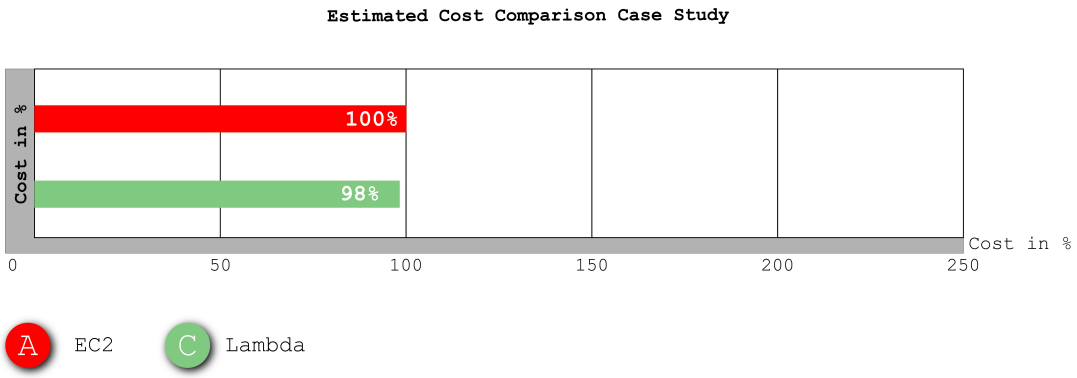

Figure 4.15: Estimated Lambda Cost Difference Compared to EC2

savings of serverless computing approaches. In order to really benefit from cost savings using serverless computing, functions should be used for appropriate tasks.

5 Summary and Future Work

The main goal of this thesis was the evaluation of serverless computing approaches. Having introduced the event-driven approach serverless computing as part of the cloud computing paradigm, several providers for serverless offerings were introduced while some of them were compared to each other. This comparison aimed at assessing the possibility of deploying serverless functions across multiple providers. Many providers supported a different set of languages, whereas Node.js was supported by all of them. However, a comparison of their cost model has shown that hybrid cloud deployment most probably is not very economic because the charges for transferring data out of the provider's platform are rather high. Another aspect impeding hybrid deployment is the deployment of proprietary APIs which limits the portability of functions.
Of course, there are various platform frameworks for building up own serverless solutions. Unfortunately, many of them are limited to HTTP endpoints as event triggers, however, there are a few platforms which provide specific components or rule based mechanisms for associating various events to function invocation.
In order to evaluate the suitability of serverless functions a case study was conducted focusing on scalability, monitoring capabilities, response times and costs. Furthermore the Serverless Framework, which supports the user at managing and deploying functions, was evaluated in this study. The Serverless Framework has proven to be a powerful tool which facilitates developing, managing and deploying functions. It allows to locally test functions, provides automated deployment as well as supports automated creation of required services on the target platform (such as storage, API Gateways).
The case study itself was conducted on AWS using AWS EC2 and AWS Lambda. Additionally the open-source tool Prometheus was used for monitoring the applications and comparing it to AWS CloudWatch. While the data gathered by CloudWatch seems very coarse grained, Prometheus gathers and stores more detailed information. However, Prometheus requires additional services and endpoints to be able to scrape the data from target applications. The core features of Prometheus for this study was the Push Gateway. This Gateway allows functions to push gathered metrics prior to their termination allowing Prometheus to scrape these metrics from the Gateway. The metrics gathered were slightly different to those of CloudWatch because the function has to measure them during execution. However, Prometheus has shown to be useful monitoring tool which combined with Grafana allows to visualize and filter metrics in an easy and efficient manner. Most importantly it is free to use and allows gathering metrics independent of any provider.
The evaluation of response times and scalability has revealed several advantages and disadvantages of the serverless computing approach. On the one hand the response times remained rather stable even during workload peaks. The serverless computing approach has shown that it is highly scalable. On the other hand the overall response times were

slightly higher compared to the application deployed on the EC2 instance. Looking at single functions it was revealed that serverless functions deal better with computational tasks than EC2. In case of access storage services, such as databases, it was determined that serverless functions have higher response times because they have to create a new connection with each invocation.
Serverless computing has shown various benefits compared to previous cloud offerings. The event-driven approach and the promising response times of computational tasks might prove to be a valuable asset for realizing background tasks in a highly efficient manner. Although, it does not appear to be a suitable approach for all kinds of tasks it might be a powerful extension to existing cloud services.

future work Based on the findings in this thesis the possibilities of serverless computing approaches with respect to extending existing cloud applications could be examined. It remains unclear how serverless functions may support other cloud offerings during high workload periods. Furthermore this thesis focused on comparing Amazon's cloud offerings whereas there are several other providers worth looking into.

