FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Evaluation of Microservice Implementation Approaches for Image Processing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Masterstudium Software Engineering/Internet Computing

eingereicht von

Manuel Kruisz, BSc
Matrikelnummer 01026780

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Mag. Dr. Horst Eidenberger

Wien, 19. Jänner 2020

Manuel Kruisz	Horst Eidenberger

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Evaluation of Microservice Implementation Approaches for Image Processing

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Master's programme Software Engineering & Internet Computing

by

Manuel Kruisz, BSc
Registration Number 01026780

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Mag. Dr. Horst Eidenberger

Vienna, 19th January, 2020

Manuel Kruisz	Horst Eidenberger

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

Erklärung zur Verfassung der Arbeit

Manuel Kruisz, BSc

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

Wien, 19. Jänner 2020

Manuel Kruisz

Acknowledgements

My special thanks goes to Professor Eidenberger, the supervisor of this thesis. His outstanding patience, feedback and guidance played an important role in the creation of this work.

I would further like to thank everyone who enabled me to become a software engineer, which includes everyone involved in the Technical University of Vienna, current and past employers, colleagues and everyone openly sharing their knowledge in any way.

Kurzfassung

In den letzten Jahren ist ein wachsendes Interesse an dem relativ jungen Thema der Microservice-Architekturen zu beobachten. Einhergehend damit steigt die Anzahl der Erfahrungen zu Microservices und besonders Vorteile dieser Architektur, als auch zunehmend potentielle Nachteile sind ein häufig diskutiertes Thema. Während bereits erste Erfolgsrezepte und Architekturmuster zu der Konstruktion solcher Systeme entstehen, wird noch unzureichend thematisiert, wie ein Microservice System am besten gebaut werden kann und in welchen Fällen dieser Architekturstil wirklich anwendbar ist. Viele Erkenntnisstände und Informationen zu diesem Thema sind noch allgemeiner Natur und spezielle Anwendungsfälle mit besonderen Eigenschaften völlig unerforscht.

In dieser Arbeit wurden Microservices im Kontext von Bilderverarbeitungssystemen näher betrachtet. Als Basis der Evaluierung diente ein System, das fünf verschiedene Arten von verbreiteten Bildtransformationen anbietet. Anhand der bestehenden Anforderungen wurde das System ohne Frameworks, mit einem Framework und mit einem *Function-as-a-Service* Ansatz in der Microservice-Architektur implementiert. Diese drei resultierenden Systeme wurden quantitativ durch den Entwicklungsaufwand und ihre Leistungsfähigkeit bewertet. Qualtiativ wurden Vor- und Nachteile der einzelnen Entwicklungsansätze evaluiert und allgemeine Erkenntnisse zu Microservice-Architekturen im Kontext der Bildverarbeitung festgehalten.

In den Resultaten wurde festgestellt, dass für solch ein Bildverarbeitungssystem kein klar zu bevorzugender Ansatz existiert. Jede der drei Implementierungen bietete verschiedenartige Vor- und Nachteile in Form eines Kompromisses zwischen Flexibilität und Entwicklungsgeschwindigkeit. Es zeigte sich jedoch, dass im allgemeinen ein *Function-as-a-Service* Ansatz hier sehr gut geeignet sein kann und sowohl die Entwicklungszeit als auch die Komplexität des resultierenden Quellcodes positiv beeinflusst.

Die Kombination von Bildverarbeitung und Microsrevices brachte Eigenschaften mit sich, die starke Auswirkung auf die resultierende Architektur hatten. Primär wurde diese durch die lange Verarbeitungszeit von einzelnen Anfragen und die Netzwerklast durch den Transfer von Bilddateien zwischen Services beeinflusst. Als Resultat dessen sind verbreitete, simple Ansätze nur mit gravierenden Nachteilen umsetzbar, und es werden komplexe Mechanismen wie asynchrone Kommunikation im gesamten System notwendig. Jedoch zeigte sich, dass die korrekte Anwendung von Microservices zu einfacher Wartbarkeit und Skalierbarkeit eines solchen Systems führen kann.

Abstract

In recent years we can observe an increasing interest in the relatively young topic of Microservice Architectures. As a consequence thereof, the number of experiences with Microservices is increasing and both advantages, as well as disadvantages become a commonly discussed topic. While first best practices and architectural patterns regarding the construction of such systems emerged, there is a lack of discussion regarding the construction and applicability in certain contexts of Microservice Systems. The current state of knowledge revolves primarily around general properties of microservices and therefore use cases in contexts with special characteristics are still uncharted terrain.

