Daniel Schoberegger, BSc

Evolutionary Development of WEB-based Information Systems

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Software Development and Business Management

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl-Ing. Dr.techn. Nikolai Scerbakov

Institute of Interactive Systems and Data Science
Head: Univ.-Prof. Dipl-Inf. Dr. Stefanie Lindstaedt

Graz, September 2018

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Abstract

The development of a modern web-based information system is hard. Especially for startups with a limited budget. A startup thinks visionary and plans usually a web-based information system for millions of concurrent active users. A web-based information system runs on multiple server instances at a cloud provider. The cloud provider establishes the necessary infrastructure and charges computing resources. The problem is that a cloud provider also charges idle servers instances.

The purpose of this thesis is to find an evolutionary development of web-based information systems in regards to avoid idle server instances and reduce infrastructure costs.

The result is an evolutionary development environment based on the ”Unix philosophy” and extended with the best practices ”Twelve-Factor App” and ”jHipster”. An intensive literature research compares the architectural pattern monolithic, microservice and serverless. In addition, an industrial case study of the command device ’Amazon Alexa’ is implemented in each architectural pattern. Common pitfalls and problems are shown in the process of developing a web-based information system. The architectural pattern monolith, microservice and serverless is discussed in regards to the impact on infrastructure costs. Especially their application characteristics high availability and scalability have a different impact on infrastructure costs. The case study shows that the evolutionary environment with a serverless architectural pattern can reduce infrastructure costs by 30 percent. The results of the thesis change the economy of hosting. Therefore, a startup has only to pay for actual utilization instead of reserved capacity and idle server instances.

Contents

List of Figures

List of Figures

1 Introduction

Consider you as an entrepreneur start your own company and you launch a new web-based information system worldwide. You think visionary and you plan your web application for millions of concurrent active users. You run your application on server instances at a cloud provider. The cloud provider charges you for this infrastructure costs. As a startup, you have a limited budget. For example, in the beginning, you do not have active users on your application and so you do not generate revenue. But the cloud provider still charges infrastructure costs. The problem is that you have to pay these fix costs, even if you have no active users on your application.

The main purpose of this thesis is to find an evolutionary development environment for a modern web-based information system.

1.1 Problem Statement

I am working on architectural patterns for web applications because I want to find out how fix infrastructure costs can be reduced in order to help startups to work longer with their limited budget.

1.2 Motivation

A cloud provider charges costs even for idle server instances. My motivation is to raise awareness for fix infrastructure costs and help startups to avoid them. In addition, I provide a shared vocabulary for discussing this problem. I use technology on its best edge and I do not accept why I have to pay for something that I do not use.

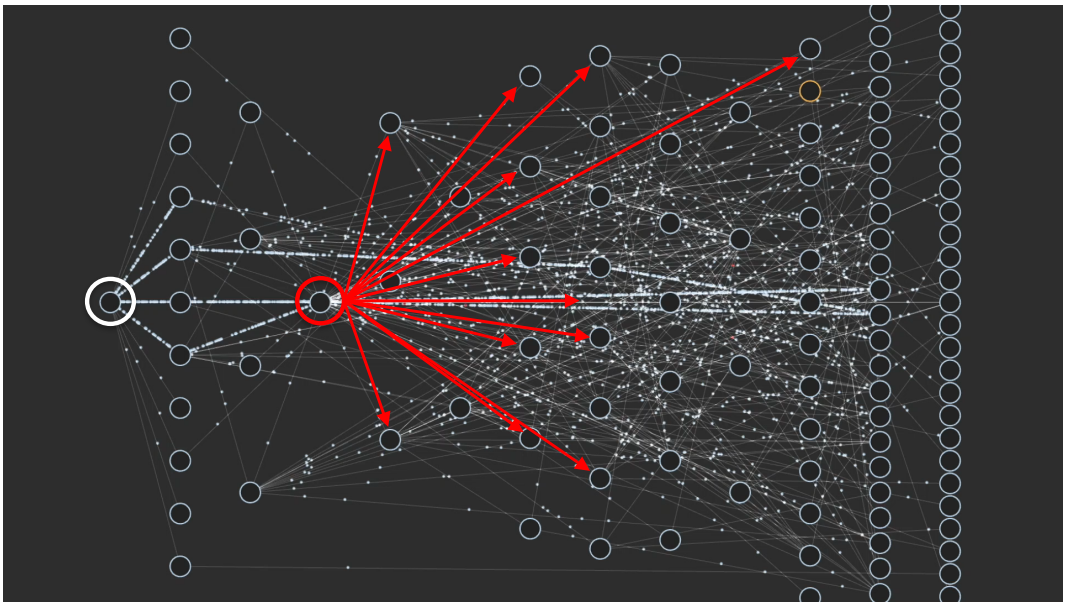

Figure 1.1: Mastering Chaos - Source: Evans (2016)

I believe that you can reduce infrastructure costs by tackling software challenges on the level of architecture most efficiently. If you choose the correct architectural pattern, you can save infrastructure costs. Therefore, I discuss the three architectural patterns: **monolithic**, **microservice** and **serverless**.

The lead developer Josh Evans at Netflix inspired me with an impressive presentation on the InfoQ Conference: Evans (2016). The presentation is called "Mastering Chaos - A Netflix Guide to Microservices". In this presentation he came up with the figure 1.1.

The figure 1.1 shows an entire web application. Each circle represents an individual service, which offers functionality to the end user. The relationships between these services are shown by the arrows. As you can see in this figure, a modern application consists of many services and they correlate each other. If you do not want to end up in chaos with high infrastructure costs, you have to choose the correct architectural pattern. As Evans (2016) mentioned in his talk, you have to embrace the tension between order and chaos. It motivates me to find a healthy mix of well-proved disciplines and

methodologies. I call this mix in this thesis as **an evolutionary development of web-based information systems**.

As a software engineer in the last five years, I learned myself that every developed solution was born out of pain for me and my customers. I am very vigilant to set up the best architecture for web application and evolving them continuously. The knowledge about architectural patterns helps to find the best solution in a chaotic and vibrant world of web applications.

1.3 Definition

This thesis sees a **web-based information system** as a **web application** with following characteristics:

- involves persistent data
- data accessed concurrently
- integrate with other web applications
- using different interfaces and offers HTTP endpoints

The term web-based information system and web application is used equally and stands for the same meaning in this thesis. The short version for application is ”app”, which is the same thing as an application or web-based information system.

This thesis sees **infrastructure costs** as the costs of all server instances.

A server instance is a unit where the web application runs on it. The costs are charged and determined by the cloud provider. This paper analyzes the pricing of the cloud provider Amazon AWS. The charges of a server instance vary from the computing resource of a server instance. In this paper computing resources are memory processor, memory, storage, and network capacity. Infrastructure costs can be only analyzed if you know the software requirements. For example, if you know that you have one request per hour for your application then you do not need a powerful and expensive infrastructure. Therefore this paper defines requirements for a case study application and the same application is implemented in three different **architectural pattern**: monolithic, microservice and serverless.

1.3.1 Case Study - Functional Requirements

The following requirements define the demonstration application. The demonstration application is used to determine the infrastructure costs at the cloud provider.

1. **New Alexa skill in the Amazon skill store:** The demonstration application is a new Alexa skill and is available in the skill store worldwide. Everyone with a valid Amazon Alexa account can add the skill on his own Alexa device.
2. **Web application:** The Alexa skill itself is a web application and offers HTTP endpoints to communicate with users and Alexa devices.
3. **Persistent data in PostgreSQL:** The text description of a cryptocurrency fact is stored in a relational PostgreSQL database. In total 20 facts.
4. **Communication format is JSON:** To each user request a random fact will be chosen and sent it back to the user's device via JSON.
5. **Persistent data in MySQL:** Newer Alexa skill devices can display an image as well. Therefore each cryptocurrency fact contains one image. The URL to the image is stored in a MySQL database. In total 20 links.
6. **Images on file storage:** The image itself is stored on the file system and each image is accessible via HTTPS Endpoint.
7. **High availability:** The web application has to be redundant and no single point of failure exist. Failure of one service does not mean the failure of the entire web application (high availability).
8. **SSL certificates:** HTTPS endpoints are secured with SSL (HTTPS).
9. **Response time:** The maximum response time is 3000 ms for each fact request.
10. **Latency time:** The maximum latency time is 2000 ms for each fact request.
11. **Throughput:** The maximum throughput is 10 fact requests per second.
12. **Scalability:** The web application scales up intermediately. From zero requests per second to the maximum of 10 requests per second. This happens with no communication errors and each request has to be under the mentioned response time (3000 ms) and latency time (2000 ms).

1.3.2 Case Study - Non-Functional Requirement

1. **Concurrent device failures:** Services are not designed to sustain concurrent device failures. As I mentioned in the requirements the design can handle single point of failures. This does not mean that the system can sustain concurrent device failures. Due to the requirements, the service runs redundant on two different server instances (requirement number 7). Each service runs redundant on two server instances. Each server instance is located in a physically separated availability zone to eliminate the single point of failure. In the worst case both server instances can fail concurrently and then the service fails as well.

As you can see the demonstration application is defined by functional and non-functional requirements. The demo application delivers the business value retrieving cryptocurrency facts via a new Alexa skill. The mentioned requirements have an impact on server infrastructure costs and will be analyzed in this paper.

1.3.3 Tasks

The main purpose of this thesis is to find an evolutionary development environment for a modern web-based information system.

The thesis consists of the following further tasks:

- I prove that the 'Unix philosophy' is the basis for a modern development environment (2.8.1).
- I confirm that an evolutionary development environment follows the best practice guidelines "Twelve-Factor-App" and "jHipster" 2.8.5.
- I come up with a clear definition for architectural pattern 2.2.1.
- I prove that the architectural patterns monolithic, microservice and serverless can be described via pattern description format. The pattern description format consists of a context, problem and solution 2.2.4.
- I build an industrial case study for the Amazon voice control device Alexa. This application is used to determine the infrastructure costs on the cloud provider Amazon AWS. The result shows that the explained

evolutionary development environment with a serverless architectural pattern reduces infrastructure costs by 29.47 percent (Section 4).

- I define the case study with specific requirements. I prove that not all requirements influence the server infrastructure cost. The result is that high availability mostly influences the infrastructure costs (Section 2.3).
- I implement the case study in three independent application architectures: monolithic, microservice and serverless. The result is that the same business value is delivered in three different architectural pattern. I prove that all three applications fulfill the requirements, but each architecture ends up with different infrastructure costs (Section 4).
- I come up with a clear definition for monolithic, microservice and serverless architectural pattern.
  Currently, the research community uses a quite range of unclear definitions (Section 2).
- I prove that infrastructure costs consist of fix and variable costs. The result is that monolithic and microservice architecture have the same fix costs. The serverless architecture has the lowest fix costs (Section 4.5.3).
- I prove that all three application can sustain a performance test. During a throughput of 10 requests per second, the response time is lower than 3000ms and the latency is lower than 2000ms (Section 4.7).
- I prove that a monolithic application does not scale efficiently like a microservice or serverless architecture. The result is high infrastructure costs for a monolithic application (Section 4.3).
- I prove that a monolithic application gains high availability easily. The result is a duplication of the entire application and this doubled the infrastructure costs.
- I prove that the case study application requirements can be decomposed into independent services. This result is 5 independent services (Section 4.3.1).
- I prove that all three applications have no single point of failure. The result is higher infrastructure costs to gain this requirement (Section 4.4).
- I prove that all the microservice architecture is a more agile approach. The result is a finer granularity of scalability and this reduces the

infrastructure costs by 4.72% in comparison to the monolithic approach (Section 4.4).

2 Fundamentals and Related Work

The purpose of this thesis is to analyze web application environments and their architectural patterns because I want to find out how fix infrastructure costs can be reduced in order to help startups to work longer with their limited budget.

First of all, I explain what do I mean by the term web application in this thesis.

2.1 Web Application

It is hard to come up with a precise definition of the term web application because it would not clear enough to express this term. It is better to speak about common characteristics of how I understand the term modern web application. The author Fowler (2002) argues in his book "Patterns of Enterprise Application" in the same way. He cannot give a definition, but he comes up with the following characteristics of an application. I adapt these characteristics of how this thesis sees a web application.

Web application involves **persistent data**. The data is persistent because data needs to be around between multiple runs of the application. This thesis deals with persistent data in the context of the fact description and the corresponding image URL.

There is **usually a lot of data**. A modern application has over 1TB of data organized in tens of millions for records. The demo application handles much less data because data management is not the goal of this thesis. For example, the demo application has in total 20 fact records in a relational database. This results in a few KB of used storage.

Many user access **data concurrently**. For example, the demo application can handle a throughput of 10 requests per seconds. This means 10 users access data at the same time and this should not cause errors.

A web application has **different interfaces** and exposes data via HTTP protocol. For example, the demo application sends data via JSON format over the HTTP protocol to the voice control device Alexa.

Web applications **integrate with other web applications**.

Of course, there may be further characteristics of a modern web application but these mentioned characteristics are sufficient for further discussion in this thesis. This thesis defines a web application with the characteristics:

- persistent data
- huge dataset
- concurrent data
- different interfaces
- integration with other web applications

A web application consists of different components. And how these components work together are explained with the term architectural pattern. In the next step, I explain how this thesis sees the term architectural pattern.

2.2 Architectural Pattern

Scerbakov (2018) explains in his lecture notes "Internet-Based Information Systems": A web application is based on the client-server architecture. A client sends an HTTP request, and the server returns an HTTP response. The demo application is based on this mentioned client-server architecture. For example, the Alexa device sends an HTTP request as JSON data to the server, the server process the request and sends back an HTTP response as JSON data with a fact description and image URL.

This thesis sees the client-server architecture in more finer granularity. I introduce the three architectural pattern: **monolithic**, **microservice** and **serverless**. In the entire thesis, the term application architecture always relates to one of these three approaches: monolithic, microservice or serverless.

It does not mean that the client-server architecture is not valid anymore. A monolithic, microservice and serverless web application is still based on a client-server architecture. The difference is this: A monolithic, a microservice and a serverless architecture influences infrastructure costs in a different way. The main purpose of this thesis is to analyzes these influences. The term client-server architecture does not influence infrastructure costs on that fine granular level because the term is valid for all three architectural patterns without any distinguishes.

2.2.1 Pattern Definition

Kamal and Avgeriou (2010) defined in their research work an architectural pattern as follows: "Architectural patterns provide proven solutions to recurring design problems that arise in a system context. They specify guidelines for designing the structural and behavioral aspects of a system. An architectural pattern details a fundamental solution to a design problem in the form of pre-defined pattern participants like pattern-specific components, classes, or objects that work together to resolve the identified problem."

Similarly, Buschmann et al. (1996) defined the term: "An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them."

In summary, a web application consists of multiple components. And the architectural pattern describes how the components work together.

The term architectural pattern is a subcategory of the common term pattern. In the following section, I define the common term pattern.

As Tešanović (2001) described in her report, that the common term pattern as a widely discussed term between software engineers. But patterns are an effective way to communicate with software developers. Patterns bring order in a chaotic life of the software development process. Because pattern represent best practices, proven solutions and lessons learned to the engineering discipline of software development. Therefore, patterns improve

the software development process. Every software developer should be comfortable to analyze and use a different pattern when designing and implementing a web-based information system. Experienced engineers have extensive knowledge about a large set of patterns.

The term pattern originally was introduced and defined by the architect Alexander (1979): "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

And later the term pattern was adopted to software engineering. In software design the term pattern appeared in the late 1980s when Beck (1998) developed a set for patterns for developing elegant user interfaces in the programming language Smalltalk. At around the same time, the researcher Coplien (1992) was creating a catalog of pattern in the programming language C++ and called them idioms.

Beck et al. (1996) said that "a pattern is a particular form of recording design information such that design which has worked well in particular situations can be applied again in similar situations in the future by others. Patterns are grouped in categories. In the next section, I describe these categories.

2.2.2 Pattern Categories

The researcher Buschmann et al. (1996) divided the common term pattern into three categories:

- architectural pattern
- design pattern
- idioms

As I already mentioned in section 2.2.1, these categories are subcategories of the common term pattern.

Tešanović (2001) analyzed in her work the term pattern and proved that the difference between these three categories is in their level of abstraction and detail.

Buschmann et al. (1996) acknowledged the difference in the granularity of abstraction. Architectural patterns are high-level strategies that deal with global properties and mechanism of a system. An architectural pattern is a fundamental design decision and has an impact on the entire software system. A design pattern has a finer granularity. In contrast to an architectural pattern, a design pattern define microarchitectures of subsystems and components. Design pattern does not influence the entire software system. Design patterns are smaller in scale than architectural patterns. A design pattern has no impact on the fundamental structure of a software system. The finest granularity of abstractions are idioms. Idioms deal with the implementation of particular design issues. They are specific for a programming language or programming technique.

Buschmann et al. (1996) define architectural pattern, design pattern and idioms as follows: "An **architectural pattern** expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them."

"A **design pattern** provides a scheme for refining the subsystem or components of a software system, or the relationship between them. It describes a commonly-recurring structure of communication components that solves a general design problem within a particular context."

"An **idiom** is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language."

In summary, the three mentioned pattern categories abstract a software system on different levels. An architectural pattern is a **high-level** abstraction of the entire system. A design pattern is a **medium-level** abstraction of a specific component. Finally, an idiom is a **low-level** abstraction in a specific programming language.

Of course, the placing of a pattern in one of the three categories can be difficult, because several patterns could be placed into at least two categories. This problem the researcher (Tichy, 1997) mentioned in his work. It is not always possible to place one pattern into exactly one category. For example,

the more problem areas a pattern includes, the more likely a pattern is to be placed in multiple categories.

Therefore, Tichy introduced a catalogue with following categories:

- **Decoupling:** "dividing a software system into independent parts in such a way that the parts can be built, changed, replaced, and reused independently."
- **Variant Management:** "treating different objects uniformly by factoring out their commonality."
- **State Handling:** "generic manipulation of object state."
- **Control:** "control of execution and method selection."
- **Virtual Machines:** "simulated processors."
- **Convenience Patterns:** "simplified coding."
- **Compound Patterns:** "patterns composed from others, with the original patterns visible."
- **Concurrency:** "controlling parallel and concurrent execution."
- **Distribution:** "problems germane to distributed systems."

These mentioned categories are more precise and mutually exclusive.

But this thesis focuses on the three patterns: monolithic, microservice and serverless. And these three patterns can be placed precisely into the group of architectural patterns, defined by Buschmann et al.

A modern development of a web-based information system combines architectural, design and idiom patterns. As Buschmann et al. (1996) mentioned, pattern categories help to pre-select potentially useful patterns for a given design problem. Rech and Ras (2011) analyzed, that skilled software engineers have a good knowledge of patterns and their usage. And skilled software engineers often reuse fundamental existing pattern and apply established processes to construct complex web-based information systems. Rech and Ras (2011) says, "without the reuse of well-proven knowledge, e.g., in the form of software patterns, we would have to rebuild and relearn it again and again." The already mentioned pattern categories help to classify patterns. In the next section, I discuss the pattern integration in an evolutionary development process of an web-based information system.

2.2.3 Pattern Integration in a Evolutionary Development Approach

A modern development uses architectural, design and idiom patterns. Experienced software engineers have a broad knowledge of theoretical facts of patterns and their usage. But it is hard to identify the correct pattern of the huge amount of all existing pattern. Also, the already existing experience of each engineer plays a role in choosing a pattern. Each engineer has its own favorite pattern and their bad or good experiences influence the choosing process. As Buschmann et al. (1996) say, "patterns help you build on the collective experience of skilled software engineers. They capture the existing, well-proven experience in software development and help to promote good design practise. Every pattern deals with a specific, recurring problem in the design or implementation of a software system. Patterns can be used to construct software architectures with specific properties".

An architectural pattern can be used at the beginning of the planning process of the entire application. Design patterns are used during the design phase of specific components. And idioms are used during the implementation phase in a concrete programming language. For example, the case study shows such a design problem. The requirements in section 1.3.1 define the design problem. But using the correct pattern is a highly discussed problem in the computer science research community. Ras, Rech, and Weber (2009) summarized in their research that many software projects are developed under tight deadlines and with changing stakeholder requirements. Therefore, the process for evaluating, discussing and implementing a pattern is very limited. As I already mentioned, pattern categories help to group patterns. But you still need a concept of how you can describe a specific pattern. Therefore, Buschmann et al. (2000) explained a pattern description format. The format consists of three sections: context, problem, and solution. This format follows the pattern form described originally by the author Buschmann et al. (1996). In the next chapter, I discuss this specific pattern description format.

2.2.4 Pattern Description Format

Buschmann et al. (2000) describe each pattern with three sections: a corresponding context, problem, and solution. In this thesis, I use the same pattern format to describe the monolithic, microservice and serverless architectural pattern.

- **Context:** Expressed as generally as possible, to avoid limiting their applicability to a particular configuration.
- **Problem:** Whenever such a problem arises the pattern can be applied.
- **Solution:** Present more detailed information about how the pattern works.

This described form for patterns is self-contained and allows to present the essence and the key details of a pattern. It gives an overview of the pattern's fundamental ideas, as well as how the patterns work in depth for engineers who want to know all the details. It helps to formalize pattern.

Researches and software engineers write out the three sections, context, problem, and solution, for each pattern. And then they keep them in a catalog. In addition, if someone uses a pattern, he or she gains experiences with a concrete pattern. The author Rech and Ras (2011) analyzed in his work this aggregation of experiences. He used experience factories and established management techniques to summarize and preserve valuable knowledge from an old software project. The results are new software patterns. Buschmann et al. (1996) confirmed that the most successful patterns will be created bottom-up, by generalizing from the collective experience of expert engineers and software architects. "This inductive process has a better chance of success that approaches that try to define pattern-oriented methods and principles top-down".

So far we know a description format for patterns. This form consists of three sections: context, problem, and solution. But researchers still trying to find a more general and formal format. Therefore, they try to form a pattern description language.

2.2.5 Pattern Description Language

Researchers try to go from a pattern description format, explained the previous chapter, towards to a general pattern description language.

Why a pattern language description language is important?

As I mentioned in section 2.2.2, patterns belong to categories. And each category has a different level of abstraction. "An architectural pattern introduce a structure and defines the base architecture of an entire system. Each component in such an entire system is complex by itself, and often these components can be implemented using other design patterns. Therefore, it is important to express the relationship between such patterns to determine which pattern to apply first and which later" (Buschmann et al., 2000). Adding relationship means connecting patterns. Patterns should strive to connect patterns that can complement and complete each other (Alexander, 1977).