A Appendix

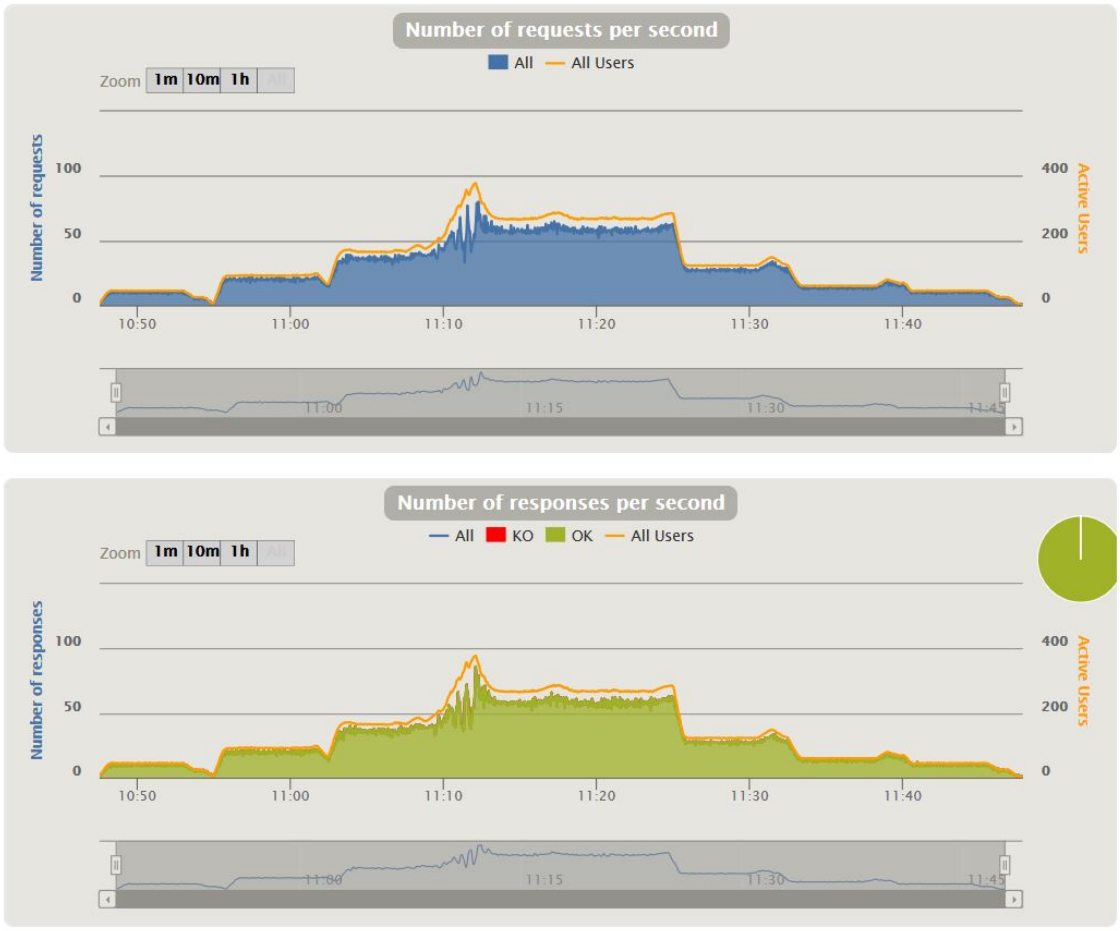

Figure A.1: Number of Requests/Reponses per Seconds and Active Users (EC2)