This thesis provides a closer look at Microservices in the context of image processing systems. The evaluation was performed based on a system offering five commonly used image transformations. With a given set of requirements this system was implemented without the use of a framework, with a framework and with an *Function as a Service* approach to achieve a Microservice Architecture. All three of the resulting applications were evaluated via quantitative metrics such as the time spent on development and performance characteristics. A qualitative evaluation was performed as well, which compared advantages and disadvantages of the different approaches and yielded findings regarding Microservice Architectures in the context of image processing.

The results showed that no single, generally preferable approach exists for such a system. All three implementations offered different advantages and disadvantages in the form of a trade-off between flexibility and development speed. However, there were indications that generally a *Function as a Service* approach can yield the most benefits, because of a strong reduction in both development time and the complexity of the resulting source code.

The combination of image processing and Microservices lead to properties which had a strong influence on the resulting architecture. This was primarily caused by the long processing time of the requests and the high load on the network caused by the transfers of image files between services. As a result thereof, commonly used and simple approaches would often have resulted in serious disadvantages, and more complex mechanisms such as asynchronous communication were necessary throughout the whole system. However, it became apparent that a correct application of microservice principles leads to better maintainability and scalability of such a system.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The microservice architectural style is now used in most of the well-known companies in the industry. Names like *Uber*, *Netflix*, *Amazon* and *Ebay*, among numerous others, come to mind when there is talk about microservices. And the interest in this architectural style does not appear to die off either. Since the term started to become popular in 2014, we are able to observe a growth in interest in year. A search on the IEEE Digital Library [IEE] for the terms *Microservice* and *Microservices* reveals a significant increase in the number of articles published on the topic per year, as shown in Table 1.1.

Table 1.1: IEEE search results for Microservices.

Results	Year
3	2014
24	2015
76	2016
152	2017

A similar trend can also be observed in Figure 1.1, which shows a Google Trends [MST] search for the term *Microservices* over the last five years. Here also, the interest has been constantly growing.

As a result of the growing interest in this young architectural style and its fast adoption, we can observe that research, tools and approaches related to microservices, are all still constantly evolving. Our approaches to the construction of microservice systems are shifting from building microservices from scratch over using dedicated frameworks to approaches like Function as a Service (FaaS), which provide an abstraction to build

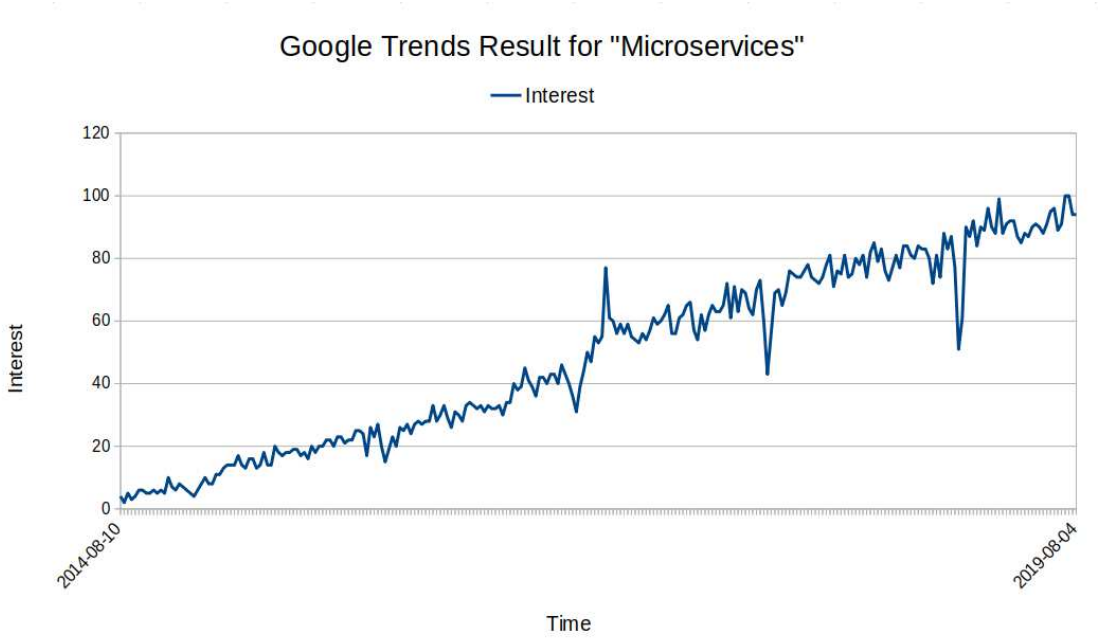

Figure 1.1: Microservices Google Trends.

microservices in the form of simple functions. With every evolutionary step in these approaches we are moving towards a higher level of abstraction and transparency for commonly used parts of systems.