This thesis deals with three architectural patterns: monolithic, microservice and serverless. And all three patterns belong exactly to one category, called architectural pattern. In this thesis, I do not deal with design patterns and idioms. Therefore, I do not have to analyze which pattern comes first. An architectural pattern comes always before a design pattern or an idiom. In conclusion, you can use either a monolithic pattern or a microservice pattern or a serverless pattern at the beginning. An architectural pattern defines the basic structure of the entire system. Therefore this thesis does not identify and analyze relationships between patterns. And the thesis does not define a pattern ordering based on the relationships. But the essential goal of a pattern description language is to find a way how different pattern correlates with each other and how you can describe these relationships in a formal way. These relationships also help to find what pattern comes first in a software development process. What comes first means, what is an underlying pattern and has to be defined at the beginning of the design process. As I mentioned in chapter 2.2.3, an architectural pattern is an underlying pattern and has to be defined at the beginning of the design. An architectural pattern is the basis of the entire system. Design pattern and idioms are more granular and are applied for individual components of the entire system. Therefore, design pattern and idioms come after architectural

pattern. As I mentioned, this thesis only analyses the architectural patterns and therefore the relationship and order are not a part of this thesis. The architectural pattern always comes first. First means at the beginning of the planning process of a web-based information system. Later means at the time of implementing a specific component of the system.

In conclusion, a pattern description language gives more information than a pattern description format. More information means additional details about their relationships with each other. A pattern description language also helps to order pattern. Therefore, you can say what pattern comes at the beginning during the planning process. And what pattern comes later during the concrete implementation phase. This thesis deals with architectural pattern and as you can see an architectural pattern always comes first: At the beginning of the planning process of the entire web-based information system. Therefore the pattern description format is sufficient to describe the patterns monolithic, microservice and serverless in this thesis. I do not use a pattern description language.

In the next chapter, I clarify common misconceptions about patterns.

2.2.6 Common Pattern Misconceptions

A common misconception about patterns can be summarized as follows (Tešanović, 2001):

- patterns are only object-oriented
- patterns provide only one solution
- patterns are implementations
- every solution is a pattern

Patterns are not only reduced to object-oriented programming languages. Of course, many patterns are implemented in an object-oriented programming language. But patterns also exist in other programming concepts, for example, functional programming. As I already mentioned in section 2.2.2, if you are on the level of programming languages, then you deal with patterns on a very fine granularity. They are called low-level pattern and they are specific to a programming language and language concept. These patterns belong

to the pattern category idioms. In this thesis, I only explore architectural patterns. Therefore, the distinguish in different programming languages are not necessary. As Beck et al. (1996) acknowledged, patterns can be found in a variety of software systems, independently of the programming concept used in developing those systems.

Patterns provide more than one solution. A common misconception is that a pattern provides exact one solution for the problem. A pattern describes a solution to the reducing problems. It is rather a collection of more than one solution than exactly one solution. As (Tešanović, 2001) confirmed: "a pattern is not an exact implementation: a pattern may provide hints about potential implementation issues. The pattern only describes when, why and how one could create an implementation".

Not every solution, algorithm or heuristic can be viewed as a pattern, said (Tešanović, 2001) in her work: "In order to be considered as a pattern, the solution must be verified as a recurring solution to a recurring problem." The identification of a recurring problem is done by the pattern description format, mentioned in section 2.2.4. The context is the design situation that raises a design problem. The problem is a set of forces occurring in that context. A solution is a form of rule that can be applied to resolve the problem.

In summary, patterns are not only object-oriented, they are also not only one implementation to solve a problem and not every solution is a pattern.

2.3 Infrastructure Cost

A web application runs on a server instance at the cloud provider. Each server instance allocates computing resources (Chhabra and Dixit, 2015). In this thesis compute resources refers to a processor, memory, storage, and network capacity. When computing resources are allocated, the cloud provider charges them. In this thesis, I call this costs infrastructure costs. When a cloud provider runs your application this process is also called hosting (Spillner, 2017). As investigated by Marathe et al. (2014), a cloud provider separates costs into fix and variable cost. In this thesis, fix cost

relates to processor, memory, and storage Variable costs dependents on the amount of user request. The thesis sees the network capacity as variable costs.

The main problem is that the cloud provider charges fix costs during the entire time that the application is hosted, regardless of whether the application was used. The cloud provider does not take idle time into account. Idle time means that there are no requests or the application is not running.

Koomey Jonathan (2015) investigated idle server instances at cloud providers. The core findings of the study were that 30 percent of server instances were idle. These instances have not delivered information or computing capacity in six months or more.

The findings support previous research performed by the Uptime Institute[1], which also found that around 30 percent of servers are unused.

Kaplan James M. (2008) from McKinsey and Company analyzed the utilization of servers. The core findings of the study were that no more than 6 percent of the examined server instances reach their maximum computing output in one year.

As you can see a cloud provider charges you for unused compute capacity. So far we know that around 30 percent of server instances is idle and does not offer any value. The main purpose of this thesis is to eliminate idle server instances.

This thesis monetizes the costs for idle servers and so we get a fixed amount of infrastructure costs. A monetizing of costs is only possible when we know the requirements and how the requirements influence the costs. When your web application is accessed by only a few users per second then you do not need a powerful infrastructure. For example, the number of concurrent users is defined by performance metrics.

Therefore, we have to analyze the defined requirements in 1.3.1.

The requirements define a modern demo web application. This thesis decomposes the requirements into the following cost factors. A cost factor establishes infrastructure costs.

[1]https://uptimeinstitute.com/resources/asset/comatose-server-savings-calculator

Cost Factor 1: Amount of stored and transferred data. The application has persistent data. First, the facts description and corresponding image URL is stored in a relational database. Second, the image itself is stored in a file system storage.

The number of records in the database and the number of files influence infrastructure costs.

Cost Factor 2: High availability. The application is designed to sustain the failure of a server instance. We call this high availability. Failure of a server instance does not follow in a failure of the entire application. If one failure would lead to failure of the entire application, a single point of failure exists. The application is designed to have no single point of failure. High availability is established when you spread your application on multiple server instances and let them run redundantly. As proposed by Len Bass (2012) a load balancer is required to support availability. A load balancer distributes each user request across these multiple server instances. The entry point of your application is only the load balancer. In case of a server instance failure the load balancer recognize the failure and does not redirect any requests anymore to this failed instance. The same application runs redundantly on another server instance and the load balancer redirects all user requests to only correct working server instances. We call a redundant application on another server instance also a replica in this thesis. All server instances together behind a load balancer are called a cluster.

In this thesis, a load balancer is maintained by the cloud provider. This means there is no additional effort to set up a load balancer from scratch. But the cloud provider also charges you infrastructure costs for the load balancer.

In summary, high availability needs a load balancer and additional server instances. Therefore, high availability is a cost factor for infrastructure costs.

Cost Factor 3: Scalability As I mentioned in the previous section, high availability means adding multiple server instances behind a load balancer. Adding new server instances results in allocating more computing capacity. Basically, we enlarge the entire application. We call this enlarging of application scalability.

This thesis distinguishes between high flexibility and low flexibility for scaling. When you replicate the entire application behind a load balancer we call this low flexibility for scaling. If you only duplicate parts of your application then we speak about more flexibility in regarding scalability. We also use the synonym 'granularity of scalability' for the meaning 'flexibility of scaling'.

Higher flexibility of scaling concludes in a higher amount of server instances and this finally impacts the infrastructure costs.

Cost Factor 4: Performance An increase in computing capacity also contributes positively to the overall performance of your application. For example with additional server instances your application can handle more user request per second than with few server instances. López and Spillner (2017) confirmed a directly proportional relationship between the number of server instances and the performance.

According to the defined requirements of the demo application in 1.3.1, the application has to handle 10 user requests per second without any communication errors. Therefore, we have to find a suitable number of server instances to reach this performance metric.

The mentioned 10 user requests per seconds are called throughput. Additional performance metrics in this thesis are response time and latency. Response time is the elapsed time from just before sending the request to just after the **last response** has been received. Latency is the time from just before sending the request to just after the **first response** has been received[2].

In summary, the performance metrics influence the total amount of server instances. The total count of server instances is responsible for the infrastructure costs.

This section list the cost factor that influences the infrastructure costs. As you can see the thesis investigates four cost factors:

- Amount of data
- High availability

[2]https://jmeter.apache.org/usermanual/glossary.html

- Scalability
- Performance

The demo application is defined by its requirements in 1.3.1. These requirements lead to cost factors of the infrastructure. As you remember the goal of this thesis is to reduce infrastructure costs. Now we have to find leverages where we can optimize these cost factors. The easiest way would be to cancel requirements. For example, high availability is not necessary anymore and so you do not need replicas and so you decrease the total amount of server instance. This approach is not an option in this thesis because we do not want to reduce the functionalities of the demo application.

I follow the idea to investigate in different application architectures. I implement the application from scratch in three different architectures: monolithic, microservice and serverless. Each architecture has an impact on the previously mentioned cost factors. Therefore, choosing different application architectures gives us leverage to optimize infrastructure costs.

Villamizar et al. (2017) further pointed out, that a serverless architecture reduces infrastructure costs by up to 77.08%. He examined the effects in laboratory experiments.

Adzic and Chatley (2017) presented in his paper two industrial case studies how migrating an application to a serverless deployment reduced hosting cost by between 66% and 95%.

Carrasco, Bladel, and Demeyer (2018) confirmed that more and more companies adopt the microservice architecture, but he did not monetize the costs.

Here in this thesis, we discuss the three architecture on a real business application. We investigate the mentioned cost factors and monetize the costs for each architecture.

2.4 Monolithic Architectural Pattern

The term comes from the Unix community and appears the first in the book 'The Art of Unix Programming' by Raymond (2003). The author describes

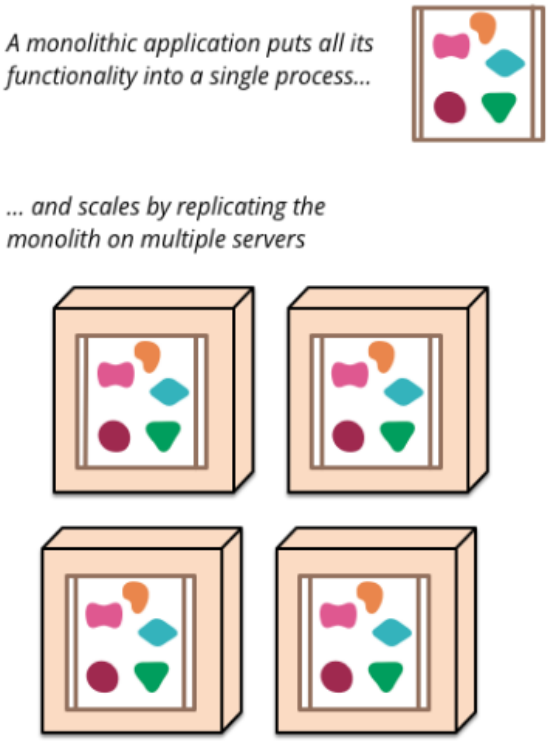

Figure 2.1: Monolith - Source: Fowler (2014)

a system that gets too 'big' as a monolithic application. The characteristic 'bigness' does not really fit into my implemented demo monolithic application because However, the characteristic 'big' is subjective and a monolithic application has to be defined more precisely. The same author Raymond mentioned in his book another characteristic. He says a monolithic application runs in one single process. This is much more accurate than big. He did not come up with a definition and so other researchers grasp the term and formed a definition. For example, Fowler (2014) defined:**"A monolithic application puts all its functionality into one application and runs the application as a single process."**

For the further discussion, this thesis uses exact this definition. In figure 2.1 you can the illustration of the monolithic architectural pattern.

As I mentioned in the methodological framework in section 3 the demo application runs on the web server Wildly. All functionality is packed into one single process. This has the advantage that a monolithic application gets high availability easily. You just have to run multiple copies of the whole application behind a load balancer.

The advantage is that the scalability is not very detailed, because you always replicate the entire application. You cannot separate your application into smaller parts because the whole application runs as a single process.

A monolithic application runs on multiple server instances. This results in linear scalability (1X, 2X, 3X etc.) (Villamizar et al., 2017). If you want to scale your monolithic application at a more granular level (1.1X, 1.4X, 2.5X etc.) you have to use server instances with different computer resources. However, these strategies are difficult. Villamizar et al. (2017) found out in his study, that Amazon AWS only allows the same server type four auto-scaling groups, therefore a monolith can be only scaled in a linear way. In addition, a monolithic can only scale in one dimension. It scales by running more replicas and so you always add all computer resources (processor, memory, storage, and network) with each new instance. This is called on one dimension. If you just want to scale one part of your application, for example, the component which handles database connections, you cannot do this with a monolithic approach. This is called scaling in another dimension for example as mentioned database component.

2.4.1 Pattern Description Format

The monolithic pattern can be explained by using the pattern description format in section 2.2.4.

Context

A monolithic application is built as a single unit.

Problem

The application cannot be split into individual services or modules.

Solution

All the code is in a single codebase that is compiled together and produces a single application, which runs in its own single process.

2.4.2 Summary

In summary, a monolithic application can be scaled easily based on a linear strategy in one dimension by running multiple replicas behind a load balancer. But the replication of entire application results in an allocation of unnecessary computer resources because the scalability is not on a fine level. This results in high infrastructure costs.

Therefore we need a more agile approach to avoid infrastructure costs. In the next section, we discuss the microservice architecture, which decomposes the entire application into smaller parts.

2.5 Microservice Architectural Pattern

As Fowler (2014) defined, **a microservice architecture puts each element of functionality into a separate service**. Therefore we have to analyze the requirements in section 1.3.1 again and derive individual services. Each service runs as a separate process and communicates with each other. In figure 2.2 you can find an illustration of the microservice architectural pattern. The microservice architectural pattern designs the application as suites of independently deployable services.

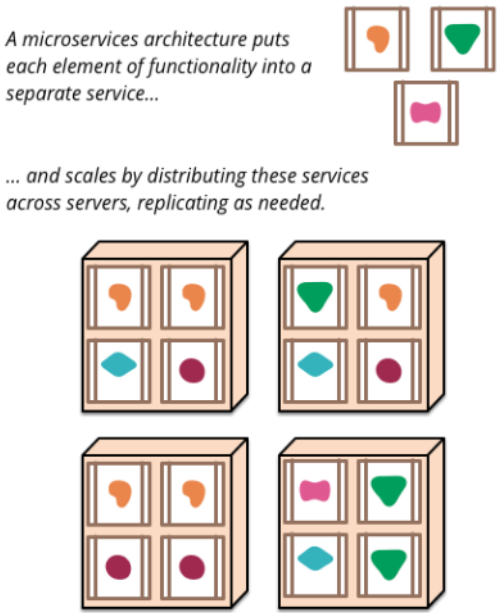

Figure 2.2: Microservice - Source: Fowler (2014)

2.5.1 Microservice Philosophy

The philosophy of a microservice architectural pattern is essentially equal to the Unix philosophy of ”do one thing and do it well”. This philosophy was stated by the Unix pioneer (McIlroy, 1978). When splitting a monolith into smaller microservices, then each service has to be small and focused on doing one thing well. This can be difficult because the codebase of a monolithic application is large. The concept is called ”cohesion”.

Newman (2015) stated cohesion as follows: ”Cohesion is the drive to have related code grouped together. This is an important concept when we think about microservices.”

R. C. Martin and M. Martin (2006) reinforced the concept cohesion and defined the ”Single Responsibility Principle”, which states ”Gather together those things that change for the same reason, and separate those things that change for different reasons.”

In a monolithic application, you force the ”Single Responsibility Principle” by trying to get clear modules.

Microservices use the same approach to get independent services.

2.5.2 Characteristics

The characteristics of a microservice pattern are as follows (Fowler, 2014):

1. Componentization via Services
2. Organized around Business Capabilities
3. Products not Projects
4. Smart endpoints and dumb pipes
5. Decentralized Governance
6. Decentralized Data Management
7. Infrastructure Automation
8. Design for failure
9. Evolutionary Design

Componentization via Services

As I already defined in the pattern definition section 2.2.1, a web-based information system consists of multiple components. And in the microservice architectural pattern, a component is realized as its own service.

Organized around Business Capabilities

When you split an application into smaller parts, you may have the problem that you split them around business capabilities. This well-known phenomenon is called Conway's law and is discussed in section 2.8.7 in this thesis.

Products not Projects

The microservice model uses the approach to build and operate each service as a product over the full lifetime: The development team takes full responsibility for the software in production.

Smart endpoints and dumb pipes

Microservices uses simple communication protocols to communicate with each other. The logic itself is implemented in the service.

Decentralized Governance

A decentralized governance means, that you can use different programming languages and technologies for solving a problem. Each microservice can be implemented in a different programming language and communicates with a lightweight protocol each other.

Decentralized Data Management

Each microservice has its own persistent data and the management is decentralized.

Infrastructure Automation

An application with an microservice architectural pattern consists of many services. Therefore, you need as much automation as possible. Otherwise, you can not deploy and test efficiently.

Design for failure

In a microservice architecture, the failure of one service does not mean that the entire application is down.

Evolutionary Design

The microservice approach stands for an evolutionary approach. Therefore, I discuss this approach in this thesis. Together with the other mentioned aspects in this thesis, I form an evolutionary development of web-based information systems.

The communication between services is an additional challenge because it has to be organized.

Firstly, you have to know which service is running and available. As provided by Newman (2015), a service discovery application is needed that registers all available services. In this thesis, we call a service discovery application also registry application.

As proved by Torkura, Sukmana, and Meinel (2017) the microservice registry is an essential component of the microservice architecture. It ties all components together and enables them to communicate with each other.

Secondly, you need a central entry point for your user. In a microservice architecture, we call this entry point 'gateway'. As proved by (Wizenty et al., 2017) a microservice gateway is the central access point to the user. The gateway routes each user request to the appropriate service application. It also verifies if the client is authorized to perform the request to the service.

In regards to infrastructure costs, we need additional server instances for the microservice gateway a registry.

A microservice pattern also decentralizes the database storage. Microservices prefer that each service handles its own database. In figure 2.3 you can see the difference between the persistence model of a monolith and microservice architectural pattern.

In comparison to the monolithic architecture, a microservice architecture has a higher flexibility for scaling. As Fowler (2014) mentioned, a microservice architecture scales by distributing services across servers. Therefore we do not need to replicate the entire application, just small parts as needed. A part of an application allocates less computing capacity than the entire

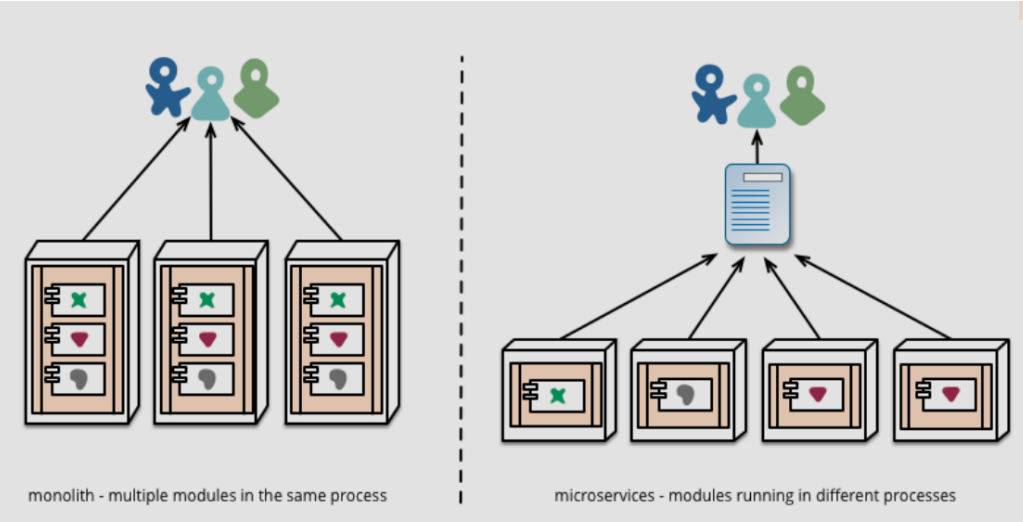

Figure 2.3: Database Monolith vs. Microservice - Source: Fowler (2014)

application. Therefore a microservice application has higher flexibility for scaling and this ends up in fewer infrastructure costs. In figure 2.4 you can see the difference between the monolith and microservice architecture.

2.5.3 Pattern Description Format

The microservice pattern can be explained by using the pattern description format from section 2.2.4.

Context

The microservice pattern designs the application as suites of independently deployable services.

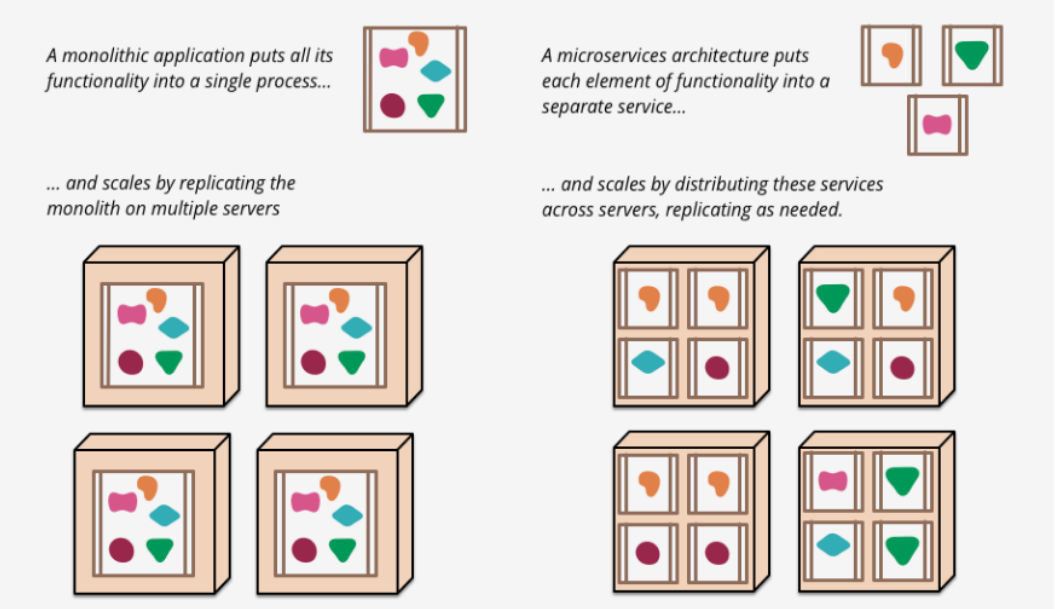

Figure 2.4: Overview Monolith vs. Microservice - Source: Fowler (2014)

Problem

Any changes to the application involve building and deploying the entire application. You have one big application, which is not maintainable.

Solution

As Fowler (2014) defined, "a microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communication with a lightweight mechanism, often an HTTP resource API."