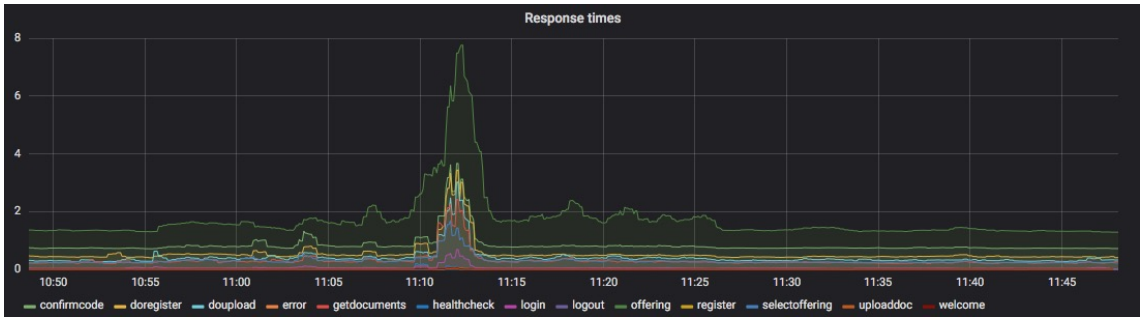

Figure A.2: AWS EC2 Response Times

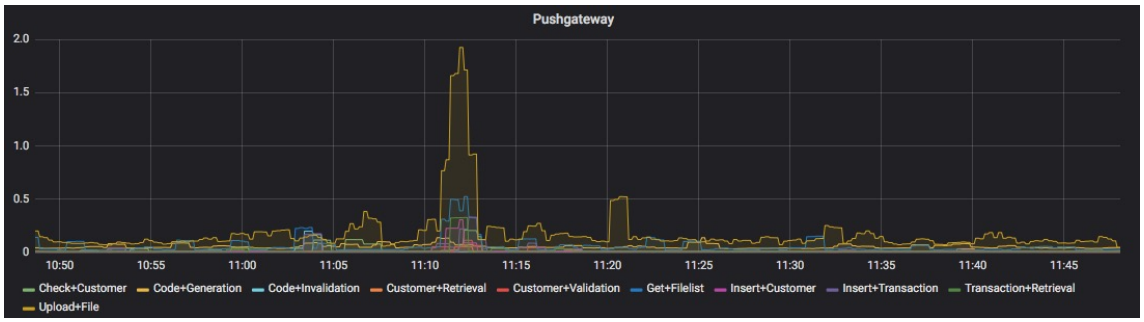

Figure A.3: AWS EC2 Data Access Times

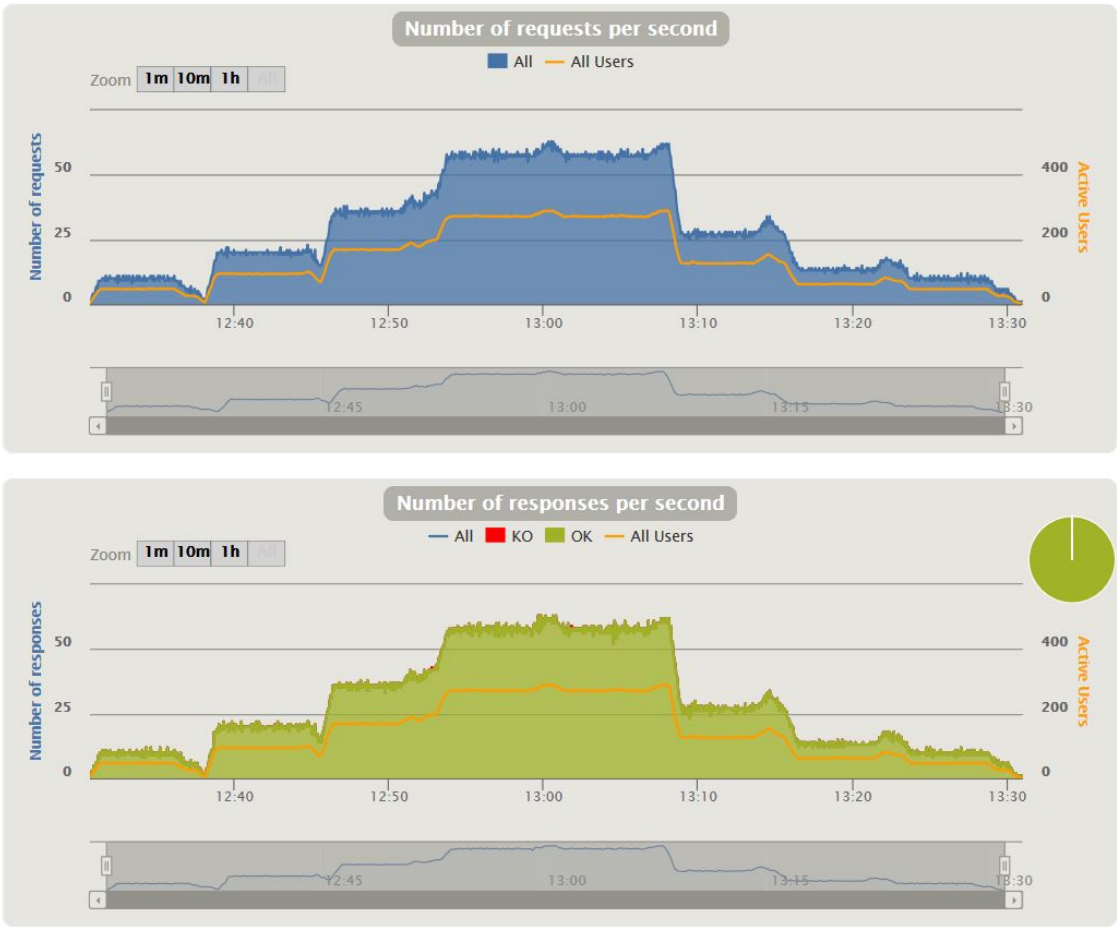

Figure A.4: Number of Requests/Reponses per Seconds and Active Users (Lambda 1024)