But are microservices applicable to every domain? Is this architecture suited for domains like image processing? If so, do current best practices, approaches and tools yield benefits for this domain or are practitioners better off by building infrastructure code on their own? In this thesis we are trying to answer these and other questions.

1.2 Aim of the Work

Because of this field's relatively young age, research is still ongoing and many areas, applications and effects of this architectural style are yet to be researched. One such area is *image processing*. In this work we want to answer some of the questions related specifically to properties of this domain.

For this purpose we create and evaluate different approaches for the construction of an image processing system via microservices. This in turn enables us to answer the following research questions:

- How well is the implementation of image processing services supported by this architectural style?

- Are the identified best practices applicable in the given domain?
- Are there limits to these approaches in the given context?
- Is one approach preferable to the others?
- What are the advantages and drawbacks of each approach in the given domain?
- What are these advantage's and disadvantage's influences on the implementation of the system as a whole?

An additional goal of this work is to uncover further ideas for research based questions. Due to the relatively young age of the field and the limited amount of similar work, we expect to find numerous additional research questions. If applicable to the overall focus of this work and the nature of the additional questions we try to investigate and answer them. In any case we strive to give as much insight into the problem as possible to enable future work to answer them.

1.3 Methodological Approach

The chosen approach consists of several phases, which build on each other. In the first phase we conduct a literature review, which provides us with insights into the known best practices and challenges in the context of building microservice-, image processing- and distributed systems.

Based on these results, we closely inspect the given requirements for the intended application and design an architecture in the second phase. The resulting architecture should fulfill all given requirements and the identified best practices for a microservice system.

On the basis of this architecture we build the image processing system via three different approaches in the next phase. Each of the different approaches is realized as a prototypical implementation. From an outer perspective the behaviour of each system should appear identical, but the inner workings and the construction process can drastically differ. During the process of building the systems we measure aspects like *time spent for the construction* and take note of interesting aspects and challenges encountered during the implementation phase for each approach.

The first prototype is implemented without the use of any frameworks. We expect this to result in an application structure without any constraints on the inner and outer workings of the system, but to provide the least support from existing tools.

The second prototype is implemented using a general purpose microservice framework. This approach should offer some support in the implementation while still maintaining a certain level of flexibility.

The third prototype is implemented using a FaaS Framework. In this approach a significant part of the required infrastructure of the system is already provided and it is only necessary to add the logic for the specific image processing tasks.

Once the implementation of the different prototypes is complete, we begin the evaluation phase. By combination of the research questions, the knowledge gained from the literature review, insights gained during the implementation phase and the actual resulting systems, we evaluate each approach on its own and in relation to the other prototypes.

1.4 Structure of the Work

The structure of the work closely relates to the different phases used in the methodological approach.

Relevant Background contains information about the underlying architectural principles in software construction and especially those related to distributed system architectures. In this section take a closer look at the overall concepts of the microservice architectural style. We discuss how this style relates to existing principles and the recent concepts that enabled microservices to emerge. Thereafter, we dive deeper into the currently known positive and negative effects of using this architectural style. These advantages and disadvantages lead us to different patterns which yield the mentioned benefits and try to mitigate the known and common downsides of such systems.

Since microservice systems possess some very specific requirements in their infrastructure we have a look at the most important developments, techniques and technologies that enable this architectural style.

The background part is concluded by a close inspection and discussion of related work, which consists primarily of, but is not limited to, research in the domain of microservices and especially the application of this architectural style in the context of the construction of various systems. Other areas of interest considered in this section are architecture related research, but also research related to software construction in general and the different effects of using certain techniques.

The **Design** section describes the requirements for the given image processing systems. This part contains functional, as well as non-functional requirements and gives us an understanding on how the system should behave from a client's perspective and which use cases are to be supported. After the requirements are clear we present and discuss an architecture, which supports them and the best practices found in the *Background* chapter. The last part of the *Design* chapter gives an overview of the technology choices taken to build the system. Here we briefly mention the advantages as disadvantages of each technology in regards to the system we are building.

The **Implementation** section presents the concrete implementation of the three different prototypes. For each prototype there are explanations how the system is built and fulfills the various requirements.