2.5.4 Complexity Monolith vs. Microservice

As you can see, splitting a monolith into individual services takes additional effort. And you need a registry and gateway for the microservice infrastruc-

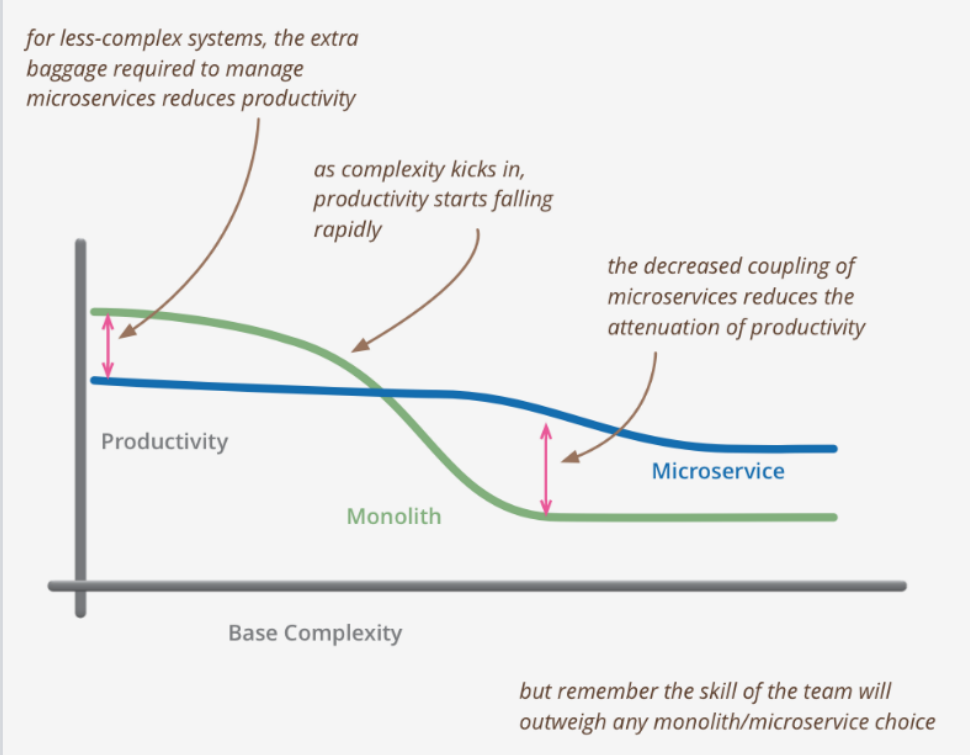

Figure 2.5: Complexity Monolith vs. Microservice - Source: Fowler (2015)

ture. In the beginning, a microservice infrastructure is more complex than a monolithic setup. In figure 2.5 you can see an illustration of the complexity for monolithic and microservice setup.

2.5.5 Summary

In summary, a microservice architecture decomposes an application into services. Instead of one process, you have a network of communicating process. This communication is organized via a registry application. The access point is a gateway application. Scalability works on basis of a finer granularity

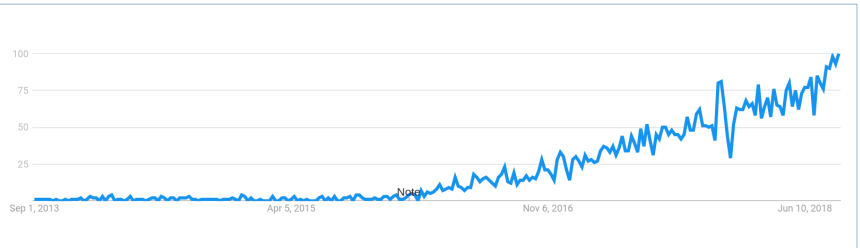

Figure 2.6: Google Trend Analysis

and therefore infrastructure costs are lower in comparison to a monolithic application. But you still not tackle the problem with infrastructure cost for idle sever instances. Finally, we discuss the agilest approach that is called serverless architecture.

2.6 Serverless Architectural Pattern

The term serverless is very new. A Google Trends[3] analysis indicates the novelty of serverless.

2.6.1 Novelty Serverless

Figure 2.6 shows how often the term serverless is searched globally on Google[4].

The interest over the last 2 years has been increasing rapidly. The number on the y-axis represents search interest relative to the highest point. The time is given by the x-axis. A value of 100 is the peak popularity for the term in August 2018. A value of 50 means that the term is half as popular as the peak. Likewise, a score of 1 in August 2015 means the term was 1% as popular as the peak.

[3]https://trends.google.com
[4]https://www.google.com

Therefore, a cloud provider such as Amazon AWS[5] or Google[6] is trying to brand the term with their products. Each company does this in a different way and this leads to misunderstanding and confusing in the IT community.

Amazon AWS announced via Twitter[7] on 12 October 2017:

```
Enterprises using #serverless don't manage infrastructure.
Faster time to market + lower costs = happier customers.
```

In a similar way, Google Cloud advertised their products via Twitter[8] on 9 March 2017:

```
Now in beta: Google Cloud Functions #serverless #googlenext17
Platform for building event-based microservices.
```

Both companies speak in a very general way about serverless. Amazon AWS connects with the term unmanaged infrastructure, faster time to market, lower costs and happier customers.

Google Cloud sees a platform for event-based microservices. This is a very broad spectrum and supports the misunderstanding in the IT community.

Due to the novelty of the term serverless, also only a few pieces of scientific literature exists about the term. Eyk, Iosup, Seif, et al. (2017) confirmed, that we have a lack of clearly defined terminology of serverless in the scientific literature. In addition, they provided that the term serverless suffers from a community problem that faces every emerging technology, which has also hampered cloud computing a decade ago: lack of clear terminology and scattered vision about the field.

Roberts (2016) acknowledged this missing definition and confirmed that serverless is a trend and there is no one clear view of what is serverless.

In contrast to previous research work and the mentioned marketing terms, I focus on clarifying the term serverless in this thesis.

[5]https://aws.amazon.com
[6]https://cloud.google.com
[7]https://twitter.com/awscloud/status/918625698323030017
[8]https://twitter.com/awscloud/status/918625698323030017

2.6.2 Definition

The term serverless is used one of the first times in an article in the year 2012 by Fromm (2012). Fromm mentioned 'serverless thinking' among other things. Software developers should not think so much about the underneath server infrastructure and so they can focus more on creating business value. He wanted to improve the developer experience. In the same year, 2012, the platform Iron.io[9] used the term serverless as well and speaks from an idea of an 'on-demand' or 'pay as you go' model.

But in the year 2012, the term serverless was not broadly known in the IT community.

As you can see in figure 2.6 the term become more popular in 2015, when Amazon launched their serverless service 'AWS Lambda' and advertised the platform in 2014.[10]

Developer experience is also mentioned by Roberts (2016). But he also mentioned the benefit of cost optimization. He confirmed that a serverless architecture reduces costs in regards to availability and scalability. Because the cloud provider always guarantees high availability and scalability for offered serverless services and so the user does not need extra server instances to establish these requirements. Of course, the cloud provider will host your application on several servers. Serverless does not mean that we are no longer use servers. The cloud provider also charges extra costs when he offers services such as high availability and scalability to the user. But Eyk, Iosup, Seif, et al. (2017) found out that costs for a serverless architecture are up to 77.08% lower than costs for a monolithic architecture. The difference is the cost model. You only pay for the duration of the executing time. Therefore you also only pay the cost for high availability and scalability when your application is actually running and needed.

So far we have found an architecture that tackles our main problem with idle server instances.

[9] https://www.iron.io/

[10] https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf

Eyk, Iosup, Abad, et al. (2018) acknowledged in a study that a serverless architecture minimizes infrastructure costs in comparison to server instances that are used for monolithic and microservice applications. In a previous study, the same author Eyk, Iosup, Seif, et al. (2017) defined a serverless architecture by three key characteristics:

- **Granular billing**: The user is charged only when the application is actually executing.
- Almost **no operational costs**: The cloud provider takes over about scalability and high availability. These costs are already included in the charge for the application. But the difference is that the cloud provider only charges the actual executing time and no idle time. Therefore the total infrastructure costs for a serverless application are lower than for a monolithic application.
- **Event-Driven**: Interactions with serverless applications are short lived. The application responds to events when needed. In regards to the demo application, an event represents a user request from the Alexa device.

The Cloud Native Computing Foundation (CNCF) acknowledge these three factors in their current serverless white paper (CNCF, 2018).

In this thesis serverless architecture is defined by these three characteristics: **granular billing**, **no operational costs** and **event-driven**.

In a serverless architecture, an application runs as a serverless function. Eyk, Iosup, Seif, et al. (2017) define a serverless function as a **small**, **stateless**, **on-demand service** with a **single functional responsibility**.

- The characteristic small represents a function with a single functional responsibility. On function implements exact one specific business logic. For example, in the demo application, one specific business logic is reading all available facts from the database.
- A stateless function is a function that does not persist any state from one invocation to the next invocation (Baldini et al., 2017). This means, each user request has its own data and does not transfer any data to the next user request. Even when the same user triggers several times the same function, stateless means data are not shared. Except for data that is stored persistently in a database.

- The characteristics on-demand stands for a function that only runs when the function is actually needed. Therefore a serverless function is never idle and the cloud provider does not charge any costs for idle server capacities.

In this thesis, we reduce the term serverless to a serverless function. Whenever we speak about serverless architecture we refer the term to one or more serverless function as described in this section.

But remember, servers are still required to run a serverless architecture. The main difference is the on-demand cost model.

2.6.3 Pattern Description Format

The serverless pattern can be explained by using the pattern description format in section 2.2.4.

Context

Software developers do not have to think about the underneath infrastructure, they can focus more on creating business value.

Problem

You have already split your application into individual services, but you still have high costs for idle server instances.

Solution

The cloud provider offers an on-demand cost model for their infrastructure. You only pay costs when you run a specific function. Therefore, a serverless function is never idle.

2.6.4 Summary

Finally, I use a serverless approach to run the code for the Alexa skill logic. As a developer, I do not care of the underneath server infrastructure. My primary goal is to deliver business value. The business value in the practical example to deliver the Alexa skill.

So far we have investigated in all three software architectures and their infrastructure costs. We know how scalability and high availability influence the costs in each architecture. When you consider the requirements in section 1.3.1 we still have undiscussed functionalities such as HTTP endpoints or the communication format JSON.

In the next section, we discuss the remaining requirements and their influence on infrastructure costs.

2.7 Further Characteristics of the Case Study

In the previous section, we have discussed the requirements scalability, high availability, and performance. These requirements turned out as cost factors for serer infrastructure. These cost factors vary from architecture to architecture. We have seen that a serverless architecture has fewer fix costs than a monolithic architecture.

In the next section, we discuss further requirements of the demo application and how they influence infrastructure costs.

2.7.1 HTTP Endpoint

As I already mentioned the demo application uses the client-server model. The client sends a request to the server and the server sends a response back to the client. The server offers entry points to the client for the communication. Such an entry point is called an **HTTP endpoint**. The offering of HTTP endpoints allocates computing resources and the cloud provider

charges costs. The total costs depend on the number of endpoints and how often they are called by the clients.

The structure of such a request to an HTTP endpoint is defined in the Hypertext Transfer Protocol (HTTP). Berners-Lee (1989) originally introduced the term hypertext that stands for an ”human-readable information linked together in an unconstrained way”. And HTTP is the protocol to send and receive hypertext.

The open standard organizations ’Internet Engineering Task Force’ (IETF)[11] and the World Wide Web Consortium (W3C)[12] redefined the HTTP protocol in a series of publication. For example, R. Fielding and Reschke (2014b) defined in an IETF standard the HTTP method **POST**. The demo application sends all request via the POST method. A POST request sends data as a block of data in the request to the server and the server processes this data. The processing on the server allocates computing capacity and the cloud provider charges costs. The more data the more computing capacity is needed and the more the cloud provider charges costs to you. The block of data can be called ’payload body’ as well.

R. Fielding and Reschke (2014a) also defined the HTTP as a **stateless** protocol. We have already discussed the characteristic stateless in context of serverless functions in section 2.6.2. In the context of HTTP, stateless has the same meaning: Each request can be understood as isolation and does not share any state or data with other requests.

In this thesis, each HTTP endpoint is implemented as a **RESTful** web service. R. T. Fielding (2000) introduced and defined the term Representational State Transfer (REST) in his doctoral dissertation. REST is not a standard itself, but it is most commonly used over HTTP.

REST basically means that all necessary parameters for calling a web service are encoded into an URL (Scerbakov, 2018).

As Newman (2015) summarized in his work, ”REST itself does not really talk about underlying protocols. Although it is most commonly used over HTTP. Most important is the concept of resources: You can think of a resource as a

[11] https://www.ietf.org
[12] https://www.w3.org

thing that the service itself knows about, like a Customer. The server creates different representations of this Customer on request. How a resource is shown externally is completely decoupled from how it is stored internally. A client might ask for a JSON representation of a Customer, for example, even if it is stored in a completely different format. Once a client has a representation of this Customer, it can then make requests to change it, and the server may or may not comply with them".

In summary, the demo web application is implemented via RESTfull web services. Data are sent as a POST request to the server and the communication is stateless over the protocol HTTP. The entry point of a web service is called an HTTP endpoint.

Each request has a specific format of the data. In the next section, we speak about the data format JSON.

2.7.2 JSON

As I already mentioned in the previous section, data are sent as a block in the request to the server. This block has a specific format. In this thesis, we use the JavaScript Object Notation (JSON). Bray (2017) defined the format as a lightweight, text-based data interchange format. The basic formatting rules are key/value pairs. A single comma separates a key from the following pair. Each key/value pair is surrounded with curly brackets.

JSON is an open standard and therefore you do not have to pay any costs for the usage of the format. But the format is text-based and the more data is transferred, the more data has to be processed on the server. The processing on the server itself allocates computing resources and therefore the cloud provider charges costs. And so the amount of data influences the infrastructure costs.

The demo application only accepts data in JSON format as input for all web services. The generated output of all web services is also in JSON format.

In the next section, we discuss how different services can be invoked.

2.7.3 Synchronous and Asynchronous Invocation

The demo application is decomposed into services. Each service is responsible for one specific business logic of the entire web application.

The user invokes a service through the HTTP endpoint. The service processes the request and responses with data in the JSON format.

A service can also invoke other services. For example, one service uses the output from another service as input. Therefore, another service has to be invoked. In this case, the service, that invokes another service, can only process a request as fast as the other invoked service can perform them. In regards to the demo application, a sequence of service invocations might have a longer response time than just an invocation of one single service. The maximal response time of the demo application is defined in section 1.3.1. Therefore, the invocation type has to be considered when the demo application fulfills all requirements.

Hohpe and Woolf, 2003 described in their book the different invocation types: synchronous and asynchronous.

- **Synchronous invocation:** The caller invokes a service and waits for a response. It is also called a *request-reply* or *sequence* model.
- **Asynchronous invocation:** The caller invokes a service and does not wait for a response. The caller continues its remaining work. This is a *parallel* invocation model.

Of course, an asynchronous invocation model only works when the caller service does not directly rely on data from the invoked other services. In the case of the demo application, the caller service always relies on data from invoked services and therefore all services are synchronously invoked.

In summary, the demo application consists of multiple services. Each service can be invoked synchronously or asynchronously. Synchronous invocation means that the caller service waits for the response of the invoked service. In contrast, the asynchronous model does not wait for a response, but an asynchronous model only works when the caller service does not directly need the other service's output for the remaining work. When a service waits, the waiting has an influence on the maximal response time of the

entire application and therefore we consider the invocation model in the demo application.

2.7.4 Summary

The user invokes a service through an HTTP endpoint. Each service is implemented as a RESTful web service. All services accept only JSON data and respond accordingly with JSON data. When a service invokes another service, then the invocation model is synchronous. The caller service waits for the response of the called service and then the output of the called service is used for further work.

In the next section, I discuss the term 'evolutionary development'.

2.8 Evolutionary Development

This thesis follows the basic rules of the 'Unix Philosophy'. On top of these basic rules, I use the guidelines 'Twelve-Factor App' and 'jHipster' to analyze and implement the case study.

The development of a modern web application is hard.

Evolutionary development is an iterative, agile and flexible methodology to implement the case study web application in each specific approach (monolithic, microservice and serverless).

Wright and Perry (2012) analyzed in his paper common pitfalls for hosted web applications. He also investigated the relationship between software architecture and development process.

During software development, you have to tackle problems and pitfalls. Best practices help to avoid these problems. Especially, this thesis focuses on reducing infrastructure costs and therefore I follow best practices to avoid infrastructure costs.

The ”one-size-fits” methodology for software development does not exist, but best practices are exist to tackle common problems. Wiggins (2018) provide with ”Twelve-Factor-App” a set of practices for architecting , building and maintaining modern web applications.

First of all, I explain the basics of the Unix Philosophy.

2.8.1 Basics of the Unix Philosophy

Evolutionary development of web-based information system follows the basic rules of the 'Unix Philosophy'. The Unix Philosophy is a set of cultural norms and philosophical approaches to minimalist and modular software development. It is based on the experience of leading developers of the Unix operating system.

The Unix philosophy is not a formal design method, created by theoretical computer scientists. It is pragmatic and grounded in experience. It is not to be found in official methods and standards, but rather found in the implicit knowledge of the expertise in the Unix culture community. It encourages a sense of proportion and skepticism. (Raymond, 2003).

The most important concepts in software engineering are modularity and reusability. In these concepts are well summarized in the Unix philosophy.

In conclusion, the Unix philosophy helps to build simple, short, clear, modular and extensible code that can be easily maintained and changed. It gives us the basis of evolutionary development of web-based information system. In the next section, I discuss further details of the Unix philosophy.

2.8.2 Do One Thing and Do It Well

The researcher McIlroy (1978) firstly documented the Unix philosophy and characterized them as follows:

- Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.

- Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input
- Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them
- Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

It was later summarized by Salus (1994) as follows:

- Write programs that do one thing and do it well.
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface.

2.8.3 KISS Principle

Basically, all the Unix philosophy boils down to one iron law: the KISS principle. KISS stands for 'Keep It Simple, Stupid!' (Raymond, 2003). This is the Unix philosophy summarized in one sentence. And also the entire operating system Unix is an excellent example and base for applying KISS principle. Unix developers followed the KISS principle and created, for example, the simple and powerful command line tools cat or grep.

In this thesis, you see the KISS principle in the REST endpoints and the simple JSON format. There is a lot of hype surrounding REST or JSON, but I do not adopt or reject it uncritically. I carefully follow the KISS principle and keep them simple.

As you can see, the 'KISS principle' and 'Do one thing and do it well' concept are basics of evolutionary development of a web-based information system.

In conclusion, this is the Unix philosophy defined by Salus (1994): "Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface."

On the top of these basics, researches created following further rules.

2.8.4 Eric Raymond's 17 Unix Rules

The Unix enthusiast, programmer and researcher Raymond (2003) provided a series of design rules:

1. Rule of Modularity: Build modular programs
2. Rule of Clarity: Write readable programs
3. Rule of Composition: Use composition
4. Rule of Separation: Separate mechanisms from policy
5. Rule of Simplicity: Write simple programs
6. Rule of Parsimony: Write small programs
7. Rule of Transparency: Write transparent programs
8. Rule of Robustness: Write robust programs
9. Rule of Representation: Make data complicated when required, not the program
10. Rule of Least Surprise: Build on potential users' expected knowledge
11. Rule of Silence: Avoid unnecessary output
12. Rule of Repair: Write programs which fail in a way easy to diagnose
13. Rule of Economy: Value developer time over machine time
14. Rule of Generation: Write abstract programs that generate code instead of writing code by hand
15. Rule of Optimization: Prototype software before polishing it
16. Rule of Diversity: Write flexible and open programs
17. Rule of Extensibility: Make the program and protocols extensible

As Raymond (2003) confirmed, most of these mentioned rules are highly recommended in software engineering. The operating system Unix is a great example. All these rules are conducted within Unix. Therefore, we also adapt these rules to the evolutionary development of web-based information systems.

Rule of Modularity: Write simple parts connected by clean interfaces.

The researcher Kernighan and Plauger (1976) stated an impressive result: "Controlling complexity is the essence of computer programming". This mean, if you want to write complex software, you have to control its global complexity. Therefore, you have to build simple local parts connected by well-defined interfaces. If one small local part gets down, then the remaining global system still works.

Rule of Clarity: Clarity is better than cleverness.

In the Unix tradition, you choose algorithms and implementations for future maintainability. This advice goes beyond just commenting your code. A code is written to the human beings who will read and maintain the source code in the future. Not otherwise. You do not write code for the computer.

Rule of Composition: Design programs to be connected with other programs.

As Raymond (2003) summarized, "To make programs composable, make them independent. A program on one end of a text stream should care as little as possible about the program on the other end. It should be made easy to replace one end with a completely different implementation without disturbing the other."

Rule of Separation: Separate policy from mechanism; separate interfaces from engines.

In general, you should separate the interface (=front-end) from the engine (=back-end). The front-end implements policy and the back-end implements mechanism.

Rule of Simplicity: Design for simplicity; add complexity only where you must.

You should encourage a software culture, that knows that small is beautiful and resists complexity. The goal is to put a high value on simple solutions.

Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.

A big program is large in volume of code and of internal complexity. Large programs are difficult to maintain.

Rule of Transparency: Design for visibility to make inspection and debugging easier

Raymond (2003) acknowledged, that debugging often occupies three-quarters or more of development time. Therefore, you should ease debugging by designing for transparency and discoverability.

Rule of Robustness: Robustness is the child of transparency and simplicity.

A robust software performs well under unexpected conditions. Therefore you need a transparent and simple software. "A software is transparent when you can look at it and immediately see what is going on. It is simple when is going on is uncomplicated enough for a human brain to reason about all the potential cases without strain" (Raymond, 2003).

Rule of Representation: Fold knowledge into data, so program logic can be stupid and robust.

You should shift complexity from program logic to the data structure. Then you will see a difference in transparency and clarity. Because procedural logic is hard for humans to verify, but complex data structure is fairly easy to model and express (Raymond, 2003).

Rule of Least Surprise: In interface design, always do the least surprising thing.

The easiest program interfaces use the pre-existing knowledge of their users.

Rule of Silence: When a program has nothing surprising to say, it should say nothing.

These design rules say you should treat the user's attention only when it is necessary.

Rule of Repair: Repair what you can — but when you must fail, fail noisily and as soon as possible

Software should be transparent. In case of an error, the software should never quietly quit the error and hide the error to the user.

Rule of Economy: Programmer time is expensive; conserve it in preference to machine time

This rule means, that the machine should do more of the low-level work of programming. For example, a programmer does not teach memory management to the machine or write it from scratch. The programmer shifts this low-level work to the machine itself. This lets to the next following rule.

Rule of Generation: Avoid hand-hacking; write programs to write programs when you can

In the Unix tradition, you use generic and abstracted code. The best example is the usage of a code generator. You do not start every project from scratch and you automate error-prone detail work. In this thesis, we use the code generator jHipster. This generator is described in section 2.8.6.

Rule of Optimization: Prototype before polishing. Get it working before you optimize it.

In the Unix world exists a long tradition: Make it work first and then make it work fast.

Rule of Diversity: Distrust all claims for "one true way".

The Unix tradition includes a healthy mistrust of "one true way" in software design and implementation. You always have multiple languages, open extensible systems, and much customization.

Rule of Extensibility: Design for the future, because it will be here sooner than you think.

Software should be able to add new features without rebuilding the architecture.