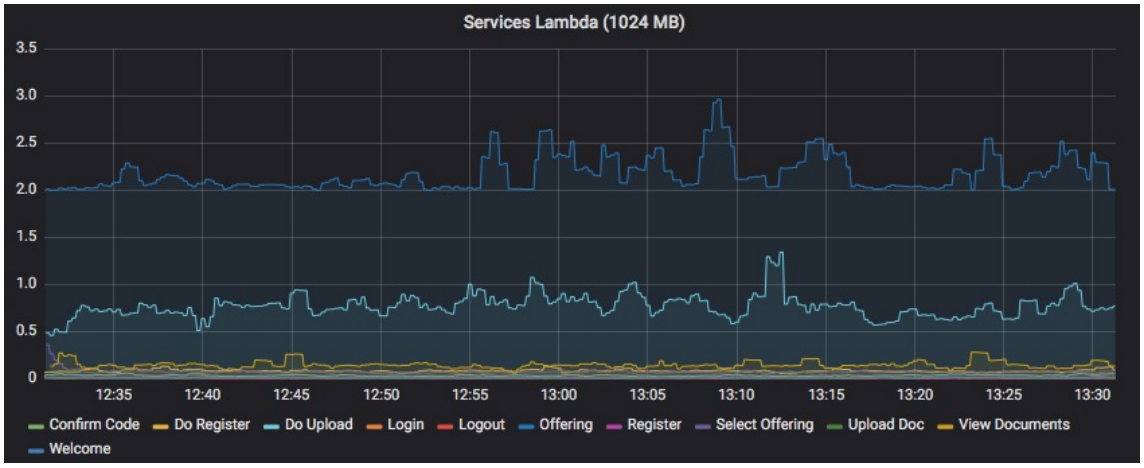

Figure A.5: AWS Lambda (1024 MB) Function Response Times

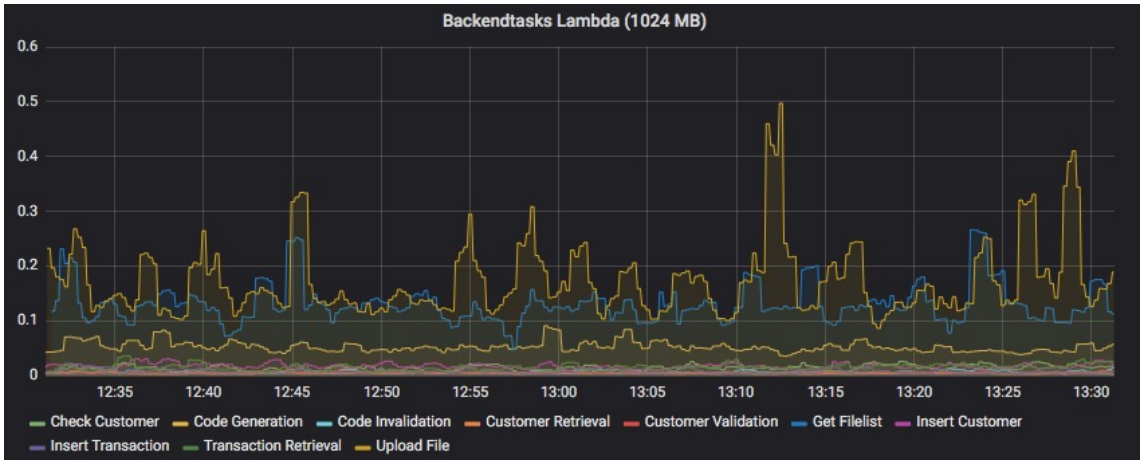

Figure A.6: AWS Lambda (1024 MB) Function Data Access Times

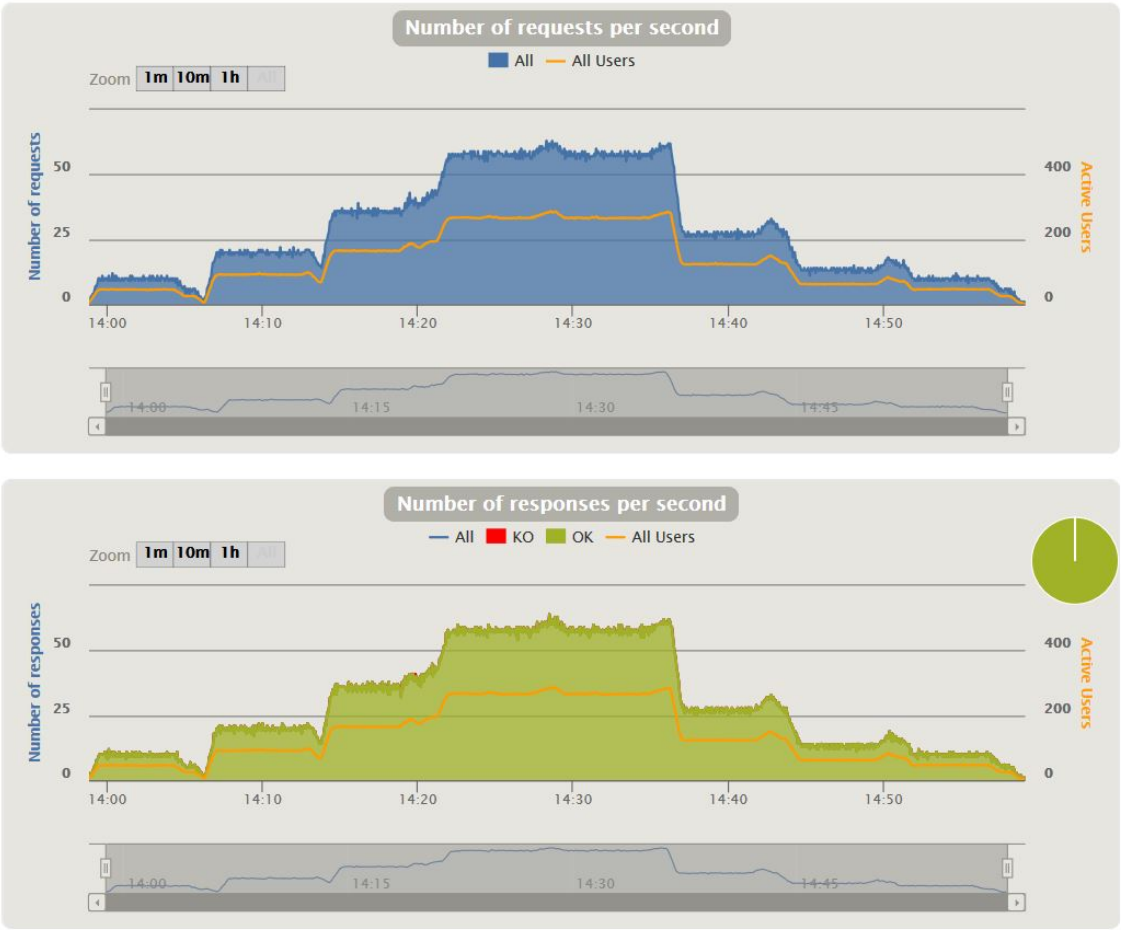

Figure A.7: Number of Requests/Reponses per Seconds and Active Users (Lambda 2048)