The **Evaluation** section compares the prototypes and their different aspects via quantitative and qualitative metrics. By combining the results the comparison and the theoretical base we have built in the *background* chapter, we answer the research questions of this work. We conclude this chapter with a critical reflection of the chosen approach and the decisions made during the implementation and evaluation phase, by having a look at their possible impact on the results and discussing the effects of possible alternative approaches.

In the end of this work we draw conclusions from our findings and discuss future work.

CHAPTER 2

Relevant Background

2.1 Software Architectures

2.1.1 Monolithic Architecture

The term *Monolith* describes a software system that is packaged and deployed as a single executable file. This is often viewed as the traditional approach to design a software system [RIC].

The resulting application usually consists of several components and layers providing the different capabilities of the system. A monolith commonly contains all code for the graphical interface, business logic, integrations to third-party systems and database access. All functionality required by the business is present in the single codebase [RIC]. Supporting all requirements of the business leads to vastly different parts of functionality with little to no overlap. Without a strategy to manage and structure these different capabilities and features the code might quickly become unmanageable [APP]. In order to avoid these problems two main strategies in the structuring of such an application are employed. These are *layering* and *components* as shown in figure 2.1.

In layering the application is sliced horizontally by the different technical layers. A typical layering would be to separate the *user interface* (UI), business logic and database access layers [APP]. These different parts are then ordered in a clear hierarchy that only allows one layer to communicate with the layer directly underneath [BCK12]. This and similar approaches being so common lead to the emergence of well-known patterns like the *Model-View-Controller* Pattern [SW14].

The second strategy for structuring is to separate the system into *components*. Components are independent parts of the program that can be composed and linked. One component provides and is responsible for a given part of the overall functionality [SHO]. This could for example be a *user component* that handles user management. Another

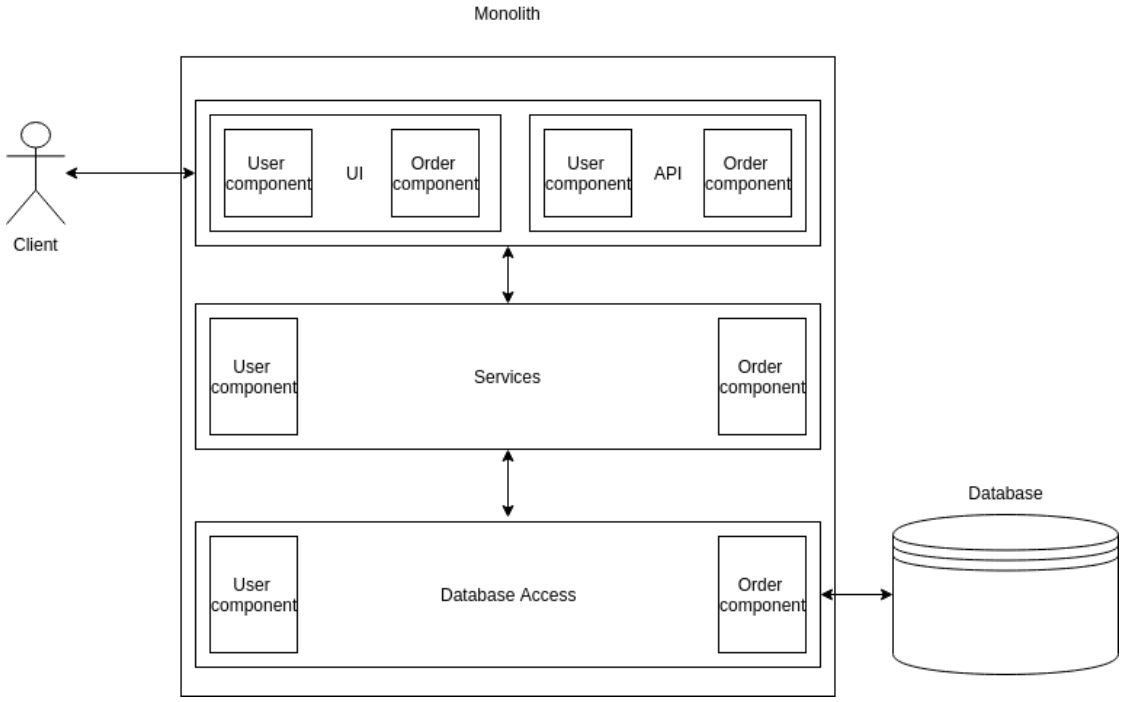

Figure 2.1: A monolithic system separated into different layers for technical capabilities and different components within the layers for different functionality [APP] (*modified*).

component, for example an *order component*, could then call the *user component* to receive certain data about a given customer. However, when components reside next to each other, there is usually very little resistance to change the way they communicate, which can both act as a blessing and a curse [RIC].