Summary

The Eric Raymond's 17 Unix rules give a fundamental setting for software development. The success you can see on the operating system Unix, which is used worldwide. In addition, this thesis adds best practice guidelines and forms the evolutionary development of web-based information systems. One of the best practices is called "The Twelve-Factor App", described in the following section.

2.8.5 Twelve-Factor App

The "Twelve-Factor App"[13] is a set of guidelines for building modern web-based information systems. The Twelve-Factor App has elements of **costs** and **architecting**, therefore I discuss the guidelines in this thesis. An evolutionary development environment means, that you find methodologies in the development process to build web-based applications efficiently.

As Lerner (2014) summarized in his article, "the well-known hosting company Heroku[14] looked at thousands of web applications and tried to extract from the factors that made it more likely that they would succeed."

Heroku's CTO, Adam Wiggins, wrote up all recommendations and named them the "Twelve-Factor App". These twelve factors describe practices that Heroku believes will make an evolutionary development of web application and more likely result in a successful and maintainable application.

In this section, I discuss the twelve factors for an evolutionary development of a web-based information system. These factors describe best practices for a maintainable web application. I describe each factor what they mean and how you can use them. In addition, I discuss how you can use the factors in each architectural pattern in this thesis: monolithic, microservice and serverless.

Here you can find the 12 factors for an evolutionary development, summarized by Lerner (2014):

[13] https://12factor.net/
[14] https://www.heroku.com

1. **Codebase**
   One codebase tracked in revision control, many deploys.

2. **Dependencies**
   Explicitly declare and isolate dependencies.

3. **Config**
   Store config in the environment.

4. **Backing services**
   Treat backing services as attached resources.

5. **Build, release, run**
   Strictly separate build an run stages

6. **Process**
   Execute the app as one or more stateless processes.

7. **Port binding**
   Export services via port binding

8. **Concurrency**
   Scale-out via the process model

9. **Disposability**
   Maximize robustness with fast startup and graceful shutdown

10. **Dev/prod parity**
    Keep development, staging, and production as similar as possible

11. **Logs**
    Treat logs as event streams.

12. **Admin processes**
    Run admin/management tasks as on-off processes.

Codebase

A twelve-factor app is always tracked in a version control system, for

example Git[15]. This is called code repository.

The case study uses for each implementation a single repository. You can find them in the methodology section 3.

Dependencies

Twelve-factor apps never rely on implicit existence of system-wide packages. Therefore, the case study uses a central configuration file with strict versions. All dependencies are included as well in the central configuration file and the software never loads a new library into the project automatically.

Config

An evolutionary development of web-based information systems uses different configurations for deployment environments, such as development, staging or production. You should store these configuration parameters in a separate place than the actual codebase. This is especially critical for passwords. Twelve-factor app methodology recommended, that the best practice is to store configuration values in environment variables that are populated during deploy.

Backing services

The Twelve-Factor methodology sees any service, which is consumed over the network as backing service. For example, the database MySQL[16] or PostgreSQL[17]. These are attached resources. An attached resources can be swapped without code changes.

Build, release, run

The twelve-factor app uses strict separation between the build, release and run stages. The build stage converts a code repository into an executable bundle known as a build. The release stage takes the build and combines it with the deploy's current config. The result is ready for immediate execution in the execution environment. The run stage runs the app in the execution environment.

[15]https://git-scm.com
[16]https://www.mysql.com
[17]https://www.postgresql.org

Processes

Twelve-factor processes are stateless and share nothing. Any data that needs to persist must be stored in a baking service, for example, a database service. The monolithic application runs as one single and stateless process. Each microservice in the microservice architectural pattern runs as own stateless process as well. And each serverless function is a stateless function.

Port binding Twelve-factor apps do not rely on the underlying infrastructure. It always exposes all necessary ports itself. If a web server is necessary, then the application itself brings its own web server.

Concurrency The application runs by one or more processes. It is the process model that you know from the operating system. Scaling an application just means starting a new process on the same or different server.

Disposability

The twelve-factor app's processes are disposable. This means, that they can be started or stopped at a moment's notice.

Dev/prod parity

The twelve-factor app is designed for continuous deployment by keeping the gap between development and production small.

Logs Logs should be written to the output stream of the application. The stream can be routed to a file or to a terminal. But the application never concerns itself with routing or saving of the log output stream.

Admin processes

All admin process must be run in isolated processes on the identical environment as the production.

Summary

The Twelve-factor App principles help you to create a robust web-based information system. These principles are an essential part of an evolutionary development environment.

In the next chapter, I discuss the code generator jHipster.

2.8.6 jHipster Policies

As I already mentioned a modern application is configured via code generator. You do not start from scratch. You use already a best practice setup. For example, you can use the code generator JHipster (Raible, 2018). This technique is also called ”scaffolding” tool. You can build a minimal setup easily. That enhances your productivity and satisfaction when building a modern web application.

The JHipster development team follow *coding policies*. The term ”best practices” or ”guidelines” can be used as well and stands for coding policies in this thesis.

- **Policy 0: Polices are voted by the development team**
  JHipster has a core development team of 19 developers and over 350 contributors[18]. They all discuss together their own policies on the mailing list[19]. Each policy comes from the team itself and it is never forced from one individual person or institution without any discussion.

- **The Technology used by JHipster have their default configuration used as much as possible**
  Each technology has already a very good default configuration and there is no reason to change usual naming and coding conventions. Many changes in the standard configuration lead the system to work in an ”unusual way” and are difficult to understand.

- **Only add options when there is sufficient added-value in the generated code**
  Only add complex options to the JHipster code generator. A complex option implies a configuration or coding in several components.

[18] http://www.jhipster.tech/team
[19] http://www.goo.gl/ABR9s3

- **Follow the IntelliJ formatting and coding guidelines**
  The default Java rules provided by IntelliJ IDEA are the standard formatting and coding guidelines for the JHipster project [20].

- **Use strict versions for third-party libraries**
  Different library versions lead to conflicts. Never load a new library version into your project automatically. It is primarily a JavaScript problem, but be safe and declare your strict version in a centralized configuration file, for example, *bower.json* or *package.json*.

2.8.7 Conway's Law

When you talk about architectural patterns, then you also have to take a look at the organizational structure of a company. Because the organizational structure is responsible for producing the software and their architecture. And the structure of a company influences the produced software. The researcher Conway (1968) introduced this idea and it states that "organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations. Any piece of software reflects the organizational structure that produced it." For example, "if you have four teams working on a compiler you will end up with a four pass compiler."

Conway argues, that multiple developers must communicate frequently with each other to create a functional software. Therefore, the produced software will reflect the social boundaries of the organization. This described sociological phenomenon is called Conway's Law.

Raymond (1996) restated Conway's law: "The organization of the software and the organization of the software team will be congruent".

Coplien and Harrison (2004) acknowledged Conway's Law as well: "If the parts of an organization (e.g., teams, departments, or subdivisions) do not closely reflect the essential parts of the product, or if the relationship between organizations do not reflect the relationships between product parts,

[20] https://www.jetbrains.com/help/idea/code-style-java.html

then the project will be in trouble... Therefore: Make sure the organization is compatible with the product architecture."

Additionally, Conway's Law can be also seen in the design of corporate websites. Bevan (1999) stated: "Organizations often produce websites with content and structure which mirrors the internal concerns of the organization rather than the needs of the users of the site."

Why do we discuss Conway's Law in this thesis?

When you split a monolithic application into smaller microservices, then you often focus only on the technology layer. This means that you split teams across the technology. For example a UI team, server-side logic team, and database team. These teams will produce silo application architecture if they do not communicate effectively with each other. In this example, you can see Conway's Law in action. The figure 2.7 illustrated the silo architecture, because of Conway's law.

The solution for tackling this problem is setting up cross-functional teams. A cross-functional team includes the full range of skills: user experience, database, and project management. In figure 2.8 you can see the cross-functional teams.

In conclusion, you can see if you talk about architectural patterns, then you also have to take a look at the organizational structure of a company. Because the structure of a company influences the produced software. The goal is to build cross-functional teams in tackling the Conway's Law.

2.8.8 Problems and Pitfalls

Idle Server Instances

As I mentioned in chapter 2.3 the main problems are **idle server instances** that increase server infrastructure costs. Koomey Jonathan (2015) and Kaplan James M. (2008) proved this in their reports.

Software Erosion

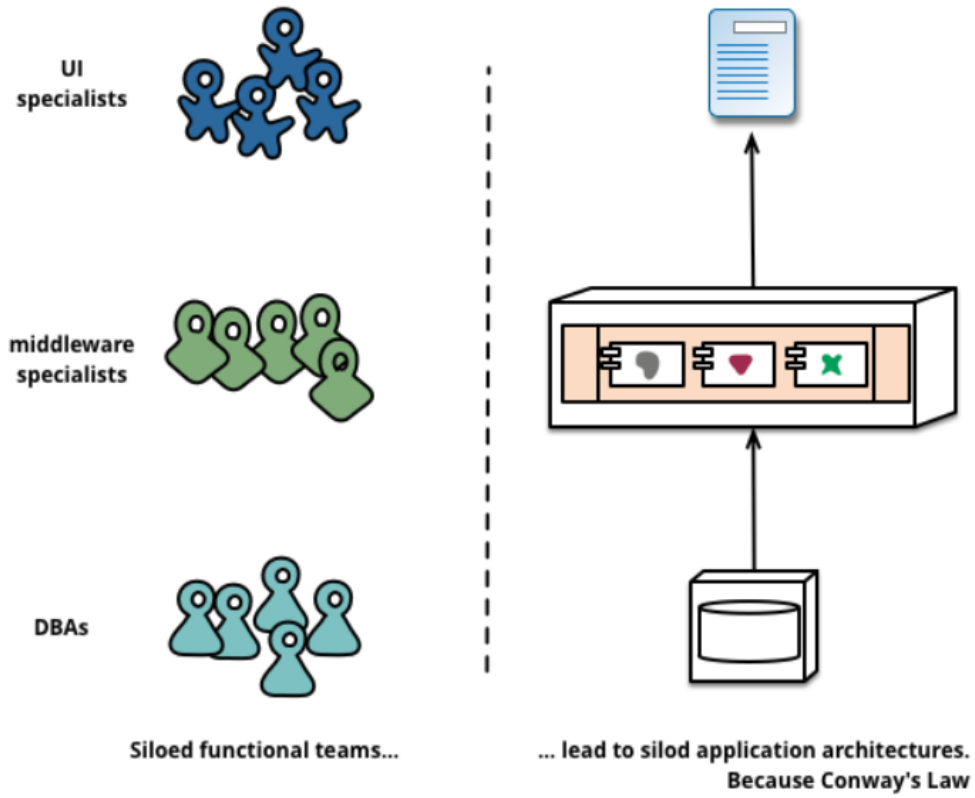

Figure 2.7: Conway's Law in Action - Source: Fowler (2014)

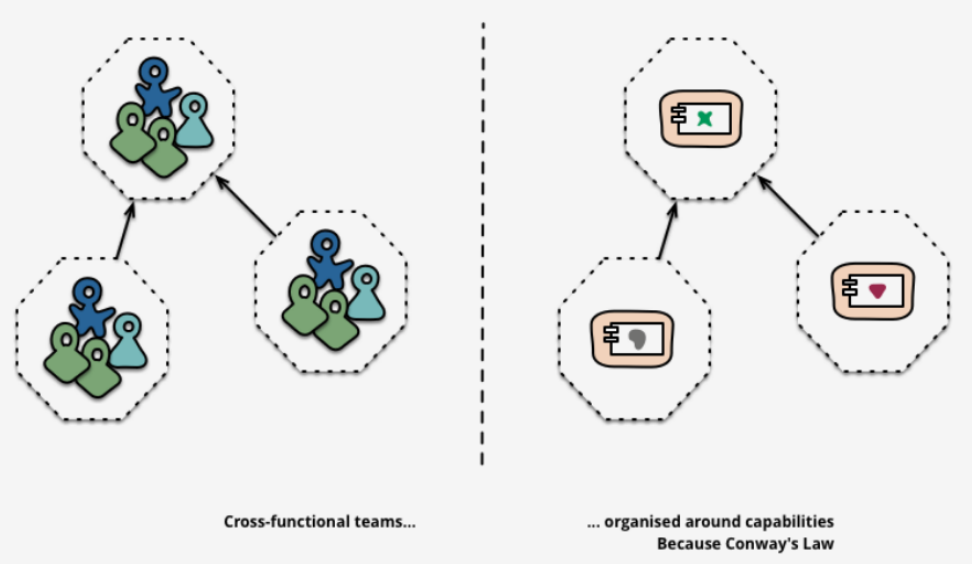

Figure 2.8: Cross-functional teams - Source: Fowler (2014)

Another problem is called **software erosion**. Wiggins (2011) argues in his report that software erosion is a heavy cost. Software erosion describes the process of software that is becoming faulty. A software run in an environment and this environment changes continuously. For example, the operating system installs updates on the server instances. This is a change of the underlying environment in which the software is running. In this case, the software has also to be tested and if needed the software needs updates as well. If no one takes care of the software, the software erodes. A software development team has to fight against software erosion and these leads to additional costs. The chosen software architecture has an impact on software erosion. For example, a monolithic application packs all services into one big process. This process runs on a web server on a server instance. The software development team is in charge of the entire monolithic application. This means, the software team has to update the operating system, the web server and the application itself. As you can see this takes much effort to update all components of a monolithic application. It is a traditional server-based development. The code, config, processes,

and logs are deeply coupled with the underlying server setup.

In comparison to a microservice architecture, these update activities are even more complicated. Because in a microservice approach all services are spread across multiple server instances. A change has an impact on the system-wide microservices. For example in case of an operating system update, multiple server instances have to be updated and the running service on it has to be tested and updated as well. This thesis sees a microservice architecture deeply coupled with the underlying server setup.

The easiest way the avoid software erosion is the serverless approach. Because as I mentioned, the cloud provider takes care about the underlying server instances. The cloud provider guarantees a running system if you follow the requirements for a serverless function. It is the most abstract model and therefore all manual updates are reduced to a minimum. Basically, the cloud provider avoids for you the software erosion.

The goal for a modern software development team is to avoid software erosion and establish a platform which is erosion-resistant. In this thesis, the serverless function platform on Amazon AWS is seen as an erosion-resident platform. Amazon AWS provides an execution environment for your web application[21]. AWS supports following runtime versions listed at September 20, 2018.

- Linux kernel version – 4.14.62-84.118.amzn2.x86-64.
- Node.js – v8.10, v6.10 or v4.3
- Java – Java 8
- Python – Python 3.6 and 2.7
- .NET Core – .NET Core 1.0.1, .NET Core 2.0, and .NET Core 2.1
- Go – Go 1.x

If you follow an evolutionary software development process as described in this thesis, then your software team build the software for an environment runtime, for example, Java 8. At the cloud provider, a serverless architecture is configured which automatically setup up the necessary infrastructure for Java 8 applications. The cloud provider guarantees that the underlying infrastructure is always compatible to Java 8 and therefore the cloud provider

[21] https://docs.aws.amazon.com/lambda/latest/dg/current-supported-versions.html

fights against the software erosion on your behalf. This reduces costs in a modern application development team.

Another reason for software erosion is the fact that you have unused services which run on your server instances and these services are responsible for unnecessary infrastructure costs. Boss et al. (2016) analyzed how such unused services can be discovered. This mentioned research is focused on embedded software development. Especially embedded software development is a good example of reducing unnecessary computing resources and avoiding costs. Because an embedded software run on hardware where the resources are very limited. In comparisons to a server instance at a cloud provider, a hardware with a embedded software only has few MB memory, for example 256MB. A evolutionary software development team chooses a process that efficiently uses this limited resources. And detecting unused services is a key factor in am modern application. This mentioned paper implemented software checks into continuous integration to check absolute services. However, this thesis has not the focus on discovering unused services in the application.

Gurp and Bosch (2002) acknowledged that erosion is a common problem in software engineering. They proved that a higher complexity of architecture results in a higher software erosion over time. Software tends to erode over time, and redesign from scratch becomes a viable alternative compared to prolonging the life of the existing architecture. As you can see the described monolithic architecture in section 2.4 and the microservice architecture in 2.5 have a higher complexity than the serverless approach in section 2.6. The monolithic architecture is strictly coupled with the underlying web server and operating system. The same coupling is valid for the microservice approach. In addition, the microservice architecture is spread across multiple servers and illustrates even a more complex architecture. Therefore, a monolith and microservice application tend to software erosion. The serverless function approach has a less complex architecture for the development team and therefore software erosion is lower and finally less costs are necessary for the software development team. As you can see a correct chosen architectural design are essential to avoid software erosion.

Mair, Herold, and Rausch (2014) discovered a divergence between the intended software architecture and the actually realized architecture during

the development phase. This means you have a progressive divergence during the development phase of a web application. The reasons for this divergence are manifold. For example, coding workaround through time pressure or employee turnover. (Muthig and Lindvall, 2008) analyzed that also adopting new requirements leads to software erosion.

Iyengar et al. (2009) proved that eroded software becomes too costly to be maintained and might need to be replaced be expensive re-developments. An eroded software is difficult for maintaining and adapting. Iyengar et al. (2009) describe it as an unmanageable monolith. The most expensive solution is that developers often rewrite the entire application. For very large systems such approaches are typically impossible to conduct. As an alternative, the modularization approach decomposes the entire application into modules. Each module has a well-defined interface for the communication. As you can see the mentioned microservice approach uses this modularization approach. Each service is an individual modul and communicates with each other. The HTTP endpoints are the well-defined communication interfaces. Iyengar et al. (2009) suggest following additional guidelines for modules:

- Put cohesive functions, data, and files together in one module. A good module provides a set of services related to a specific purpose.
- One module should operate independently of other modules. A module should capture and encapsulate a set of design decisions. The implementation is hidden from other modules. The interaction between modules is through module interfaces, for example, HTTP endpoints.
- A module should not share data structures among other modules. As well function definitions should not be shared across other modules. The result is that each module is built and test independently.
- Reduce compile time dependencies among modules. The result is easier configuration management of the entire application.

Ramage and Bennett (1998), as well as Bisbal et al. (1999) indicated a positive correlation between software maintainability and modularization.

Furthermore, Iyengar et al. (2009) mentioned a layered module approach. A system should be organized into a layered architecture. Each layer has a specific responsibility. For example, a database layer is responsible for database access. A layered architecture has the following characteristics:

- Modules that reside in the same layer can communicate with each other through interfaces.
- Modules that reside in upper layer levels can communicate with the layers below.
- Modules that resides in a layer at lower level should not communicate with modules in higher layers.

This mentioned layered architecture is also included in the described microservice architecture in section 2.5. As you remember, the gateway is the entry point for the microservice approach. The gateway represents an upper layer. The upper layer communicates with the lower layers. The multiples services represent the lower level.

Moreover, Iyengar et al. (2009) use the modularization approach to form modules around business domains. The business domain is a set of functionalities to deliver the same business value. For example, loan calculation. However, some business domains have many business operations. Therefore you have to decompose a complex domain module into submodules.

The researchers Iyengar et al. (2009) mentioned another important practice: The usage of a code generation tool. A coder generator creates a significant part of the code automatically. This thesis uses the code generator jHipster[22]. JHipster creates base code for a Java Spring Boot Application. It uses a modern approach to create a monolithic or microservice application.

Also, the researcher Lerner (2014) tried to find a way to avoid clutter and chaos in web application projects. He suggested the prominent framework "Ruby on Rails" among other things, which uses the model of "convention over configuration". It means, that "developers should sacrifice some freedom in naming conventions and directory locations". Therefore, you can maintain better your web application project. For example, if the methodology dictates the naming conventions and directory locations, then developers understand faster the application and can begin to improve the code faster. The mentioned code generator jHipster gives automatically a structure and naming conventions for your web application.

[22]https://www.jhipster.tech

2.8.9 Modularization

As described in section 2.8.8 a modularization approach decomposes the entire application into smaller parts. Smaller parts can plan, develop, operate and maintain easier than the entire application. The microservice approach introduces its own set of complexity.

"When you use microservice you have to work on automated deployment, monitoring, dealing with failure, eventual consistency and other factors that a distributed system introduces" (Fowler, 2015).

Also within a monolith, you should pay attention to good modularity (Fowler, 2015). Modularization is also used for monolithic applications. You do not have to split everything into service, for getting good modularity.

2.8.10 Layered Architecture

As described in section 2.8.8 a layered architecture helps to avoid software erosion as well. Services on different levels keep the entire application in a better organization. As I mentioned, gateway and registry within the microservice architectural pattern run on different levels than the other services. This means that the user can only reach the gateway from outside. Therefore, the gateway works on a higher level than the other service behind the gateway.

Gurp and Bosch (2002) proved that The result is that the higher the architectural complexity is, the more likely it is that software erosion might occur.

In conclusion, the usage of a code generator, for example jHipster, helps to build modern web applications from scratch. In addition, the concept of modularization and layered architecture helps to keep the software development process in a better organization. These mentioned concepts are summarized in best practices.

But the most important thing among these describes solution is the usage of a well-known best practice methodology.

2.8.11 Summary

In conclusion, the Unix philosophy gives you a fundamental environment of a robust approach for software development. In addition, the Twelve-Factor App guidelines and the jHipster code generator creates an evolutionary environment for developing web-based information systems.

3 Methodology

In this section, I present the methodological framework and so you can replicate the results of this thesis.

Each web application is implemented in *Java 1.8*[1] and is stored in a public Git repository (Figure 3.1). A README file in each root folder explains how you can compile and deploy the application. The persistent storage is two relational database servers: *MySQL 5.7.22*[2] and *PostgreSQL 9.6.6*[3].

All applications run on the cloud provider *Amazon Web Services (AWS)*[4]. The entire setup is configured in the AWS Region *EU Irland (eu-west)*.

Figure 3.2 lists all products that are necessary to set up the deployment environment on AWS.

The costs depend on the computing capacity of the chosen product. In Figure 3.3 you can see the product name with the corresponding computing resource. The pricing of each product is listed on the *AWS Pricing Website*[5]. In the thesis, I used the prices that Amazon listed on *August 1, 2018*.