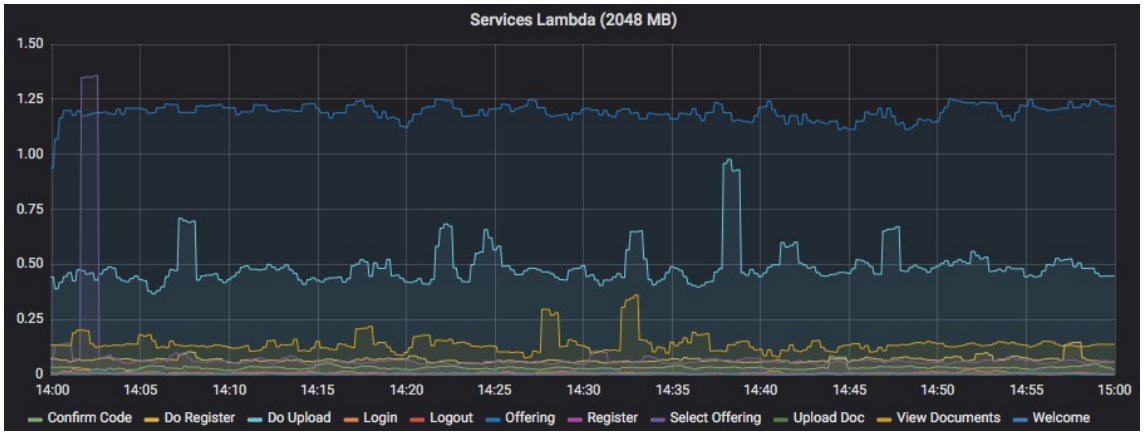

Figure A.8: AWS Lambda (2048 MB) Function Response Times

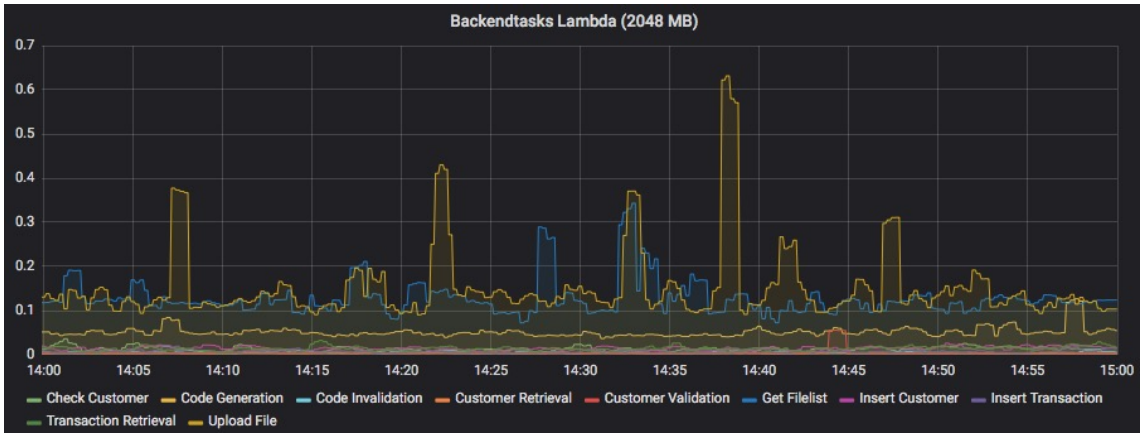

Figure A.9: AWS Lambda (2048 MB) Function Data Access Times

Bibliography

[Adz16] G. Adzic. *Claudia.js - JavaScript cloud micro-services the easy way*. 2016. URL: https://claudiajs.com (cit. on p. 38).

[AFG+09] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, et al. *Above the clouds: A berkeley view of cloud computing*. Tech. rep. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009 (cit. on p. 1).

[AFG10] Armbrust, Fox, Griffith. "A view of cloud computing." In: *Communications of the ACM* 53.4 (Apr. 2010), pp. 50–58 (cit. on pp. 2, 21, 22).

[Ama17] I. Amazon. *AWS SAM Local (Beta) – Build and Test Serverless Applications Locally*. Aug. 11, 2017. URL: https://aws.amazon.com/de/blogs/aws/new-aws-sam-local-beta-build-and-test-serverless-applications-locally/ (cit. on p. 20).

[Ama18] Amazon. *Amazon Web Services (AWS)*. 2018. URL: https://aws.amazon.com/ (cit. on pp. 21, 27).

[Aut17] P. Authors. *Prometheus - From Metrics to insight*. 2017. URL: https://prometheus.io/ (cit. on p. 21).

[Aut18a] Auth0. *Building serverless apps with webtask*. 2018. URL: https://auth0.com/blog/building-serverless-apps-with-webtask/ (cit. on p. 30).

[Aut18b] Auth0. *Webtask Getting Started*. 2018. URL: https://webtask.io/docs/101 (cit. on p. 30).

[Aut18c] Auth0. *Webtask pricing*. 2018. URL: https://auth0.com/extend/pricing (cit. on p. 30).

[Aut18d] Auth0. *What is Auth0 Extend*. 2018. URL: https://auth0.com/extend/docs/#the-value-of-auth0-extend (cit. on p. 31).

[BB16] R. BONCEA, C. BACIVAROV. "A System Architecture for Monitoring the Reliability of IoT." In: *Proceedings of the 15th International Conference on Quality and Dependability*. 2016, pp. 143–150 (cit. on p. 21).