Therefore when employing both of these layering strategies, changes within one layer are usually easy to perform, however bear a risk of breaking down the component structure within the system [RIC]. On the other hand changes that span across different layers take more time because of the additional effort to create the communication structures and interfaces between the different layers and the need to recompile the whole application if there were changes in every layer [RIC].

These days the term *Monolith* received an increasingly negative connotation over time. On first sight the consensus might appear to be that *Monoliths are bad*, which like most other generalizations, does not hold true in every case. There are still good reasons to favor a monolithic system over another architectural approach [FOWd]. In some cases it can also be an advantage that the previously mentioned boundaries created by layering the application and those created by splitting the system into components are much weaker than the boundaries in a distributed system. This makes it easier to perform changes across large parts of the overall system in monolithic systems and to achieve higher flexibility when it comes to larger changes in the general structure of

functionality [FOWc] [Ric18]. From the operational point of view, the application being a single deployable or executable file results in a relatively simple deployment process [APP]. A single executable file could even be deployed manually and does not require a complex deployment infrastructure [Ric18]. Should any upscaling be required, monolithic architectures offer the possibility to have several instances running behind a load balancer [APP] [Ric18].

However, once the application and the number of developers working on it reach a certain size these perks can turn into a burden. Over time the lack of strict boundaries causes the separation of components to break down [APP]. Developers working on the same codebase start to face conflicting changes with their peers, oftentimes producing side effects that are difficult to trace and understand. The application grows to a size that tests the limits of humans and machines alike [APP]. Developers, especially those new to the system, are no longer able to understand or even be familiar with all parts of the application [APP] [Ric18]. Integrated Development Environments (IDEs) on normal hardware stop being able to handle the amount of source code, runs of the automated test suites take days to complete and the deployment of the application takes hours [Ric18].

Therefore it is often recommended to start out with a monolithic system. Only when the application grows to a size where the drawbacks of monoliths come into effect one should consider splitting the system up [FOWc]. When this moment arrives the developers should have enough knowledge about different possible boundaries in the system. Therefore separating the system into parts of related functionality should be possible at this point in time [FOWc]. This trend can be observed amongst several companies in the industry. Many started out by using a monolithic approach and only later on split the monolith into microservices. When companies tried to immediately start out with a microservice system in many cases their design failed [FOWc].

2.1.2 Service Oriented Architecture

Service Oriented Architecture (SOA) is a structured approach to software architecture that splits a monolithic system into several loosely coupled services. Each such service contains only related functionality and is invokable via a standard-based interface over the network. By creating separate services the different parts of the system are decoupled and clear boundaries between the different components are created [PVDH07].

Figure 2.2 shows the outline of a SOA based system in which separate services are used for the handling of users and orders. In contrast to the monolithic architecture in figure 2.1, the *User Service* and *Order Service* reside in separate applications.

SOA is based on several other concepts like *Component Based Architecture* and *Interface Based Design* and *Distributed Systems* [VAMD09]. The services of SOA can typically be classified into three different types [Mah07]

- **Infrastructure services** which provide capabilities like management, monitoring, identification and security.