The entire monolithic application runs on the web server *Wildfly 11.0.0*[6]. Each microservice Java application runs on the web server *Undertow 1.4.25*[7]. All Microservice application are created via *jHipster 5.2.0*[8] projects. The

[1] http://oracle.com/java
[2] https://mysql.com
[3] https://postgresql.org
[4] https://aws.amazon.com/
[5] https://aws.amazon.com/pricing
[6] http://wildfly.org
[7] http://undertow.io
[8] https://www.jhipster.tech

Application	Git
Monolith	
Java Application	https://bitbucket.org/tugraz-thesis/monolith
Microservice	
jHipster Gateway	https://bitbucket.org/tugraz-thesis/microservice-gateway
jHipster Registry	https://bitbucket.org/tugraz-thesis/microservice-registry
Handle Input/Output (Service 1)	https://bitbucket.org/tugraz-thesis/microservice-service1
Get Random Fact (Service 2)	https://bitbucket.org/tugraz-thesis/microservice-service2
Get Fact Description (Service 3)	https://bitbucket.org/tugraz-thesis/microservice-service3
Get Image URL (Service 4)	https://bitbucket.org/tugraz-thesis/microservice-service4
Get Image from Storage (Service 5)	https://bitbucket.org/tugraz-thesis/microservice-service5
Serverless	
Handle Input/Output (Lambda 1)	https://bitbucket.org/tugraz-thesis/serverless-service1
Get Random Fact (Lambda 2)	https://bitbucket.org/tugraz-thesis/serverless-service2
Get Fact Description (Lambda 3)	https://bitbucket.org/tugraz-thesis/serverless-service3
Get Image URL (Lambda 4)	https://bitbucket.org/tugraz-thesis/serverless-service4
Get Image from Storage (Lambda 5)	https://bitbucket.org/tugraz-thesis/serverless-service5

Table 3.1: All implemented Java Applications and their Git repositories

jHipster Microservice Registry application is based on the *Netflix Eureka Server 2.1.0.* [9]

I ran the performance test via *Apache jMeter 4.0*[10]. The configuration was as follows: The tool jMeter sent 10 requests per second as an HTTP Post request to the application. In the body of the HTTP POST request was a specific JSON payload. In section Appendix chapter you can see an example JSON request to the application and the corresponding JSON response from the application. This JSON request represents a JSON request that an original Alexa device sends to the application. You can find the jMeter configuration in a separate Git repository (Figure 3.4).

The performance test was configured on a separate server instance and ran against each server architecture approach: monolithic, microservice and serverless. You can see the test scenario in figure 3.1.

In summary this section I explained the methodological framework and so you replicate the result of this thesis anytime.

[9] https://cloud.spring.io/spring-cloud-netflix/spring-cloud-netflix.html
[10] https://jmeter.apache.org

Application	AWS product name
Monolith	
Java Application	EC2 t2.xlarge
Microservice	
jHipster Gateway	EC2 t2.small
jHipster Registry	EC2 t2.small
Handle Input/Output (Service 1)	EC2 t2.small
Get Random Fact (Service 2)	EC2 t2.small
Get Fact Description (Service 3)	EC2 t2.small
Get Image URL (Service 4)	EC2 t2.small
Get Image from Storage (Service 5)	EC2 t2.small
Serverless	
Handle Input/Output (Lambda 1)	Lambda Function
Get Random Fact (Lambda 2)	Lambda Function
Get Fact Description (Lambda 3)	Lambda Function
Get Image URL (Lambda 4)	Lambda Function
Get Image from Storage (Lambda 5)	Lambda Function
Relational database	
MySQL DB	RDS db.t2.small
PostgreSQL	RDS db.t2.small
Load Balancer	ELB Load Balancer
File Storage (Images)	S3 Bucket

Table 3.2: All configured AWS products

AWS product name	computing capacity
EC2 t2.xlarge	Ubuntu Linux 64bit 4 vCPUs, 2.3 GHz, 16 GiB memory
EC2 t2.small	Ubuntu Linux 64bit vCPUs, 2.5 GHz, 2GiB memory
RDS db.t2.small	Multi-AZ Deployment 1vCPU 2GiB memory 50GB storage
S3 Bucket Storage	20GB storage
Lambda	512MB memory

Table 3.3: AWS product name and corresponding computing capacity

Application	Git repository
jMeter configuration	https://bitbucket.org/tugraz-thesis/jmeter

Table 3.4: Git repository with jMeter configuration

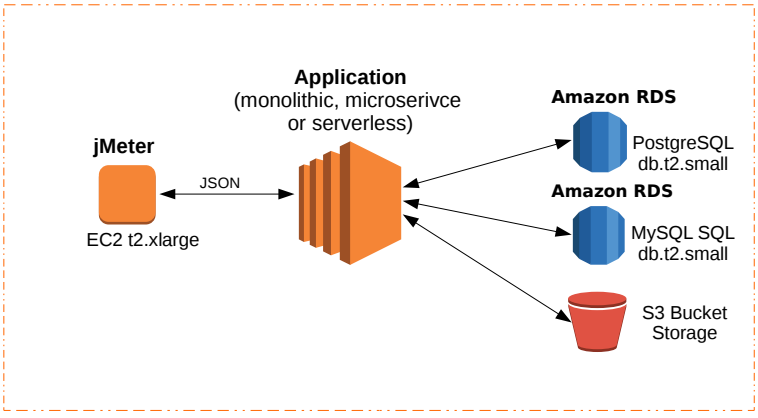

Figure 3.1: Performance test scenario with jMeter.

4 Result

In this solution chapter, I implemented the three different software architectures: monolithic, microservice and serverless. Therefore I created a demonstration application in each software architecture. The goal is to deliver the same business value in different architectures.

4.1 Demo Application

The demo application demonstrates a real business scenario. I implemented following business case: A person is interested in cryptocurrencies and can ask the voice command device *Amazon Alexa*[1] to tell a fact about cryptocurrencies.

For example, the user ask the following question: "Alexa, tell me a crypto fact."

Then Alexa answers with following example answer: "Small fractions of bitcoins are called Satoshi."

Figure 4.1 illustrates the usage of the demonstration application. The voice command device Alexa recognize the question "Alexa, tell me a crypto fact" and invokes the implemented Java Application. The Java Application has multiple HTTP endpoints and accepts JSON data as input. In the Appendix you can find an example request in the JSON format. The Java Application achieve the fact in the relational database MySQL and PostgreSQL. In addition, the URL of a fact image is added to the response and the entire answer is sent back to the Alexa device as JSON. Finally, the Alexa device

[1] https://developer.amazon.com/alexa

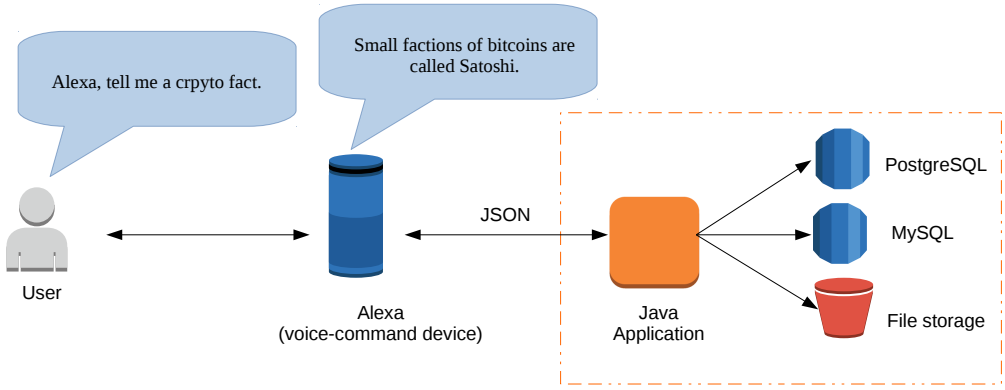

Figure 4.1: Overview Demo Application

translates the fact into voice and the user can hear a random cryptocurrency fact.

This explained usage of Alexa is an additional feature and does not come with the standard Alexa device. Amazon allows developers to expand the voice device with additional skills. In the demonstration application, I add the Alexa skill to tell a fact about cryptocurrencies. Alexa offers an interface for new skills. The Interface is free, but the developer is responsible for hosting the application and therefore infrastructure costs are incurred. In the next chapter, I analyze the infrastructure costs for the Alexa skill application.

4.2 Infrastructure Costs

Infrastructure costs can be only analyzed if you know the software requirements. For example, if you know that you have one request per hour for your application then you do not need a powerful and expensive infrastructure.

The requirements have an impact on infrastructure costs as follows:

1. **New Alexa skill in the Amazon skill store**
   The Alexa skill store is free and therefore no infrastructure costs are charged.
2. **Web application**
   The web application has to run on a web server. The webs server allocates computing resources and therefore the cloud provider charges costs.
3. **Persistent data in PostgreSQL**
   The number of records influences the PostgreSQL database server infrastructure costs.
4. **Communication format is JSON**
   The size of the JSON data influences the processing on the web server. The more data the more processing time and this leads in allocating more computing resources. Allocating computing resources results in infrastructure costs.
5. **Persistent data in MySQL**
   The number of records influences the MySQL database server infrastructure costs.
6. **Images on file storage**
   The number of images and the file size of each image influence the infrastructure costs.
7. **High availability**
   For high availability, the same application runs on multiple servers. Each replica allocates computing resources and this ends up in more infrastructure costs.
8. **SSL certificates**
   A web server has to be configured and run with a valid SSL certificate, but these maintaining costs are not investigated in this thesis.
9. **Response time, latency time and throughput**
   Server instances must have enough computing capacities to fulfill these performance metrics. More computing capacities lead to more costs.
10. **Scalability**
    A finer granularity of scalability results in more server instances and these impacts on more infrastructure costs.

The demo application delivers the business value retrieving cryptocurrency

Requirement	Infrastructure cost analysis
1. Amazon skill store	no
2. Web application	yes
3. PostgreSQL database	yes
4. JSON communication	yes
5. MySQL database	yes
6. File Storage	yes
7. High Availability	yes
8. SSL certificate	no
9. Response time	yes
10. Throughput	yes
11. Scalability	yes

Table 4.1: Requirements and impact on infrastructure costs

facts via a new Alexa skill. As you can see the mentioned requirements have an impact on infrastructure costs. In table 4.1 I summarize up the requirements and which of them are influence the infrastructure costs. In the thesis, I analyze the requirements which are marked with 'yes' in the column infrastructure costs'.

4.3 Monolith Architectural Pattern

A monolithic application packs all services in one single application.

4.3.1 Overview

In the first step, I broke down the requirements in section 1.3.1 into following 5 services:

- **Service 1 (S1)**: Handle JSON input and JSON Output
- **Service 2 (S2)**: Get the total amount of facts and chose a random fact.
- **Service 3 (S3)**: Get the fact description in PostgreSQL

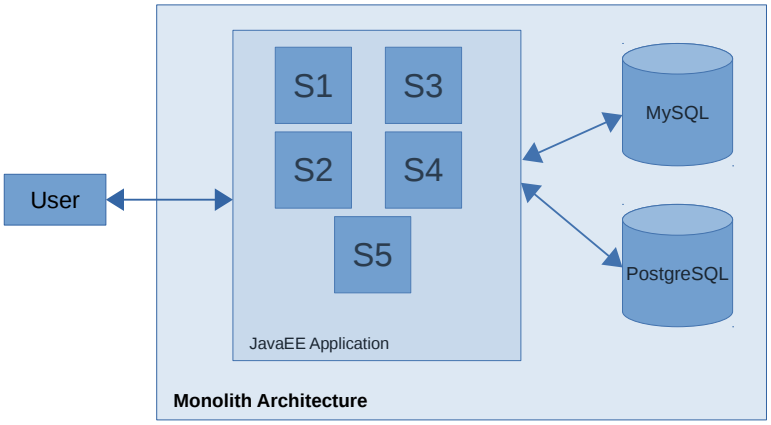

Figure 4.2: Monolith Application Overview

- **Service 4 (S4)**: Get the image URL in MySQL
- **Service 5 (S5)**: Get the image from the file storage system

Next, I set up a Java web application project with the *Google Web Toolkit (GWT)*. The application runs in one process on the application Server *Wildfly*.

You can see an architecture overview in figure 4.2. All services are packed into one big application on one server.

At the current state, the monolithic architecture has a single point of failure architecture. The web server, the two database server, and the web application run on one single server. If the server crashes, currently no redundancy exists. Even if one service crashes, for example, S1, the entire system does not work anymore.

Therefore the production environment has additional high availability configuration. As you remember, the requirements in chapter 1.3.1 define a redundancy system.

A monolith application gets redundant if you put the same application behind a load balancer on a second server. Figure 4.3 shows the monolith application in the production environment. The two database servers are

duplicated in a cluster and so you gain also redundancy for the databases. As you can see a high availability system increases the infrastructure costs, because you need a load balancer, a second server, and a clustered database server.

As you can see, you put the entire application with all services on the second server. This gives you less flexibility for scalability. For example, if only S1 has too much load, you cannot put only this service on a new machine. You have to put the entire application to a second machine, even the services with less load. This increases infrastructure costs because each copy of the whole application allocates servers resources.

In the next step, I set up the production environment on the cloud provider Amazon AWS.

4.3.2 Development on Amazon AWS

According to AWS services, the production system is shown in figure 4.4. The AWS load balancer is named with *Elastic Load Balancing (EBS)*. The two servers are virtual machine instances and are called *EC2 t2.xlarge*. The redundant database instances are called *RDS db.t2.small*. The entire setup is configured in the Amazon AWS Region *EU Irland (eu-west)*. Each Region is a separate geographic area. Each region has multiple, isolated locations known as *Availability Zones*.[2] If one or all instances fail in availability Zone A, then the instance in Zone B handle the requests and vice versa. The images itself are not stored on the local file system of the virtual server instance. They are separated from the application and can be retrieved via *Amazon S3 Bucket Storage*. Each image has a unique HTTP URL on the bucket storage and the image URL is saved in the MySQL database. Amazon offers for database an instance called *Amazon Relational Database Service (RDS)*. They are redundant and located in multiple Availability Zones. The PostgreSQL database keeps the fact description and the MySQL database stores to corresponding HTTP URL for the images. The image itself is retrieved with this URL from the file system on the Bucket Storage. This separation from

[2] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html

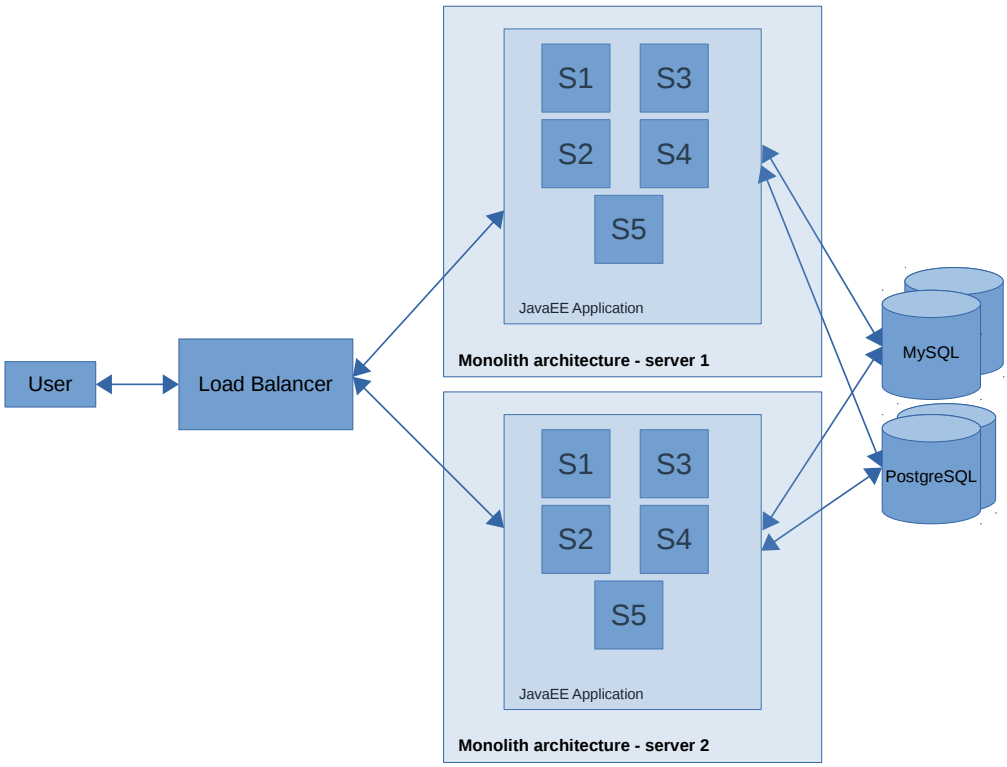

Figure 4.3: Monolith Application Architecture and High Availability

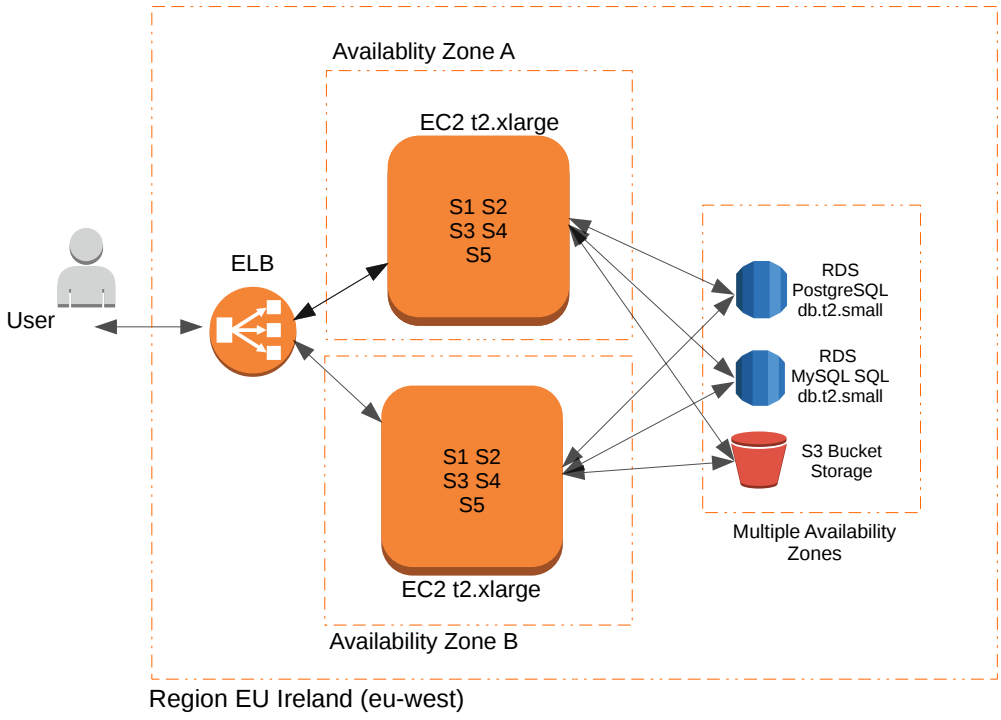

Figure 4.4: Monolith Application Deployment Architecture on AWS

the local server instance file system is necessary because we want to have "stateless" services and there should be no affinity to the underlying server instances.

In the next step, I analyze the costs at the cloud provider Amazoan AWS.

4.3.3 Amazon AWS Infrastructure Costs

In table 4.2 you can find all the detailed infrastructure costs for the monolithic approach. The two server instances cost 204.48$ per month and the load balancer 18.14$ per month. The file storage for the images costs 3.56$ per month. Amazon does not charge any data transfer costs from the storage

instances to Alexa. These are fix costs and if you have no user requests you still have to pay these fix costs. The variable costs for the load balancers increase with the number of requests. As in the requirements (chapter 1.3.1) mentioned, the maximum throughput is 10 requests per seconds. The table lists the maximum variable costs with 5.76$ per month according to the current price list.

The described monolithic application has fix costs of 488.10$ per month in total. The number of requests per minute influences the variable costs. The more requests the more variable costs. In case of the maximum throughput of 10 requests per second, you have costs of **493.86$ per month** in total.

This cost analysis shows the main problem with monolithic architectures. You have already fix infrastructure costs even without any user requests. A monolithic application is not an agile architecture, because you always put the entire application to a new instance behind a load balancer. This does not give enough flexibility for scaling because you cannot put only one service on the new server instance. This ends up in increasing unnecessary infrastructure costs.

In the next chapter, I examine the more agile approach *Microservices* to reduce infrastructure costs.

4.4 Microservice Architectural Pattern

The microservices approach decomposes the entire application into smaller parts. The smaller parts can develop, test, scale, operate and upgrade individually.

4.4.1 Overview

The monolith approach in chapter 4.3 defines the 5 services, but they are still in one big application.

- **Service 1 (S1)**: Handle JSON input and JSON Output
- **Service 2 (S2)**: Get the total amount of facts and chose a random fact.

AWS service	costs/ hour (USD)	costs/ month (24h*30d) (USD)	amount	costs/ month (USD)	total costs/ month (USD)
EC2 Linux Instance t2.xlarge (4 vCPUs, 2.3 GHz, 16 GiB memory)	0.142	102.24	2	204.48	
ELB Load Balancer fix costs	0.0252	18.14	1	18.14	
Amazon RDS PostgreSQL Multi-AZ db.t2.small (1vCPU, 2GB)	0.078	53.82	2	107.64	
Postgre SQL 50 GB SSD Storage (0.253/GB)		12.65	2	25.30	
Amazon RDS MySQL Multi-AZ db.t2.small (1vCPU, 2GB)	0.072	51.84	2	103.68	
MySQL 50 GB SSD Storage (0.253/GB)		12.65	2	25.30	
Amazon S3 Bucket Standard Storage 10GB size 200 GET requests/second (no data transfer charge to Alexa)			1	3.56	
					= 488.10
ELB Load Balander max. variable costs 200 requests/second	0.008	5.76	1	5.76	
					= 493.86

Table 4.2: AWS monolithic infrastructure costs

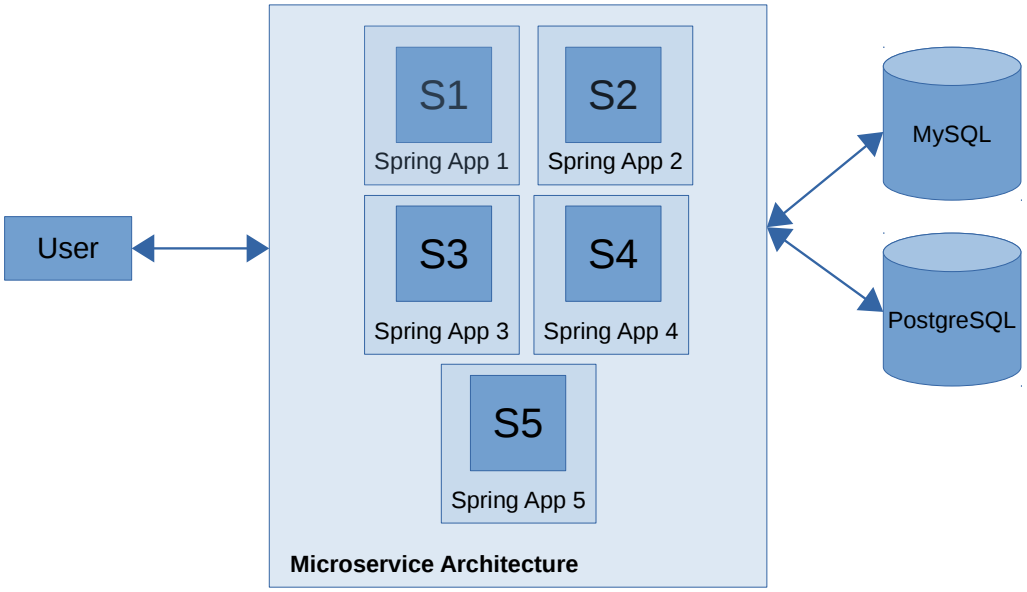

Figure 4.5: Microservice Architecture Overview

- **Service 3 (S3)**: Get the fact description in PostgreSQL
- **Service 4 (S4)**: Get the image URL in MySQL
- **Service 5 (S5)**: Get the image from the file storage system

Therefore I implemented each service as a *Spring Boot* Java Application. In figure 4.5 you can see an overview of the microservice architecture and the 5 different Java Applications. These 5 individual Java applications give you more flexibility for scalability. If only S2 has too much load, then you can just put this service on a new and stronger instance. In comparison to the monolith approach, the microservice approach gives you more flexibility for scalability. So you can reduce infrastructure costs.