[BCC+17] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, et al. "Serverless Computing: Current Trends and Open Problems." In: *arXiv preprint arXiv:1706.03178* (2017). URL: https://arxiv.org/abs/1706.03178 (cit. on pp. 3, 14–16, 19, 23).

[BHJ15] A. Balalaie, A. Heydarnoori, P. Jamshidi. “Migrating to Cloud-Native Architectures Using Microservices: An Experience Report.” In: *CoRR* abs/1507.08217 (2015). arXiv: 1507.08217. URL: http://arxiv.org/abs/1507.08217 (cit. on p. 8).

[Boy17] M. Boyd. *Security in Serverless: What gehts better, what gets worse?* May 16, 2017. URL: https://thenewstack.io/security-serverless-gets-better-gets-worse/ (cit. on pp. 18, 19).

[BWC18] BWCloud. *bwCloud*. 2018. URL: https://www.bw-cloud.org/de/projekt (cit. on p. 41).

[BYV+08] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, I. Brandic. “Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing As the 5th Utility.” In: vol. 25. 6. Amsterdam, The Netherlands, The Netherlands: Elsevier Science Publishers B. V., June 2008, pp. 599–616. DOI: 10.1016/j.future.2008.12.001. URL: http://dx.doi.org/10.1016/j.future.2008.12.001 (cit. on p. 1).

[CLN12] Chaisiri, Lee, Niyato. “Optimization of Resource Provisioning Cost in Cloud Computing.” In: *IEEE Transactions on Services Computing* 5.2 (Apr. 2012), pp. 164–177 (cit. on p. 2).

[Cui17] Y. Cui. *Running and debugging AWS Lambda functions locally with the Serverless framework and VS Code*. July 27, 2017. URL: https://hackernoon.com/running-and-debugging-aws-lambda-functions-locally-with-the-serverless-framework-and-vs-code-a254e2011010 (cit. on p. 20).

[Eiv17] A. Eivy. “Be Wary of the Economics of "Serverless" Cloud Computing.” In: *IEEE Cloud Computing* 4.2 (Mar. 2017), pp. 6–12. ISSN: 2325-6095. DOI: 10.1109/MCC.2017.32 (cit. on pp. 15, 16).