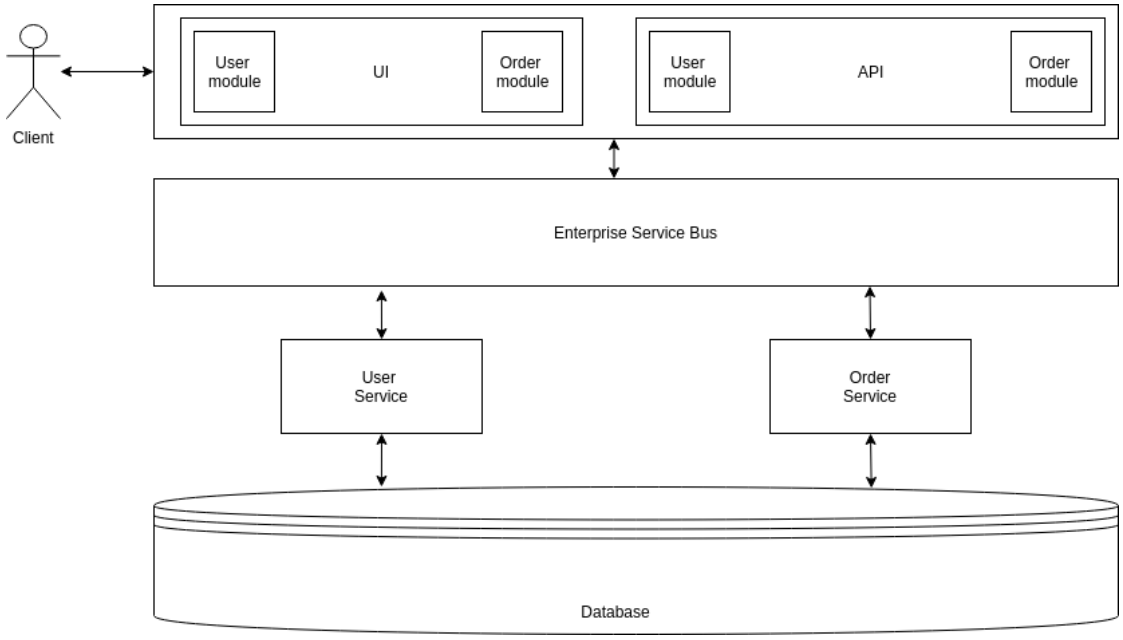

Figure 2.2: SOA system with different services that communicate and are accessible via the Enterprise Service Bus.

- **Business-neutral services** which handle the workflow, messaging broking, notifications and scheduling.
- **Business services** which are based on business domains and encapsulate related functionality of the system.

Depending on the type of interaction a service is involved in, it can either act as the *Service Provider*, which offers invokable functionality to the overall system or as a *Service Consumer* which invokes, and makes us of, the functionality provided by other services. These roles are not mutually exclusive for a single service and even for a single piece of functionality a service can act both as a provider and consumer [VAMD09].

Separating a system into smaller components via a SOA based approach promises the following benefits

- **Increased flexibility and agility**. Teams are able to work on independent applications, which enables a higher level of parallelization during development. This in turn leads to faster development cycles and makes it easier to change parts of a system. Changes in such services are smaller in scope compared to changes in a monolithic system and are less likely to produce side effects [Mah07] [SOA].
- **Well-defined building blocks around business capabilities.** Because of the clear boundaries between the services that make up the whole architecture it

is easier to determine which building block is responsible for a given part of the system. Building blocks can then be modeled after the different business capabilities, enabling a vertical slicing of the organization [Mah07] [VAMD09] [SOA].

- **Reuse**. The different building blocks can be reused by other applications in the system, whereas in monolithic systems such functionality must be implemented in each application [Mah07] [VAMD09] [SOA].
- **Ease of maintenance**. Smaller applications are easier to understand and therefore easier to modify. If the deployment process is only viewed from one service's perspective, the limited size and scope leads to easier deployment [Mah07].
- **Better scalability**. In bigger applications there are usually only parts of the system that need to be scaled up. Having separate services that are only responsible for certain parts of the functionality makes it possible to perform scaling only on the services which are under a high amount of system load [Mah07] [VAMD09].

On the other hand SOA also comes with its limitations and drawbacks

- **High overhead & vendor lock-ins** if the protocol that is used for communication between the services is proprietary [XWQ16] [Mah07]. This is especially the case if a heavy-weight protocol or middleware like an *Enterprise Service Bus* (ESB) is used.
- **Big upfront investment**. In contrast to a monolithic system, SOA takes more effort to set up. Regardless of the number of services, the full infrastructure to deploy an arbitrary amount of services and to enable the communication between these services needs to be in place. Additionally, there is much more planning involved, because services need to be grouped, whereas in a monolithic system getting such bounds wrong does not result in a costly refactoring [Mah07].

The concept of SOA is highly related to different architectural styles and can be used in conjunction with *Space-based Architecture*.

2.1.3 Space-based Architecture

Space-based Architecture (SBA) is an architectural style that uses the *tuple space paradigm* to achieve linear scalability [SBA]. Also known as the *blackboard metaphor*, the *tuple space paradigm* provides a repository of tuples in which all data of the application resides and can be accessed concurrently. This repository is then replicated amongst the processing nodes. In more general terms such a repository can be viewed as a distributed shared memory.

With SBA a monolithic system can be decomposed into several smaller applications by creating agents which each provide parts of the overall functionality [EG02]. The

actual location of the data the application operates on and possible issues in terms of concurrency are then transparently handled by the infrastructure [SBA].