The microservice approach gives you the flexibility to choose the granularity of scalability:

- Finest granularity: put each service on an individual server instance.
- Coarsest granularity: put all services on one server instance shown in figure (same as a monolithic approach, but with the possibility to

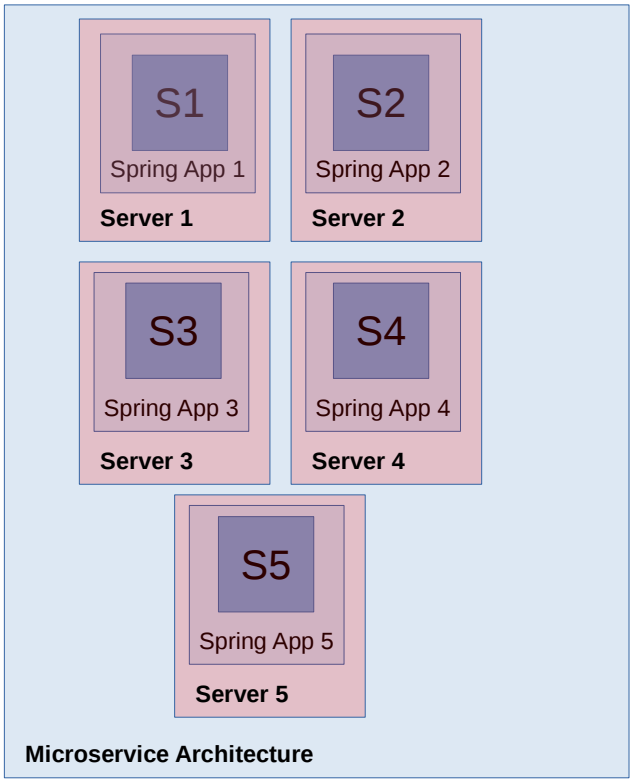

Figure 4.6: Microservice Architecture and Higher Granularity

detach services from the entire application)

In figure 4.6 you can see the highest flexibility for scaling and in figure 4.7 the lowest flexibility. Each service is an individual Java application and can be moved to another server instance anytime. Each Java application runs as an own process and holds its own data. Remember, a microservice can implement, deployed and tested independently. In a monolithic architecture, you only have one big process and you do not have this flexibility. In the demo application, I chose the finest granularity and put each service on an individual server instance as shown in 4.6.

In regards to the demo application requirements (1.3.1) the microservice

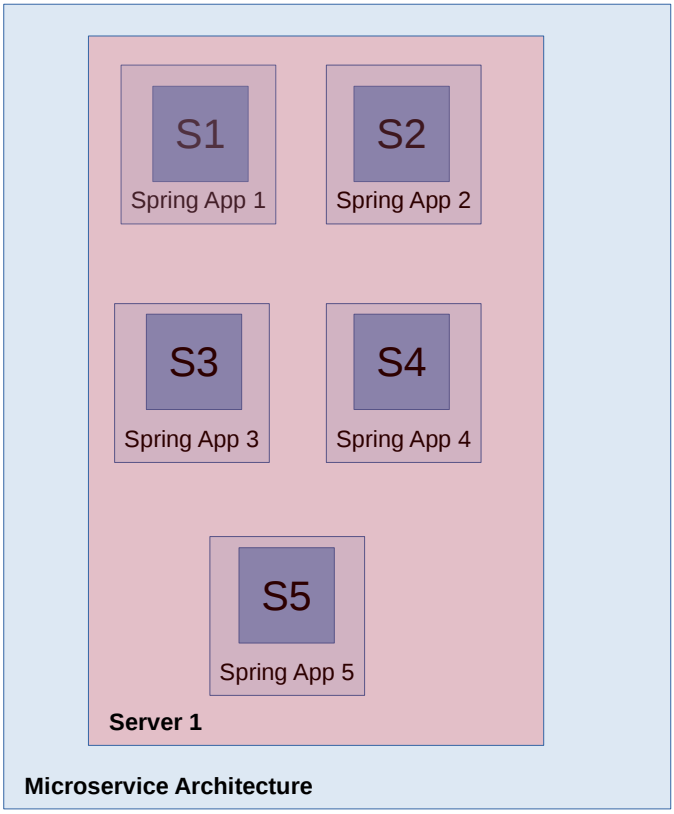

Figure 4.7: Microservice Architecture and Lower Granularity

architecture does not fulfill all requirements yet. The demo application does not achieve the high availability requirement yet. For example, if one service fails, the entire application is down because the service is not redundant yet. It does not matter if the service runs on its own server instance or if the service runs together with several services on the same server instance. Currently, all services are single points of failure. Therefore, I use a load balancer and put the same service behind the load balancer again. It is the similar concept as in the monolithic architecture. The difference is the size of the application behind the load balancer. In the microservice architecture, the service is an individual application it represents a small part of the entire application. In comparison to the monolithic architecture, the application behind the load balancer represents the entire application.

As you can see a microservice architecture deals with many individual services. In case of no redundancy, you have to handle 5 different services in the demo application. If you also add high availability, then you duplicate each service behind a load balancer and this ends up in 10 individual applications in total. In addition, each service communicates with each other via JSON. As you remember, the requirements in chapter 1.3.1 define, that the communication format between services is JSON. One of the basic principles is the usage of a *Microservice Gateway* and *Microservice Registry*.

The Microservice Gateway is the single entry point for user requests. Each user request is routed to the appropriate service application. The gateway also verifies if the client is authorized to perform the request.

The Microservice Registry is an essential component of the microservice architecture. The registry ties all components together and enables them to communicate with each other. The registry tells the gateway which microservice are available.

In figure 4.8 you can see how the components interact with each other. The user only sends a request to the gateway. If the user is authorized for the requested service, the gateway forwards to request to the service. Each service has to be registered on the registry and the gateway communicates with the registry to get all the information for the service.

In the microservice demo application, the gateway and the registry are another Spring Boot application. Both of them are deployed on an individual

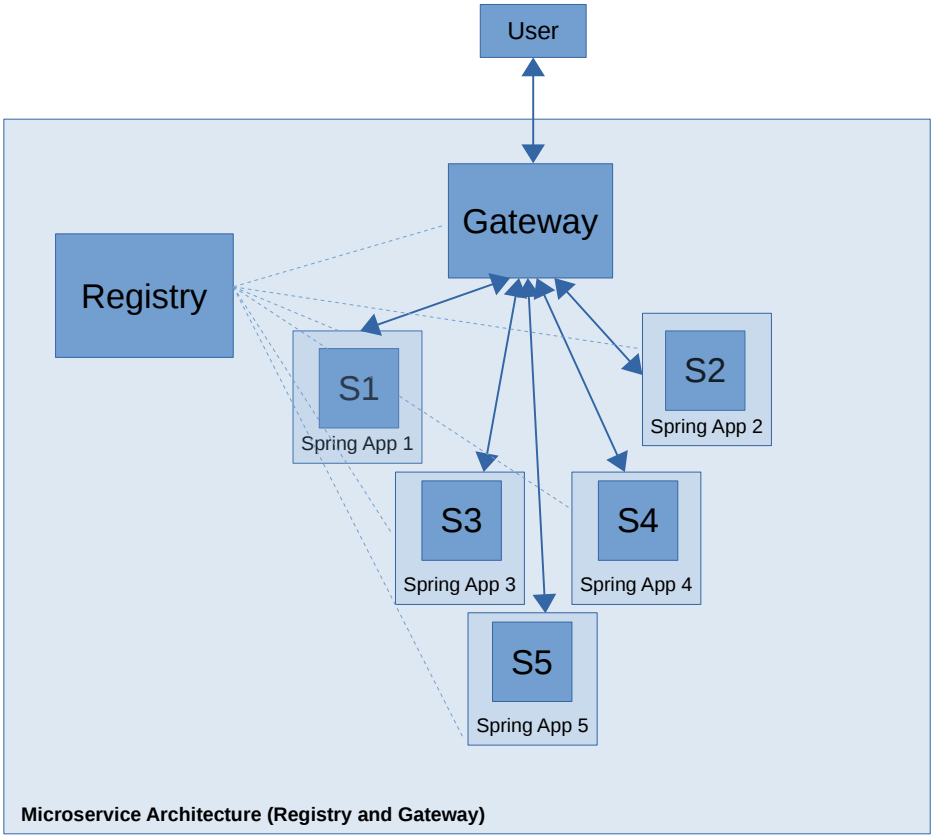

Figure 4.8: Microservice Architecture with Registry and Gateway

server instance because they are essential components of the microservice architecture.

Figure 4.8 shows almost a best practice example of a microservice architecture. Only almost because the high availability requirement is still missing. Each service is not redundant and represents a single point of failure. Even the core components registry and gateway are not redundant.

Finally, I gain high availability by putting a second gateway behind a load balancer, adding a second registry and duplicating each spring application. register each spring application twice on the registry. In figure 4.9 no single point of failure exists anymore.

Adding high availability increases infrastructure costs because you need for the gateway and registry additional individual server instances.

Quick overview about the server instances in figure 4.9

- 2 individual server instances for the 2 gateways
- 2 individual server instances for the 2 registries
- At least 2 server instances for all 5 microservice applications: It depends on the chosen granularity how many services run on one individual server instance. If you chose the lowest granularity, then you put all 5 services on one server instance. Due to the high availability, you need a second server with the same 5 services. That is the reason for at least 2 servers for the service applications.

The production environment is set up on the cloud provider Amazon AWS.

4.4.2 Development on Amazon AWS

According to AWS services, the production system is shown in figure 4.10.

The Amazon load balancer *ELB* redirects each request either to the gateway in *Zone A* or to the gateway in *Zone B*. These requests are pink-marked. As you can remember the requirements in chapter 1.3.1 define a high availability system. Amazon uses different Availability Zones. If you put all you server instances in just one Zone, for example, Zone A, Amazon does

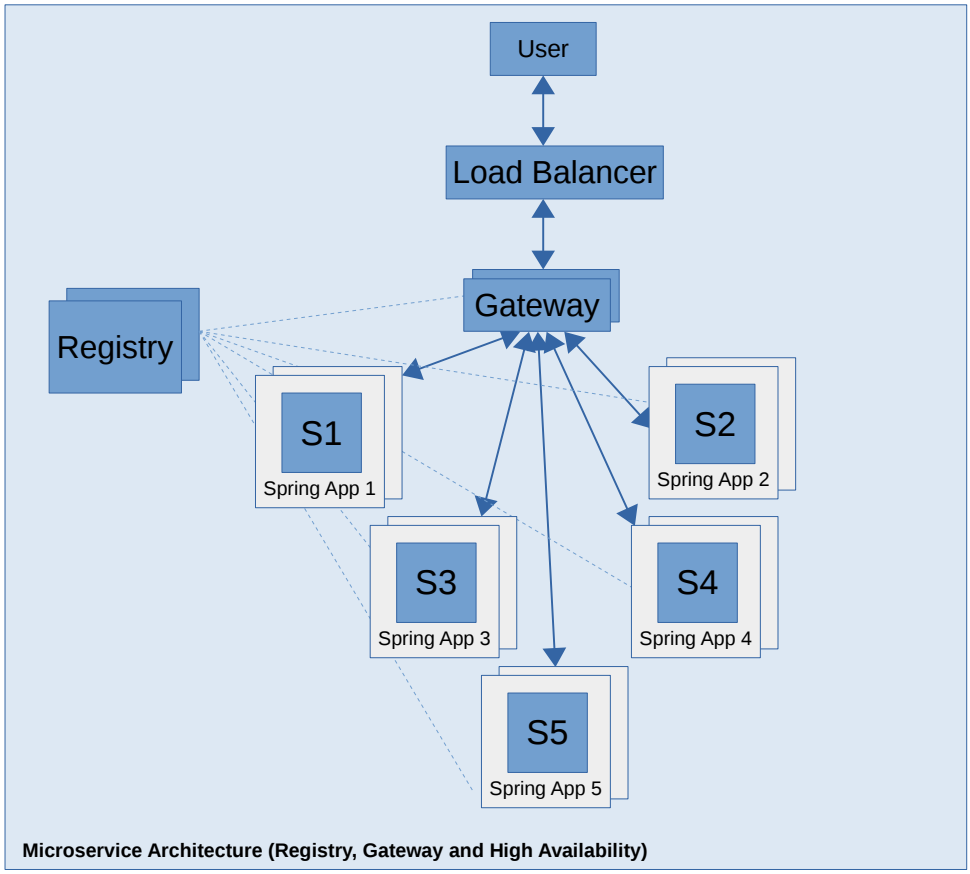

Figure 4.9: Microservice Architecture with Registry, Gateway, and High Availability

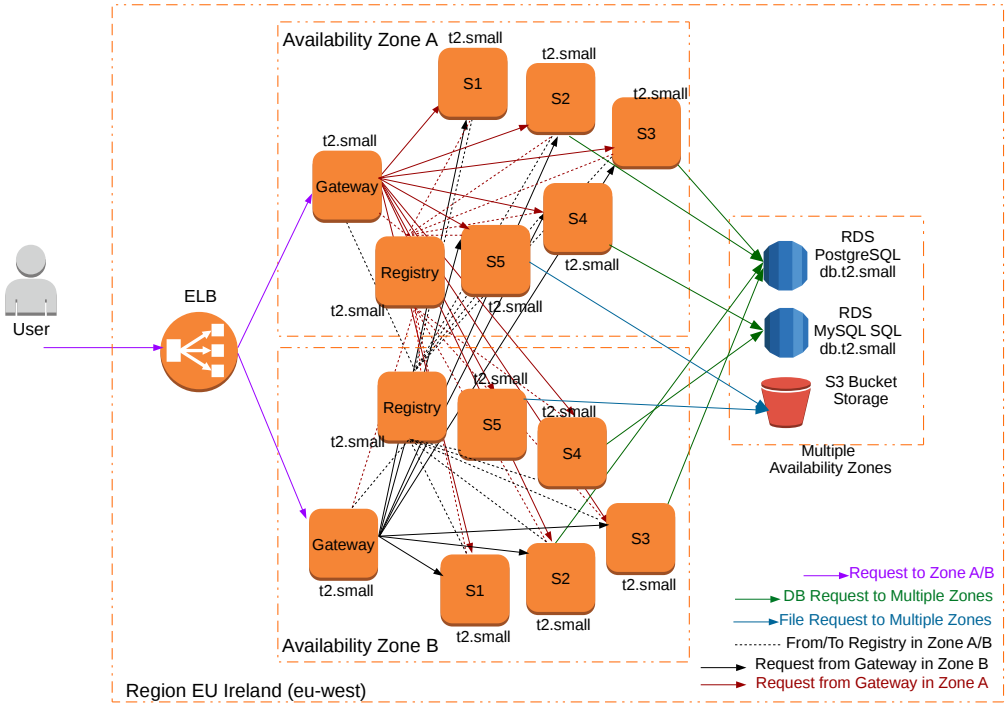

Figure 4.10: Microservice Deployment Architecture with Registry, Gateway and High Availability on AWS

not guarantee high availability. It does not matter how many redundant server instances you put behind the load balancer in the same zone. If they are all in the same Zone A you do not have high availability because the entire Zone could be down and all your server instances are not available anymore. Therefore, each component runs on a second server instance in Zone B. Adding new server instances always increases the infrastructure costs.

The gateway is the main entry point in the microservice architecture and forwards each request to the appropriate service. The gateway itself runs on an individual server instance *Amazon EC2 t2.small* in Zone A and in Zone B. Both gateways have exact the same configuration and knowledge of all available services. If one gateway crashes, the other gateway would take over the entire work. All requests from the gateway in Zone A to any service are red-marked and all requests from the gateway in Zone B to any service are black-marked. The gateway forwards the request either to a service in Zone A or to a service in Zone B. As you can see the gateway is an essential component in a microservice architecture and has to handle all user requests.

The registry is another essential component in the microservice architecture. The registry tells the gateway which service is available. Therefore, the registry has to communicate with all services in Zone A and Zone B. The communication is marked with a dashed black line. Both registries have the same configuration. If one registry crashes the other registry would take over the entire work.

Each service (S1 to S5) is an individual Java application. In regards to analyzing the maximum infrastructure costs, I chose the finest granularity. This means each application runs on an individual server instance. Each server instance allocates additional resources and increases the infrastructure costs. In total 10 server instances are necessary. Each service allocates 2 server instances. One instance in Zone A and one instance in Zone B. For example, if the server instance with S1 crashes another instance with S1 will deal all relevant S1 requests. One server instance is in Zone A and another server instance is in Zone B.

Amazon offers for databases an instances called *Amazon Relational Database Service (RDS)*. I use two instances of a MySQL database and two instances

of a PostgreSQL database. They are redundant and located in Zone A and Zone B. As you remember in chapter 4.3 I decompose the requirements into 5 services. Only the service 4 needs to establish a database connection to the MySQL server. The PostgreSQL server is contacted from service 2 and service 3. The remaining services do not have persistent data in relational databases and therefore no database connection is necessary. All database connections are green-marked.

In a microservice architecture, you have the flexibility to provide a database connection to individual services instead of the entire application. This gives you more flexibility for scaling. Let us assume the application is under heavy load, for example, 200 requests per second. The monitoring tools show that service 4, which deals with the MySQL connection, is overloaded and the server instance *t2.small* has not enough computing resources to handle all requests. The remaining services work fine and can handle the high load. So you just have to give the service 4 application more computing resources. In the demo application, you can add a third *t2.small* instance, install the service 4 application and register the service on the registry. The registry tells the gateway about a third service 4 application and so you increased the computing resources just for the service 4. This flexibility reduces infrastructure costs because you do not have to duplicate the entire application such as in a monolithic architecture. As I mentioned the remaining services work fine and there is no need to allocate also computing resources for them. Therefore a microservice architecture reduces infrastructure costs.

In the next step, you can find a detailed costs analysis for the microservice architecture.

4.4.3 Amazon AWS Infrastructure Costs

All the costs for microservice architecture are shown in table 4.3. In comparison, the monolithic architecture has server instance costs of 102.24$ per month for the entire application (4.2). In the microservice architecture, the additional small server instance for service 2 costs only 12.96$ per month.

AWS service	costs/ hour (USD)	costs/ month 24h*30d (USD)	amount	costs/ month (USD)	total costs/ month (USD)
ELB Load Balancer fix costs	0.0252	18.14	1	18.14	
EC2 Linux Instance Microservice Gateway t2.small (1 vCPUs, 2.5 GHz, 2 GiB memory)	0.018	12.96	2	25.92	
EC2 Linux Instance Microservice Registry t2.small (1 vCPUs, 2.5 GHz, 2 GiB memory)	0.018	12.96	2	25.92	
Amazon RDS, Postgre SQL Multi-AZ Deployment db.t2.small (1vCPU, 2GB)	0.078	53.82	2	107.64	
Postgre SQL 50 GB SSD Storage (0.253/GB)		12.65	2	25.30	
Amazon RDS, MySQL Multi-AZ Deployment db.t2.small (1vCPU, 2GB)	0.072	51.84	2	103.68	
MySQL 50 GB SSD Storage (0.253/GB)		12.65	2	25.30	
EC2 Linux Instance 5 services with high availability t2.small (1 vCPUs, 2.5 GHz, 2 GiB memory)	0.018	12.96	10	129.60	
Amazon S3 Bucket Standard Storage 10GB size 100 GET requests/seconds		3.56	1	3.56	
					= 465.06
ELB Load Balancer max. variable costs 10 requests per second	0.008	5.76	1	5.76	
					= 470.82

Table 4.3: AWS microservice infrastructure costs

monolith costs (USD)			microservice costs (USD)			difference (USD)
fix costs	variable costs	total costs	fix costs	variable costs	total costs	total costs
488.10	5.76	493.86	465.06	5.76	470.82	**23.04 (4.67%)**

Table 4.4: Total infrastructure costs comparison between monolith and microservice

The total costs for the microservice architecture are **470.82$** per month. In comparison to the monolithic architecture, the total costs are **493.86$** per month. The analysis shows that the microservice total infrastructure costs are **4.67%** lower in comparison to the monolithic architecture. You can see the summary in table 4.4. The microservice approach has no impact on the variable costs for the load balancer because the load balancer has to be dimensioned for max. 200 requests per seconds. If you reach this amount of requests Amazon charges 46.08$ per month. It does not matter what architecture you have behind the load balancer.

The detailed costs for the microservices architecture are as the following: The fix costs for the load balancer and the database instances are equal to the costs of monolithic architecture. The difference is the smaller server instances for the microservice itself. Each microservice run on an individual *EC2 Linux instance t2.small* and each instance costs 12.96$ per month. You have 5 server instances without high availability. According to high availability, you run the same service on a second server instance in a different Zone. Therefore, you have 10 server instances in total for the microservices. All the 10 microservices instances cost 129.60$ per month. The redundant MySQL database server instances with 50GB storage cost 128.98$ per month (103.68$ + 25.30$) and the redundant PostgreSQL database server instances with 50GB storage cost 132.94$ per month (107.64$ + 25.30$). As I mentioned, the microservices architecture needs a gateway and a registry. Both of them use an *EC2 Linux instance t2.small* and therefore you have total costs of 51.84$ per month (4 x 12.96$). Gateway and registry are duplicated in another zone and this ends up in 4 instances in total. The image retrieving from the file system is realized with the *S3 Bucket* instance. The redundant storage instance in multiple Availability Zones costs 3.56$ per month. As you remember from

the monolithic approach Amazon does not charges any data transfer costs from the S3 Bucket to Alexa.

All these mentioned costs are fix costs. You always have to pay them monthly, even without any user requests to your application. Except for the variable costs for the load balancer. The variable costs are the same as the for the monolithic architecture with 5.76$ per month.

As you can see the microservice infrastructure fix costs are **4.67%** lower in comparison to the monolithic architecture. In addition, you get more flexibility for scaling. You can move any service anytime to a new server instance. Therefore you do not have to allocate new server resources for the entire application, just for the service itself. You can safe infrastructure costs with a microservice architecture because it is a more agile approach and gives you more flexibility for scaling. However, you still have the problem with fix costs. Therefore, in the next chapter, I try to reduce the fix costs with a serverless architecture.

4.5 Serverless Architectural Pattern

The goal is to avoid fix costs from the previously explained monolithic and microservice architecture.

4.5.1 Overview

The result of the cost analysis for monolithic and microservice architecture shows that the server instances *EC2 Linux instance t2.small* and *EC2 Linux instance EC2 t2.xlarge* are the biggest block of fix costs.