[Ell18] A. Ellis. *OpenFaaS Project*. 2018. URL: https://www.openfaas.com (cit. on p. 35).

[Fis17] M. Fischer. *Introducing Spring Cloud Function*. July 5, 2017. URL: https://spring.io/blog/2017/07/05/introducing-spring-cloud-function (cit. on p. 34).

[Fis18] Fission. *Fission - Serverless functions for kubernetes*. 2018. URL: https://fission.io/ (cit. on p. 33).

[Fnp18] Fn-project. *Fn project*. 2018. URL: https://fnproject.io/ (cit. on pp. 34, 35).

[Fow14a] M. Fowler. *Microservices*. Mar. 25, 2014. URL: https://martinfowler.com/articles/microservices.html (cit. on pp. 8, 9).

[Fow14b] M. Fowler. *Microservices - a definition of this new architectural term*. https://martinfowler.com/articles/microservices.html. 2014 (cit. on p. 2).

[Fow14c] M. Fowler. *Testing strategies in a Microservice Architecture*. https://martinfowler.com/articles/mic testing/. Nov. 2014 (cit. on p. 19).

[Gat18a] Gatling. *Gatling - HTTP Request*. 2018. URL: https://gatling.io/docs/2.3/http/http_request/ (cit. on p. 38).

[Gat18b] Gatling. *Gatling - Scenarios*. 2018. URL: https://gatling.io/docs/current/general/scenario/ (cit. on p. 38).

[Gat18c] Gatling. *Gatling - Simulation Setup*. 2018. URL: https://gatling.io/docs/current/general/simulation_setup/ (cit. on p. 39).

[Gat18d] Gatling. *Gatling Performance Testing*. 2018. URL: https://gatling.io/performancetesting/ (cit. on pp. 38, 39).

[Goo18] Google. *Google Cloud Functions*. 2018. URL: https://cloud.google.com/functions/ (cit. on pp. 22, 29).

[Gra18] Grafana. *Grafana*. 2018. URL: https://grafana.com/ (cit. on p. 39).

[GV06] Gray, Vogels. "A Conversation with Werner Vogels." In: *QUEUE* (May 2006), pp. 14–21 (cit. on pp. 1, 2, 7).

[Han17] A. Handy. *Pivotal Adds a Serverless Service to its Cloud-native Platform*. Dec. 5, 2017. URL: https://thenewstack.io/pivotal-introduces-serverless-platform/ (cit. on pp. 25, 30).

[IBM18] IBM. *IBM Cloud Functions*. 2018. URL: https://www.ibm.com/cloud/functions (cit. on p. 30).

[inc14] A. inc. *Amazon Web Services Announces AWS Lambda*. Nov. 13, 2014. URL: http://phx.corporate-ir.net/phoenix.zhtml?c=176060&p=irol-newsArticle&ID=1989542 (cit. on p. 25).

[Iro18] Iron.io. *IronFuctions - Open Source Serverless Computing*. 2018. URL: http://open.iron.io (cit. on p. 34).

[Joh17] P. Johnston. *Security and Serverless*. Feb. 7, 2017. URL: https://read.acloud.guru/security-and-serverless-ec52817385c4 (cit. on pp. 18, 19).

[Kub18] Kubeless. *What is Kubeless*. 2018. URL: http://kubeless.io (cit. on p. 33).

[Ley09] F. Leymann. "Cloud Computing: The Next Revolution in IT." In: *Proceedings of the 52th photogrammetric Week*. 2009 (cit. on p. 1).

[Low16] C. Lowery. *Emerging Technoilogy Analysis: Serverless computing and Function Platform as a Service*. Tech. rep. Gartner, Sept. 15, 2016 (cit. on p. 3).

[MB17] G. McGrath, P. R. Brenner. "Serverless Computing: Design, Implementation, and Performance." In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. June 2017, pp. 405–410. DOI: 10.1109/ICDCSW.2017.36 (cit. on pp. 3, 14–16, 21, 22, 25).

[McG17] G. McGrath. "Serverless Computing: Applications, Implementation, and Performance." MA thesis. University Of Notre Dame, 2017 (cit. on pp. 21, 22).

[MG11] P. M. Mell, T. Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Tech. rep. Gaithersburg, MD, United States, 2011 (cit. on pp. 2, 12–14).

[Mic18] Microsoft. *Microsoft Azure Functions*. 2018. URL: https://azure.microsoft.com/en-us/services/functions/ (cit. on p. 28).

[Ope18] A. OpenWhisk. *Getting started with OpenWhisk*. 2018. URL: https://github.com/apache/incubator-openwhisk/tree/master/docs#getting-started-with-openwhisk (cit. on p. 32).