The shared repository in this case acts as an isolation layer between the different services, because service do not need to talk directly to each other and therefore strongly increases the independence of services [SBA]. In general this approach offers three types of decoupling in communication between services. Communication is decoupled in time, because tuples possess their own lifetime which is independent of both the sender and receiver, decoupled in destination, because senders do not know who will receive a given tuple and decoupled in space, because associative addressing can be used in the tuple space [EG02].

This leads to a higher robustness in comparison to a peer-to-peer system. If parts of the system experience a failure, replication in the system still allows the remainder of the application to function normally [EG02]. If scaling is required this architecture offers the possibility to add additional agents and spaces to the overall system [EG02].

The decoupling aspect of this architecture can also be found in the more general architectural style of *Event-Driven Architecture*.

2.1.4 Event-driven Architecture

Event-driven Architecture (EDA) is another architecture that aims to achieve loose coupling in a larger system. The core concept of this architectural style is the creation and consumption of events [LS09] [Jur10]. In EDA two different parties interact with events:

- **Event Emitters** listen for changes in the system and dispatch events to the *event channel* [Mic06].
- **Event Consumers** receive events from the *event channel* and handle them accordingly [Mic06].

By using an *event channel* between emitters and consumers we achieve a decoupling of the sending and the receiving party. Emitters simply publish events to the event channel, while having neither knowledge about the types nor the amount of consumers. The situation on the consumer side is very similar. A consumer does not have information about the emitters of the events he consumes and possesses no information about which other consumers are operating on the same events [Mic06].

2.1.5 Shared-Nothing Architecture

In strong contrast to an architectural approach like SBA, *Shared-Nothing Architecture* (SNA) is an architectural style that emphasizes the importance of creating an architecture in which components do not share any resources and only communicate by passing messages [NZT96]. The goal of this isolation is to eliminate bottlenecks and to create

a system that is easily scalable [NZT96]. Applicable to a variety of domains, SNA is present on the operating-system level, when components must not access the same pieces of hardware [Sto86], in databases when sharding is used to split up a larger data set into separate databases [SHA] responsible only for a subset of the overall data and in software architectures when applications have exclusive control over certain parts of the data.

2.2 Microservices

2.2.1 Overview of Microservice Architecture

Microservices are small, autonomous services related to a specific business capability, that work together [New15] [FOWd].

The term *Microservices* was first discussed during a software architecture workshop to describe an architectural approach the participants of the workshop had been exploring. Due to the similarities to SOA, microservices are also called *fine-grained SOA* and by enthusiasts *SOA done right* [JPM+18].

This basic approach of separating business capabilities into separate services is highly related to the previously discussed concept of SOA and microservices are often viewed as a specialization of the SOA style, taking the older concept to new limits [Sil16]. To achieve autonomy microservice systems are also implemented in an SNA style, where each Microservice has exclusive control over its data. This is often achieved by having an exclusive database for each microservice, in which the application stores and retrieves the data that belongs to the capabilities of the service [MSD].

But what makes microservices different from SOA and why are they seen as an evolutionary step from SOA in the construction of software architecture? Instead of using an *enterprise service bus* (ESB) which provides the infrastructure to enable the communication between different services, MSA puts a strong emphasis on the need for *light-weight* communication structures [XWQ16]. A commonly used expression in this case is that microservices are about creating *smart endpoints and dumb pipes* [FOWb]. The communication layer should be as light-weight and simple as possible, containing little to no logic, whereas the services themselves contain all the logic about which messages to handle, how to validate and how to transform messages. For this reason microservice APIs are often built using communication protocols such as HTTP and exchange their data via JSON over REST interfaces [XWQ16] in contrast to heavyweight formats or protocols such as SOAP and WSDL [JPM+18].

But why did microservices emerge at the time they did? And why did we have SOA first? The reasons are changes in the organizational and technological perspective of software construction [PZA+17].

Organizational Perspective

The concept of *domain-driven design* (DDD) is getting more and more popular. DDD places the focus of a system's design on the domain itself and encourages close collaboration

between technical and domain experts [LWO07]. Usually companies are separated into different departments handling distinct aspects of the business. Therefore if we map this structure directly to the architecture of software, we end up with separate, specialized services that have clear logical boundaries, a so called *Bounded Context* [BOU] [Eva04] [New15]. This result can also be explained by *Conway's Law* which states that *"organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations"* [Con68]. These services in turn map very well to separate microservices [PZA+17] [Fow16].