A serverless architecture allows you to deploy the demo application without thinking about server instances. That means you just can focus on your application and the cloud provider takes care of the server instances.

As you can remember the demo application consists of the following 5 services:

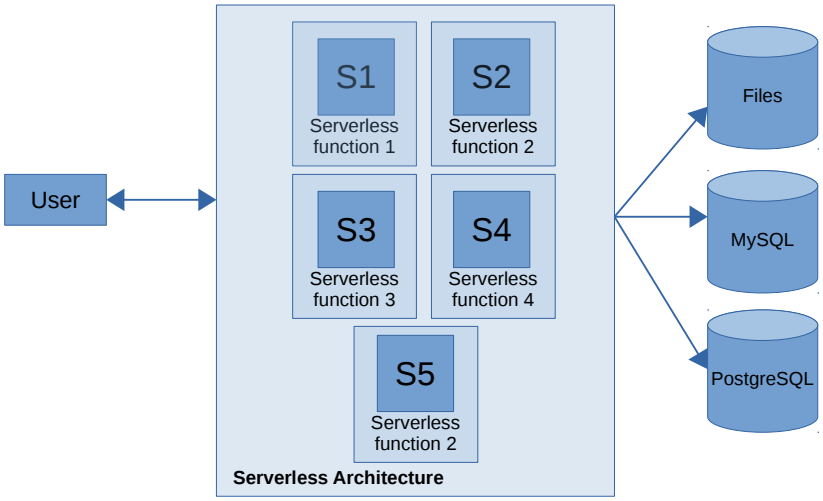

Figure 4.11: Serverless Architecture Overview

- **Service 1 (S1)**: Handle JSON input and JSON Output
- **Service 2 (S2)**: Get the total amount of facts and chose a random fact.
- **Service 3 (S3)**: Get the fact description in PostgreSQL
- **Service 4 (S4)**: Get the image URL in MySQL
- **Service 5 (S5)**: Get the image from the file storage system

In a serverless architecture, you can deploy each service independently without thinking about the underlying server instances. This influences also the charging model. The cloud provider only charges costs if you actually use the service.

Figure 4.11 shows an overview of the serverless architecture.

The services S1 to S5 are independently deployed without creating underlying server instances. In a serverless architecture, an individual application is called a *serverless function*. All 5 services are deployed as a serverless function.

I implement each serverless function as an individual Java application and run each of them as a serverless function. The difference to the Java application in the monolithic and microservice architecture is the complexity of the Java Application. As you remember the monolithic applications use

the Google Web Toolkit Framework. The microservice architecture uses the Spring Boot Framework. Here the serverless function is kept simple with few dependencies. A serverless function does not use complex frameworks such as Spring Boot or GWT. The reason is that you need a Java application which loads quickly. The cloud provider will start your serverless function on behalf of users requests. In a serverless architecture, you will never have services which run all the time. Each service is started on behalf of requests and therefore you only pay what resources you actually use. Therefore, you can avoid server instances and reduce fix infrastructure costs. Each serverless function runs only for the duration of user requests. A serverless function starts and stops automatically. The cloud provider creates the underlying server instances automatically and only charges you for the duration of the running services to handle the user inquiry.

As you remember we outsource the image storage from service 5. Images are not stored on the local file system, they are stored on an individual instance. Here you can see the reason for this outsourcing. If you bundle all your images with the corresponding serverless function, then the function cannot be started on behalf of requests quickly enough. Assume you have stored around 1,000 images with size 1MB each image. So you have local file storage of 1GB in total and for each user request you have to set-up this storage or copy the files from central storage. You could not handle user requests quickly enough. Therefore all images stored on an individual file storage instance outside of the serverless architecture. This persistent file system storage is shared across all serverless functions. A serverless function never holds its own data. Each serverless function is a stateless function.

A stateless function does not hold any database data as well. All database data are stored outside from the serverless architecture. The MySQL and PostgreSQL database run on its own server instance and can be accessed from each serverless function. Imaging each serverless function starts their own database with each request. This would lead to a long start time for each service. Therefore lambda function never holds any databases and data are always stored in an individual instance. The database instances can be accessed from all serverless functions.

Another benefit of using a serverless architecture is the high availability. The

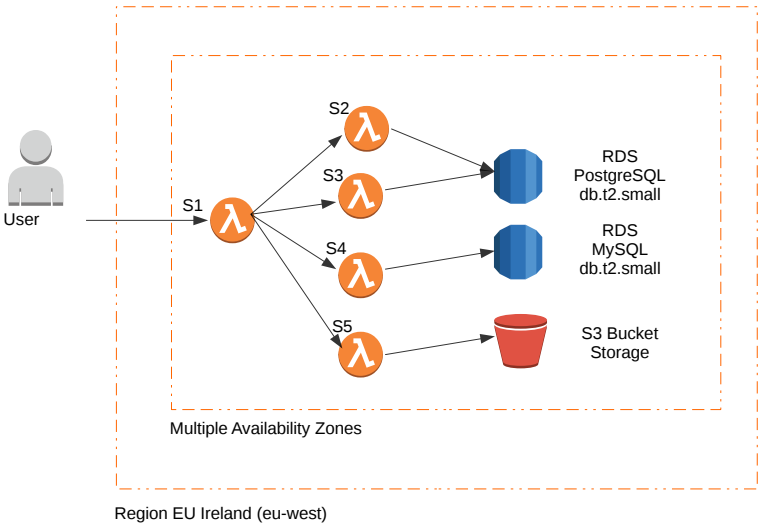

Figure 4.12: Serverless Architecture Deployment on AWS

cloud provider takes also care about redundancy and offers high availability for your serverless functions.

In the next step, I deploy the demo application with a serverless architecture on the cloud provider Amazon AWS.

4.5.2 Development on Amazon AWS

The cloud provider Amazon AWS named their serverless architecture *AWS Lambda*. As I mentioned, an application which runs in a serverless architecture is called a serverless function. Amazon named a serverless function *AWS Lambda function*.

Finally, I run each service of the demo application as a Lambda function. The deployment is shown in figure 4.12.

The 5 individual lambda functions are the core of the serverless architecture deployment on AWS. Each lambda function represents a service from the demo application. As you remember service 1 deals with the user's input and output via JSON format. Therefore, the service 1, implemented as a

lambda function 1, stands for the entry point of the user. Lambda function 1 does not perform the entire application logic alone. The application logic is spread across further 4 lambda functions. The lambda functions 2 and 3 retrieve data from the PostgreSQL database. The lambda function 4 get data from the MySQL database and the lambda function 5 reads images from the file system storage. The figure 4.12 shows the application logic in a more detailed view.

All functions are synchronously invoked. The lambda function 1 is the access point for the user. Then lambda function 1 calls all other functions and function 1 waits for the response of the invoked function.

4.5.3 Amazon AWS Infrastructure Costs

In the previously mentioned serverless and microservice architecture the server instances run always and therefore the exact duration time of the application itself is not important in regards to infrastructure cost. The serverless architecture is the exact opposite. The cloud provider measures the execution time of the application and charges them.

Therefore, we have to know how often the application is executed and how long is the execution time. The requirements in section 1.3.1 define the maximal throughput (10 requests per second) and maximal response time (3000ms). We calculate the maximum costs of infrastructure costs and therefore we assume the maximal performance metrics for the entire month. The result is the maximum infrastructure costs for the entire application.

Figure 4.13 illustrates the maximum execution time of all services and their synchronous execution.

The service 1 waits for the processing time of service 2 to service 5. The total execution time of **3000ms** are linearly separated into the 4 services with an execution time of **750ms**. Of course, the actual execution time will be shorter than the maximum response time. But we calculate the maximum possible infrastructure costs and therefore we have to take the maximum execution time as well.

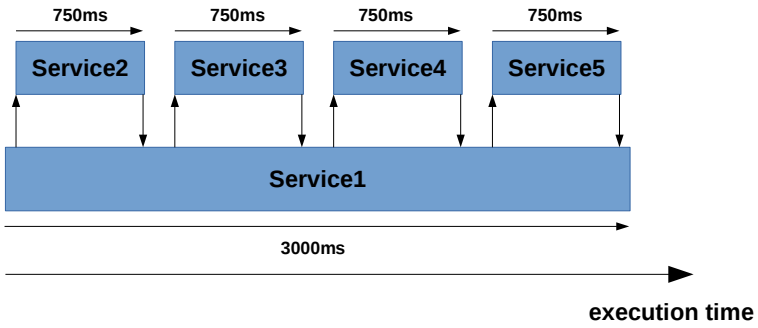

Figure 4.13: Serverless Synchronous Execution Overview

Lambda Function	Cost Factor GB/Second	Memory (GB)	Function Duration (s)	Total GB-Second (Function Duration * Memory)	Price pro request (Cost Factor * Total GB-Second)	Total request per month (=10 requests * 60* 24*30)	Total Costs (USD)
Service 1	0.00001667	2	3	6	0.00010002	432000	**43.21**
Service 2-5	0.00001667	2	0.75	1.5	0.000025005	1728000	**43.21**

Table 4.5: Lambda Function Costs

Amazon measures the execution time of each service and multiplies this time with a cost factor. The cost factor depends on the memory of the serverless function. The memory can be configured for each function. This is the maximum memory that the function can allocate. In the demo application, each serverless function has a total memory of **2024MB**.

Amazon lists the prices for serverless functions as a price for 'GB-Seconds'. The Gigabyte refers to the mentioned pre- configured memory of the serverless function. And the 'Seconds' are the execution time of the serverless function.

The calculations are done with the listed GB-Seconds price of **0.00001667$** per month.

Table 4.5 illustrates the detailed costs of all serverless functions.

The service 1 is the entry point for the user's request and has infrastructure costs of **43.21$**. The service 1 has a maximal execution time of 3000ms. During this time, service 1 executes 4 further services. Each of the 4 services has a maximum execution time of 750ms. This results in total costs of **43.21$** for the services 2-5. In total, all services have infrastructure costs of **86.42$** (43.21$ + 43.21$).

Of course, you also have costs for the database and the image file storage. Table 4.6 lists all serverless infrastructure costs.

The database and file storage fix costs are the same for the monolithic, serverless and microservice architecture. In contrast, a serverless architecture does not have an explicit load balancer as a cost factor. As I already mentioned, the cloud provider already guarantees high availability for serverless functions and offers them already redundantly. Therefore, you do need a load balancer. In total the serverless architecture has variable costs of **86.42$**. This is the maximum variable costs during a throughput of 10 requests per second over the entire month.

In summary, you have total costs of **348.34$** for the serverless architecture on Amazon AWS.

4.6 Summary Infrastructure Costs

As you can see the infrastructure costs depend on the chosen architecture. Table 4.7 summarizes the cost analysis results.

The result is that a serverless architecture can reduce infrastructure cost **by 29.47%** in comparison to the monolithic architecture.

4.7 Performance Evaluation

The demo application is implemented in the three different approaches: monolithic, microservice and serverless. In this section, we evaluate whether all three approaches fulfill the performance metrics throughput, latency

AWS service	costs/ hour (USD)	costs/ month 24h*30d (USD)	amount	costs/ month (USD)	total costs/ month (USD)
Amazon RDS PostgreSQL Multi-AZ Deployment db.t2.small(1vCPU, 2GB)	0.078	53.82	2	107.64	
PostgreSQL 50GB SSD Storage (0.253/GB)		12.65	2	25.30	
Amazon RDS MySQL Multi-AZ Deployment db.t2.small(1vCPU, 2GB)	0.072	51.84	2	103.68	
MySQL 50GB SSD Storage (0.253/GB)		12.65	2	25.30	
= fix costs					**= 261.92**
Lambda Function (Service 1)				43.21	
Lambda Function (Service 2-5)				43.21	
= variable costs					**= 86.42**
= total costs					**= 348.34**

Table 4.6: Serverless Architecture - Infrastructure Costs

monolith costs (USD)			**microservice costs (USD)**				**serverless costs (USD)**			
fix costs	variable costs	total costs	fix costs	variable costs	total costs	total costs difference to monolith (USD)	fix costs	variable costs	total costs	total costs difference to monolith (USD)
488.10	5.76	**493.86**	465.06	5.76	**470.82**	**23.04 (4.67%)**	261.92	86.42	348.34	**145.52 (29.47%)**

Table 4.7: Cost Comparison

Total Amount HTTP requests	Test Duration (s)	Throughput (requests/ second)	Min Response Time (ms)	Max Response Time (ms)	Average Response Time (ms)
100	10	10	56	163	72

Table 4.8: Summary Monolith Performance

time and response time. As we defined in section 1.3.1 during a **throughput** of **10 request**s per second the **response time** has to be lower than **3000ms** and the **latency** has to be lower than **2000ms**.

4.7.1 Performance Monolithic Architectural Pattern

The results of the monolith performance test are summarized in table 4.8.

The average response time is **72ms**. During the test, the response time reaches a maximum of **163ms**.

Figure 4.14 illustrates the response time over the performance test duration. As you can see at the beginning the application process requests slower than during the remaining test. The reason for the slower processing time, in the beginning, is that the application has to create a connection to two relational databases and establish a cache. For the remaining test, the average response time is continuous around 72ms.

In the Appendix, you can find the illustration of the latency time. The latency time is nearly as equal to the response time. The reason is the small response size. The HTTP response with JSON data has approximately a size of 952 Bytes (773 Bytes body and 179 Bytes header). The image is not transferred via an HTTP request. Just the URL to the image on the Amazon S3 is transferred.

Therefore, the latency time and the response time are nearly equal. The transfer time of 952 Bytes does not make a big difference. The different is just the time to transfer 952 Bytes from the server to the client.

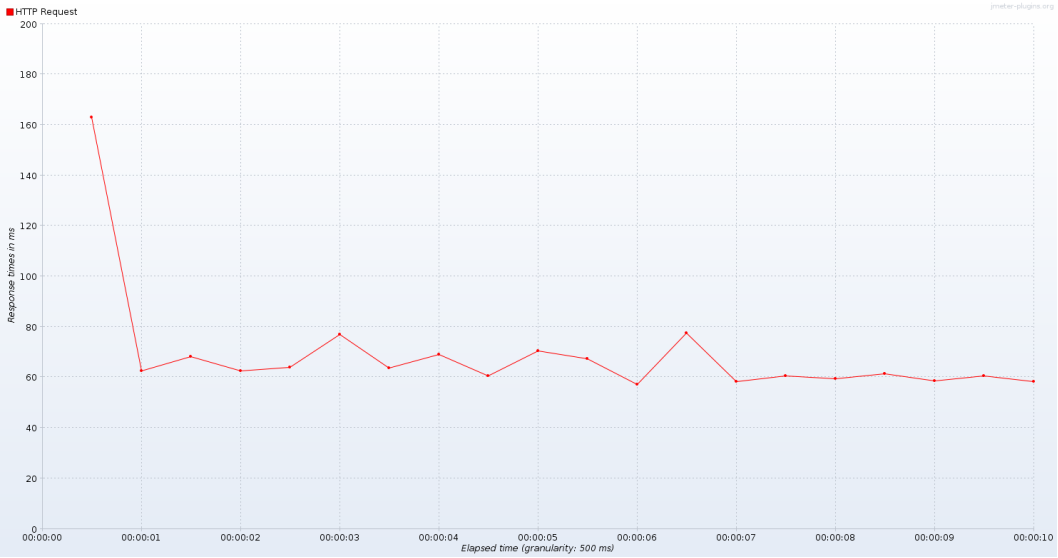

Figure 4.14: Monolith Response Time

Total Amount HTTP requests	Test Duration (s)	Throughput (requests/ second)	Min Response Time (ms)	Max Response Time (ms)	Average Response Time (ms)
100	10	10	39	86	49

Table 4.9: Summary Microservice Performance

4.7.2 Performance Microservice Architectural Pattern

The results of the monolith performance test are summarized in table 4.9.

The average response time is **49ms**. During the test the response time reaches a maximum of **86ms**.

Latency and response time are nearly equal because the size of the response is only around 952 Bytes.

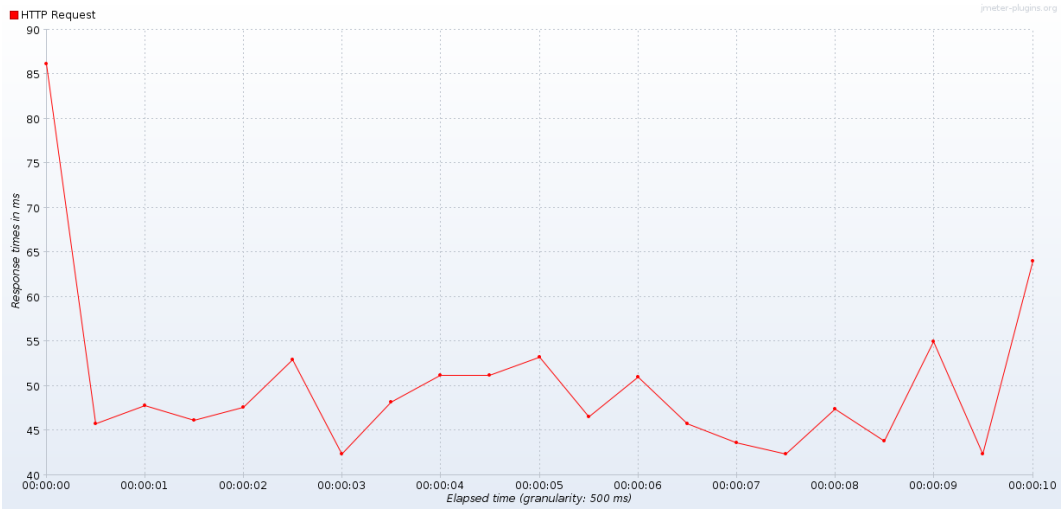

Figure 4.15: Microservice Response Time

4.7.3 Performance Serverless Architectural Pattern

The serverless performance metrics are as follows: The average response time is **216ms**. The maximum response time is 424ms.

In figure 4.16 you can see the response time over the test duration.

At the beginning the response reaches 424ms. The reason is that the serverless function is executed on-demand and the cloud provider has to set up the infrastructure on-demand. The underlying server infrastructure is provided by the cloud provider automatically. The cloud provider starts on-demand the necessary infrastructure for processing the serverless function calls. As you can see the first requests take longer than the remaining calls. But still, the remaining calls have not such a constant response time as the monolithic application in section 4.7.1. The on-demand setup from the cloud provider does guarantee such a constant response time as we can see in the monolithic approach. Each serverless function is isolated and caching mechanism from the monolithic approach does not work in the same way for serverless functions. Therefore we have more variety in the response time in comparison to the monolithic approach.

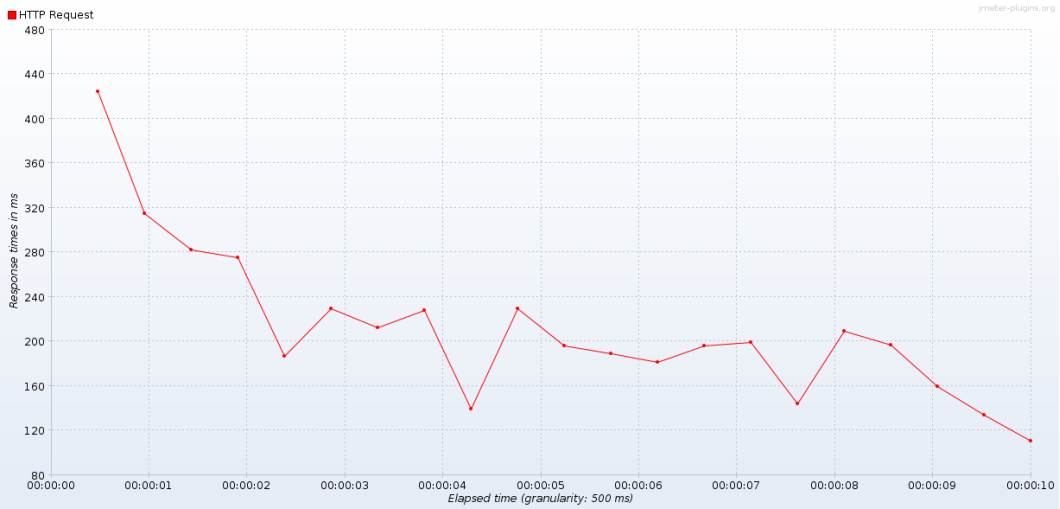

Figure 4.16: Serverless Response Time

Architecture	Average Response Time (ms)
monolithic	72
microservice	49
serverless	216

Table 4.10: Average Response Time: monolithic, microservice and serverless

Due to the small response size, the latency time is nearly equal to the response time.

4.7.4 Summary Performance Test

As you can see the performance requirements as defined in section 1.3.1 are completely accomplished by all three approaches.

In table 4.10 you can see the average response time of the monolithic, microservice and serverless approach.

5 Conclusion and Future Work

This thesis defines the ”Unix philosophy” as a basic setting for a modern software development. The result combines the ”Unix philosophy” with best practice guidelines ”Twelve-Factor App” and ”jHipster” as an evolutionary development of web-based information systems.

The result of a costs analysis shows that scalability and high availability mostly influence the infrastructure costs at the cloud provider Amazon AWS. Especially the chosen architectural pattern has a huge influence on the total infrastructure costs and idle server instances. The case study shows, that the evolutionary development environment based on a serverless architectural pattern can reduce infrastructure costs by 30 percent. The main reason is the on-demand cost model offered by the cloud provider. Therefore, an evolutionary development reduces idle server instances. It is always a good idea to think about alternatives to the monolithic architectural pattern. Further analysis shows, that infrastructure costs can be separated into fix and variable costs. Database storage and file system storage are still fixed costs in all three architectural patterns.

In further work to costs for database instances can be analyzed. Also the cloud provider Amazon AWS could be compared with other service providers in further work. The performance test acknowledges the defined throughput, latency and response time. In further work, the maximum throughput of the different approaches could be tested and compared.