[Piv18a] Pivotal. *Pivotal Function Service*. 2018. URL: https://pivotal.io/platform/pivotal-function-service (cit. on p. 30).

[Piv18b] Pivotal. *Spring Cloud Function*. 2018. URL: https://cloud.spring.io/spring-cloud-function/ (cit. on p. 34).

[PTDL07] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann. "Service-Oriented Computing: State of the Art and Research Challenges." In: *Computer* 40.11 (Nov. 2007), pp. 38–45. ISSN: 0018-9162. DOI: 10.1109/MC.2007.400 (cit. on pp. 1, 2).

[RR16] J. W. Rittinghouse, J. F. Ransome. *Cloud computing: implementation, management, and security*. CRC press, 2016 (cit. on pp. 1, 20).

[Sar16] J. Saryerwinnie. *Chalice - Python Serverless Microframework for AWS*. 2016. URL: https://chalice.readthedocs.io/en/latest/ (cit. on p. 37).

[Ser18] I. Serverless. *Serverless - The way cloud should be*. 2018. URL: https://serverless.com (cit. on pp. 35, 36).

[ser18] I. serverless. *Serverless Framework*. 2018. URL: https://serverless.com/framework/ (cit. on p. 19).

[SHM+14] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, B.-J. Kim. "Performance comparison analysis of linux container and virtual machine for building cloud." In: *Advanced Science and Technology Letters* 66.105-111 (2014), p. 2 (cit. on p. 3).

[SMM18] J. Spillner, C. Mateos, D. A. Monge. "FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC." In: *High Performance Computing: 4th Latin American Conference, CARLA 2017, Buenos Aires, Argentina, and Colonia del Sacramento, Uruguay, September 20-22, 2017, Revised Selected Papers*. Ed. by E. Mocskos, S. Nesmachnow. Cham: Springer International Publishing, 2018, pp. 154–168. ISBN: 978-3-319-73353-1. DOI: 10.1007/978-3-319-73353-1_11 (cit. on pp. 14, 15).

[Spo18] Spotinst. *Spotinst Functions - The real cloud. No servers. Just code*. 2018. URL: https://spotinst.com/products/spotinst-functions/ (cit. on p. 31).

[SRT15] Savchenko, Radchenko, Taipale. "Microservoces validation: Mjolnirr platform case study." In: IEEE COMPUTER SOCIETY, 2015. DOI: 10.1109/MIPRO.2015.7160271 (cit. on p. 19).

[Tho17] ThoughtWorks. *Serverless architecture*. 2017. URL: https://www.thoughtworks.com/radar/techniques/serverless-architecture (cit. on p. 3).

[VGC+15] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, S. Gil. “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud.” In: *2015 10th Computing Colombian Conference (10CCC)*. Sept. 2015, pp. 583–590. DOI: 10.1109/ColumbianCC.2015.7333476 (cit. on pp. 2, 7, 8).

[VGO+16] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, M. Lang. “Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures.” In: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. May 2016, pp. 179–182. DOI: 10.1109/CCGrid.2016.37 (cit. on pp. 9–11).

[WS16] B. Wagner, A. Sood. “Economics of Resilient Cloud Services.” In: *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. Aug. 2016, pp. 368–374. DOI: 10.1109/QRS-C.2016.56 (cit. on pp. 3, 16, 18).

[XWQ16] Z. Xiao, I. Wijegunaratne, X. Qiang. “Reflections on SOA and Microservices.” In: *2016 4th International Conference on Enterprise Systems (ES)*. Nov. 2016, pp. 60–67. DOI: 10.1109/ES.2016.14 (cit. on p. 8).

[YCCI16] M. Yan, P. Castro, P. Cheng, V. Ishakian. “Building a Chatbot with Serverless Computing.” In: *Proceedings of the 1st International Workshop on Mashups of Things and APIs*. MOTA ’16. Trento, Italy: ACM, 2016, 5:1–5:4. ISBN: 978-1-4503-4669-6. DOI: 10.1145/3007203.3007217. URL: http://doi.acm.org/10.1145/3007203.3007217 (cit. on p. 20).

[Zap] Zappa. *Zappa - Serverless Python Web Services - Powered by AWS Lambda and API Gateway*. URL: https://www.zappa.io (cit. on p. 38).

[ZW+13] Zhan, Wang, et al. “Cost-Aware Cooperative Resource Provisioning for Heterogeneous Workloads in Data Centers.” In: *IEEE Transactions on Computers* 62.11 (Nov. 2013), pp. 2155–2168 (cit. on p. 1).

All links were last followed on March 27, 2018.

Acknowledgement

I would like to thank my supervisors Michael Wurster (University of Stuttgart) as well as Klaus Brenner and Philipp Roßberger (Allianz Deutschland AG) for their support and guidance during my master thesis. I also thank my family and friends especially Carl Lämmle for their help and moral support.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, 04.04.18, _____

place, date, signature (Christopher Völker)