Appendix

JSON Output


```
{
    "version": "1.0",
    "userAgent": "ask-java/2.3.1 Java/1.8.0_141",
    "response": {
        "outputSpeech": {
            "type": "SSML",
            "ssml": "<speak>Small fractions of bitcoins are called Satoshi.<break time=\"1s\"/>  Would you like to hear another..</speak>"
        },
        "card": {
            "type": "Simple",
            "title": "Cryptocurrency Facts",
            "content": "Small fractions of bitcoins are called Satoshi"
        },
        "reprompt": {
            "outputSpeech": {
                "type": "SSML",
                "ssml": "<speak>Small fractions of bitcoins are called Satoshi<break time=\"1s\"/>  Would you like to hear...</speak>"
            }
        },
        "directives": [
            {
                "type": "Display.RenderTemplate",
                "template": {
                    "type": "BodyTemplate3",
                    "image": {
                        "sources": [
                            {
                                "url": "https://s3-eu-west-1.amazonaws.com/alexaskillimagestorage/images/crypto1.jpg"
                            }
                        ]
                    },
                    "title": "Cryptocurrency Facts",
                    "textContent": {
                        "primaryText": {
                            "type": "RichText",
                            "text": "Small fractions of bitcoins are called Satoshi"
                        },
                        "secondaryText": {
                            "type": "RichText"
                        }
                    }
                }
            }
        ],
        "shouldEndSession": false
    }
}
```


JSON Input


```
{
   {
        "version": "1.0",
        "session": {
                "new": false,
                "sessionId": "amzn1.echo-api.session.2aa145f6-5199-4358-af45-0036f97de6f8",
                "application": {
                        "applicationId": "amzn1.ask.skill.4e9e9096-382c-499d-bae7-773054be091a"
                },
                "user": {
                        "userId": "amzn1.ask.account.xyz"
```

```
                }
        },
        "context": {
                "Display": {
                        "token": ""
                },
                "System": {
                        "application": {
                                "applicationId": "amzn1.ask.skill.4e9e9096-382c-499d-bae7-773054be091a"
                        },
                        "user": {
                                "userId": "amzn1.ask.account.xyz"
                        },
                        "device": {
                                "deviceId": "amzn1.ask.device.xyz": {
                                        "Display": {
                                                "templateVersion": "1.0",
                                                "markupVersion": "1.0"
                                        }
                                }
                        },
                        "apiEndpoint": "https://api.eu.amazonalexa.com",
                        "apiAccessToken": "xyz"
                }
        },
        "request": {
                "type": "IntentRequest",
                "requestId": "amzn1.echo-api.request.fd16dc5f-a3e7-4700-b9c5-7a8bb274ea86",
                "timestamp": "2018-07-22T22:49:20Z",
                "locale": "en-US",
                "intent": {
                        "name": "FactIntent",
                        "confirmationStatus": "NONE"
                }
        }
}
```

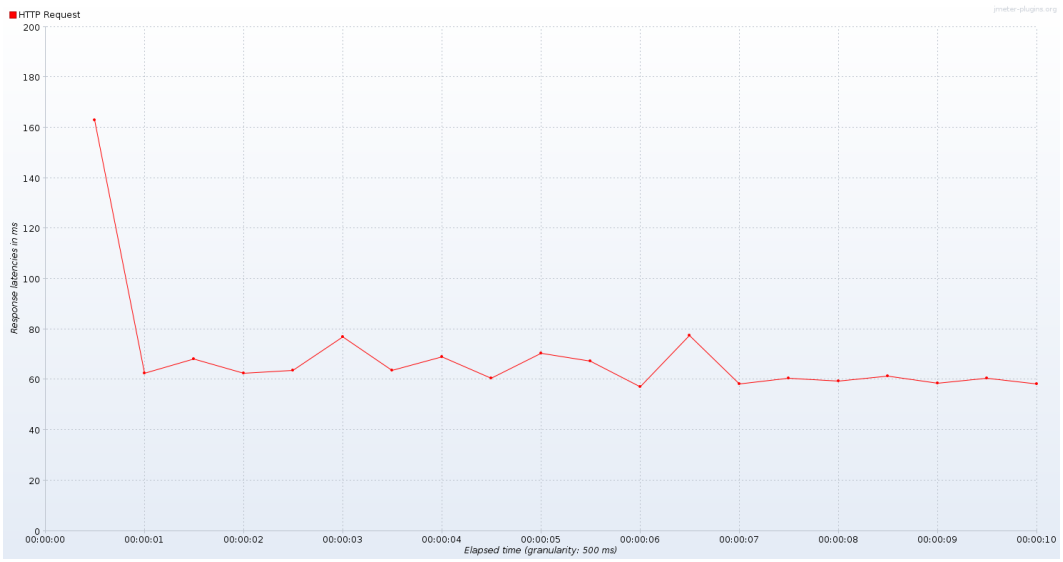

Figure .1: Monolith Latency

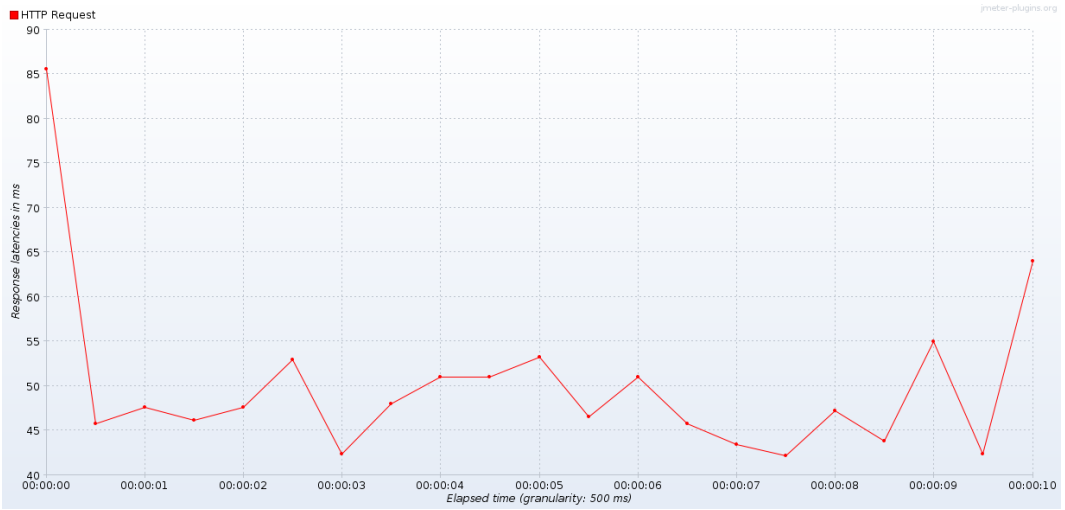

Figure .2: Microservice Latency

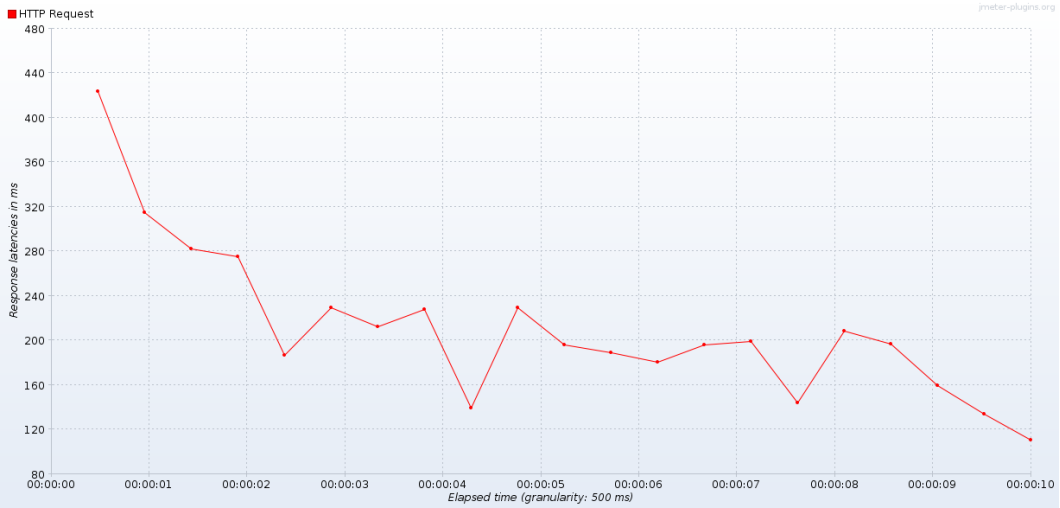

Figure .3: Serverless Latency

Bibliography

Adzic, Gojko and Robert Chatley (2017). "Serverless Computing: Economic and Architectural Impact." In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: ACM, pp. 884–889. ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3117767. URL: http://doi.acm.org/10.1145/3106237.3117767 (cit. on p. 23).

Alexander, Christopher (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press. ISBN: 0195019199. URL: http://www.amazon.fr/exec/obidos/ASIN/0195019199/citeulike04-21 (cit. on p. 17).

Alexander, Christopher (1979). *The Timeless Way of Building*. Oxford University Press. ISBN: 0195024028 (cit. on p. 12).

Baldini, Ioana et al. (2017). "The Serverless Trilemma: Function Composition for Serverless Computing." In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2017. Vancouver, BC, Canada: ACM, pp. 89–103. ISBN: 978-1-4503-5530-8. DOI: 10.1145/3133850.3133855. URL: http://doi.acm.org/10.1145/3133850.3133855 (cit. on p. 37).

Beck, Kent (1998). "Using a pattern language for programming." In: *In Addendum to the Proceedings of ooPSLA '87*. Vol. 23. ooPSLA '87, p. 16 (cit. on p. 12).

Beck, Kent et al. (1996). "Industrial Experience with Design Patterns." In: *Proceedings of the 18th International Conference on Software Engineering*. ICSE '96. Berlin, Germany: IEEE Computer Society, pp. 103–114. ISBN: 0-8186-7246-3. URL: http://dl.acm.org/citation.cfm?id=227726.227747 (cit. on pp. 12, 19).

Berners-Lee, Tim (1989). *Information management: A proposal*. URL: http://www.w3.org/History/1989/proposal.html (cit. on p. 40).

Bevan, Nigel (1999). "Usability Issues in Web Site Design." In: (cit. on p. 56).

Bisbal, Jesús et al. (1999). "Legacy Information Systems: Issues and Directions." In: *IEEE Softw.* 16.5, pp. 103–111. ISSN: 0740-7459. DOI: 10.1109/52.795108. URL: https://doi.org/10.1109/52.795108 (cit. on p. 61).

Boss, Birgit et al. (2016). "Setting Up Architectural SW Health Builds in a New Product Line Generation." In: *Proccedings of the 10th European Conference on Software Architecture Workshops*. ECSAW '16. Copenhagen, Denmark: ACM, 16:1–16:7. ISBN: 978-1-4503-4781-5. DOI: 10.1145/2993412.3003392. URL: http://doi.acm.org/10.1145/2993412.3003392 (cit. on p. 60).

Bray, T. (2017). *The JavaScript Object Notation (JSON) Data Interchange Format*. STD 90. RFC Editor. URL: http://www.rfc-editor.org/rfc/rfc8259.txt (cit. on p. 41).

Buschmann, Frank et al. (1996). *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing. ISBN: 0471958697, 9780471958697 (cit. on pp. 11–16).

Buschmann, Frank et al. (2000). *Pattern-Oriented Software Architecture, A System of Patterns: Volume 1 (Wiley Software Patterns Series)*. Wiley (cit. on pp. 15–17).

Carrasco, Andrés, Brent van Bladel, and Serge Demeyer (2018). "Migrating Towards Microservices: Migration and Architecture Smells." In: *Proceedings of the 2Nd International Workshop on Refactoring*. IWoR 2018. Montpellier, France: ACM, pp. 1–6. ISBN: 978-1-4503-5974-0. DOI: 10.1145/3242163.3242164. URL: http://doi.acm.org/10.1145/3242163.3242164 (cit. on p. 23).

Chhabra, Shruti and V. S. Dixit (2015). "Cloud Computing: State of the Art and Security Issues." In: *SIGSOFT Softw. Eng. Notes* 40.2, pp. 1–11. ISSN: 0163-5948. DOI: 10.1145/2735399.2735405. URL: http://doi.acm.org/10.1145/2735399.2735405 (cit. on p. 19).

CNCF (2018). *Serverless Whitepaper v1.0*. Cloud Native Computing Foundation (CNCF). URL: https://github.com/cncf/wg-serverless/tree/master/whitepapers/serverless-overview (cit. on p. 37).

Conway, Melvin E. (1968). "How do committees invent." In: *Datamation* 14.4, pp. 28–31. URL: http://www.melconway.com/Home/pdf/committees.pdf (cit. on p. 55).

Coplien, James O. (1992). *Advanced C++ Programming Styles and Idioms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0-201-54855-0 (cit. on p. 12).

Coplien, James O. and Neil B. Harrison (2004). *Organizational Patterns of Agile Software Development*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc. ISBN: 0131467409 (cit. on p. 55).

Evans, Josh (2016). "Mastering Chaos - A Netflix Guide to Microservices." In: *InfoQ*. URL: https://www.infoq.com/presentations/netflix-chaos-microservices (cit. on p. 2).

Eyk, Erwin van, Alexandru Iosup, Cristina L. Abad, et al. (2018). "A SPEC RG Cloud Group's Vision on the Performance Challenges of FaaS Cloud Architectures." In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE '18. Berlin, Germany: ACM, pp. 21–24. ISBN: 978-1-4503-5629-9. DOI: 10.1145/3185768.3186308. URL: http://doi.acm.org/10.1145/3185768.3186308 (cit. on p. 37).

Eyk, Erwin van, Alexandru Iosup, Simon Seif, et al. (2017). "The SPEC Cloud Group's Research Vision on FaaS and Serverless Architectures." In: *Proceedings of the 2Nd International Workshop on Serverless Computing*. WoSC '17. Las Vegas, Nevada: ACM, pp. 1–4. ISBN: 978-1-4503-5434-9. DOI: 10.1145/3154847.3154848. URL: http://doi.acm.org/10.1145/3154847.3154848 (cit. on pp. 35–37).

Fielding, R. and J. Reschke (2014a). *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230. RFC Editor. URL: http://www.rfc-editor.org/rfc/rfc7230.txt (cit. on p. 40).

Fielding, R. and J. Reschke (2014b). *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. RFC Editor. URL: http://www.rfc-editor.org/rfc/rfc7231.txt (cit. on p. 40).

Fielding, Roy Thomas (2000). "Architectural Styles and the Design of Network-based Software Architectures." AAI9980887. PhD thesis. ISBN: 0-599-87118-0 (cit. on p. 40).

Fowler, Martin (2002). *Patterns of Enterprise Application Architecture*. first edition. Addison-Wesley Professional. ISBN: 0321127420 (cit. on p. 9).

Fowler, Martin (2014). *Microservices - A Definition of this new Architectural Term*. URL: https://martinfowler.com/articles/microservices.html (cit. on pp. 24, 26, 28, 30, 32).

Fowler, Martin (2015). *Microservice Premium*. URL: https://martinfowler.com/bliki/MicroservicePremium.html (cit. on pp. 33, 63).

Fromm, Ken (2012). *Why The Future Of Software And Apps Is Serverless*. URL: https://readwrite.com/2012/10/15/why-the-future-of-software-and-apps-is-serverless/ (cit. on p. 36).

Gurp, Jilles van and Jan Bosch (2002). "Design erosion: problems and causes." In: *Journal of Systems and Software* 61.2, pp. 105–119. ISSN: 0164-1212. DOI: https://doi.org/10.1016/S0164-1212(01)00152-2. URL: http://www.sciencedirect.com/science/article/pii/S0164121201001522 (cit. on pp. 60, 63).

Hohpe, Gregor and Bobby Woolf (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional. ISBN: 0321200683 (cit. on p. 42).

Iyengar, M. K. et al. (2009). "Modularization of a Large-Scale Business Application: A Case Study." In: *IEEE Software* 26, pp. 28–35. ISSN: 0740-7459. DOI: 10.1109/MS.2009.42. URL: doi.ieeecomputersociety.org/10.1109/MS.2009.42 (cit. on pp. 61, 62).

Kamal, Ahmad Waqas and Paris Avgeriou (2010). "Modeling the Variability of Architectural Patterns." In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC '10. Sierre, Switzerland: ACM, pp. 2344–2351. ISBN: 978-1-60558-639-7. DOI: 10.1145/1774088.1774572. URL: http://doi.acm.org/10.1145/1774088.1774572 (cit. on p. 11).

Kaplan James M. Forrest William, Kindler Noah (2008). *Revolutionizing Data Center Efficiency*. McKinsey and Company (cit. on pp. 20, 56).

Kernighan, Brian W. and P. J. Plauger (1976). *Software tools*. Addison-Wesley (cit. on p. 47).

Koomey Jonathan, Taylor Jon (2015). *New data supports finding that 30 percent of servers are 'Comatose', indicating that nearly a third of capital in enterprise data centers is wasted*. Oakland, CA: Anthesis Group (cit. on pp. 20, 56).

Len Bass Paul Clements, Rick Kazman (2012). *Software Architecture in Practice*. third edition. Addison-Wesley Professional; ISBN: 0321815734 (cit. on p. 21).

Lerner, Reuven M. (2014). "At the Forge: 12-factor Apps." In: *Linux J.* 2014.245. ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=2682554.2682559 (cit. on pp. 50, 62).

López, Manuel Ramírez and Josef Spillner (2017). "Towards Quantifiable Boundaries for Elastic Horizontal Scaling of Microservices." In: *Companion Proceedings of the10th International Conference on Utility and Cloud Computing*. UCC '17 Companion. Austin, Texas, USA: ACM, pp. 35–40. ISBN: 978-1-4503-5195-9. DOI: 10.1145/3147234.3148111. URL: http://doi.acm.org/10.1145/3147234.3148111 (cit. on p. 22).

Mair, Matthias, Sebastian Herold, and Andreas Rausch (2014). "Towards Flexible Automated Software Architecture Erosion Diagnosis and Treatment." In: *Proceedings of the WICSA 2014 Companion Volume*. WICSA '14 Companion. Sydney, Australia: ACM, 9:1–9:6. ISBN: 978-1-4503-2523-3. DOI: 10.1145/2578128.2578231. URL: http://doi.acm.org/10.1145/2578128.2578231 (cit. on p. 60).

Marathe, Aniruddha et al. (2014). "Exploiting Redundancy for Cost-effective, Time-constrained Execution of HPC Applications on Amazon EC2." In: *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*. HPDC '14. Vancouver, BC, Canada: ACM, pp. 279–290. ISBN: 978-1-4503-2749-7. DOI: 10.1145/2600212.2600226. URL: http://doi.acm.org/10.1145/2600212.2600226 (cit. on p. 19).

Martin, Robert C. and Micah Martin (2006). *Agile Principles, Patterns, and Practices in C# (Robert C. Martin)*. Upper Saddle River, NJ, USA: Prentice Hall PTR. ISBN: 0131857258 (cit. on p. 28).

McIlroy, M. D. (1978). "UNIX time-sharing system: Foreword." In: *The Bell System Technical Journal* 57.6, pp. 1899–1904. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1978.tb02135.x (cit. on pp. 28, 44).

Muthig, D. and M. Lindvall (2008). "Bridging the Software Architecture Gap." In: *Computer* 41, pp. 98–101. ISSN: 0018-9162. DOI: 10.1109/MC.2008.176. URL: doi.ieeecomputersociety.org/10.1109/MC.2008.176 (cit. on p. 61).

Newman, Sam (2015). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media. ISBN: 1491950358 (cit. on pp. 28, 30, 40).

Raible, Matt (2018). *The JHipster Mini-Book*. lulu.com. ISBN: 132963814X (cit. on p. 54).

Ramage, M. and K. Bennett (1998). "Maintaining Maintainability." In: *Proceedings of the International Conference on Software Maintenance*. ICSM '98. Washington, DC, USA: IEEE Computer Society, pp. 275–. ISBN: 0-8186-8779-7. URL: http://dl.acm.org/citation.cfm?id=850947.853301 (cit. on p. 61).

Ras, Eric, Jörg Rech, and Sebastian Weber (2009). *Knowledge Services for Experience Factories* (cit. on p. 15).

Raymond, Eric S. (1996). *The New Hacker's Dictionary (3rd Ed.)* Cambridge, MA, USA: MIT Press. ISBN: 0-262-68092-0 (cit. on p. 55).

Raymond, Eric S. (2003). *The Art of Unix Programming*. first edition. Addison-Wesley Professional. ISBN: 0131429019 (cit. on pp. 23, 24, 44–48).

Rech, Jörg and Eric Ras (2011). "Aggregation of Experiences in Experience Factories into Software Patterns." In: *SIGSOFT Softw. Eng. Notes* 36.2, pp. 1–4. ISSN: 0163-5948. DOI: 10.1145/1943371.1943390. URL: http://doi.acm.org/10.1145/1943371.1943390 (cit. on pp. 14, 16).

Roberts, Mike (2016). *Serverless Architectures*. URL: https://martinfowler.com/articles/serverless.html (cit. on pp. 35, 36).

Salus, Peter H. (1994). *A Quarter Century of UNIX*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co. ISBN: 0-201-54777-5 (cit. on pp. 45, 46).

Scerbakov, Nikolai (2018). *Internet-Based Information Systems - Lecture Notes*. URL: https://coronet.iicm.tugraz.at/is/scripts/lesson08.pdf (cit. on pp. 10, 40).

Spillner, Josef (2017). "Practical Tooling for Serverless Computing." In: *Proceedings of the10th International Conference on Utility and Cloud Computing*. UCC '17. Austin, Texas, USA: ACM, pp. 185–186. ISBN: 978-1-4503-5149-2. DOI: 10.1145/3147213.3149452. URL: http://doi.acm.org/10.1145/3147213.3149452 (cit. on p. 19).

Tešanović, Aleksandra (2001). *What is a pattern?* (Cit. on pp. 11, 12, 18, 19).

Tichy, W. F. (1997). "A Catalogue of General-Purpose Software Design Patterns." In: *Proceedings of the Tools-23: Technology of Object-Oriented Languages and Systems*. TOOLS '97. Washington, DC, USA: IEEE Computer Society, pp. 330–. ISBN: 0-8186-8383-X. URL: http://dl.acm.org/citation.cfm?id=832250.832625 (cit. on pp. 13, 14).

Torkura, Kennedy A., Muhammad I.H. Sukmana, and Christoph Meinel (2017). "Integrating Continuous Security Assessments in Microservices and Cloud Native Applications." In: *Proceedings of the10th International Conference on Utility and Cloud Computing*. UCC '17. Austin, Texas, USA: ACM, pp. 171–180. ISBN: 978-1-4503-5149-2. DOI: 10.1145/3147213.3147229. URL: http://doi.acm.org/10.1145/3147213.3147229 (cit. on p. 30).

Villamizar, Mario et al. (2017). "Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures." In: *Service Oriented Computing and Applications* 11.2, pp. 233–247. ISSN: 1863-2394. DOI: 10.1007/s11761-017-0208-y. URL: https://doi.org/10.1007/s11761-017-0208-y (cit. on pp. 23, 25).

Wiggins, Adam (2011). *The Twelve-Factor App*. URL: https://12factor.net/ (cit. on p. 58).

Wiggins, Adam (2018). *The Twelve-Factor App*. URL: https://12factor.net/ (cit. on p. 44).

Wizenty, Philip et al. (2017). "MAGMA: Build Management-based Generation of Microservice Infrastructures." In: *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*. ECSA '17. Canterbury, United Kingdom: ACM, pp. 61–65. ISBN: 978-1-4503-5217-8. DOI: 10.1145/3129790.3129821. URL: http://doi.acm.org/10.1145/3129790.3129821 (cit. on p. 30).

Wright, Hyrum K. and Dewayne E. Perry (2012). "Release Engineering Practices and Pitfalls." In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, pp. 1281–1284. ISBN: 978-1-4673-1067-3. URL: http://dl.acm.org/citation.cfm?id=2337223.2337395 (cit. on p. 43).