Closely related to DDD there are also changes in the way that teams are structured [Eva04]. While in the past teams were separated horizontally, by their technical function, there is a movement towards cross functional teams. Instead of having separate teams of operations engineers, backend developers, frontend developers, mobile developers, QA engineers and product managers/owners a team in the cross-functional approach consists of every kind of party that is needed for the full life cycle of a software product. This enables each team to be fully autonomous and makes it possible to achieve end-to-end product ownership for a given application [MI00].

Technical Perspective

From a **technical perspective** the evolution to microservices was heavily influenced by the appearance of several technologies and management tools [JPM+18].

In the past it would have been very challenging to manage the amount of applications that emerge from a microservice system. Each system could have specific requirements for its environment to run in. Coordination and discovery between systems would have been difficult and it would have been nearly impossible to efficiently monitor the health of hundreds of applications. These problems were however addressed by a set of technologies and tools that emerged between 2008 and 2014. *LXC*, *Docker* and *rkt* made it possible to to create runnable and deployable containers that contained the code of the application with all the additional libraries and requirements of the application [JPM+18].

Service discovery technologies like *Zookeeper*, *Eureka*, *etcd*, *Synapse* and *Consul* made it possible to automatically register the presence of one or more instances of a given service and removed the overhead of connecting large numbers of services [JPM+18]. Monitoring tools like *Graphite*, *cAdvisor* and *Prometheus* offered a way to efficiently monitor large numbers of applications [JPM+18].

From then on more and more technologies emerged that deal with challenges in the construction of large distributed systems. They address issues like *service orchestration*, *fault tolerance* and *continuous delivery* [JPM+18].

2.2.2 Goals of Microservices

The goals of microservices span over many varied concepts. These benefits origin mainly from the concepts of distributed systems and SOA, are however pronounced by the tendency of microservices to take these approaches to their extremes [New15].

Organizational Alignment

Microservices provide an easier way to align the architecture of the system with the organization's structure. Each application is responsible for a certain part of the given business domain and services correspond directly to business capabilities of the company. Having separate applications and teams responsible for these applications, which map directly to departments of the organization, makes it easier to tailor these parts of the overall system to the needs of the specific department and its business cases and enables teams of technical and domain experts to collaborate efficiently [New15] [Con68].

Scaling

One of the key goals of microservices is to enable scaling in two different forms. The first form of scaling is of an organizational nature. By creating applications with a clear bounded context these applications are independently modifiable, deployable and maintainable by a single team. This clear ownership reduces the friction and difficulties that occur when several teams with different goals are making concurrent modifications to a shared codebase. Therefore a larger number of teams is able to work on the same overall system in parallel [Kil16].

The second kind of scalability is the technical horizontal scalability of the application [Kil16]. It is common that certain parts of the system require more resources, or are under heavy or varying load. If for example the part of the system that is responsible for sending out email notifications to users can not keep up with the amount of messages to send, it is more efficient to spin up an additional instance of the specific service. The additional instance can then help with the processing of pending requests. In contrast in a monolithic system the only possibility in this case would be to start an additional instance of the whole system which consumes a larger amount of resources, because it also needs to initialize the other capabilities of the overall system [New15].

Resilience

The claim of improved resilience might sound counter-intuitive at first, because distributed systems and therefore microservice systems typically increase the number of overall failures [Kil16]. There are however arguments that this actually increases the overall resilience of the system, because it forces the architects to take failures into account early during the construction of the system. Having few and clear boundaries between the different parts of the overall system makes it possible to design sensible fallback mechanisms if an outage of certain non-critical applications occurs. This concept is known as the *bulkhead pattern* [Kil16]. For all applications, and for critical ones in particular, the possibility to provide redundancy in form of additional running instances, advanced monitoring and automated scaling greatly reduces the risk of an overall system failure [Kil16] [New15].

Ease of deployment

As previously explained large applications tend to result in a difficult and slow deployment process over time. This is especially a problem when a change to a certain part of the system needs to get deployed quickly [New15] [Fow16]. Fixing a critical bug in the UI of

a monolithic system requires the whole application to be rebuilt and redeployed, even if the change is only in one line of a very specific part of the overall functionality. In contrast when having multiple smaller applications, the problem can be fixed in the application responsible for the erroneous behavior, while all other systems are unaffected by recompilation and deployment [New15].

Optimizing for replaceability

Technologies, tools and approaches are constantly changing and evolving in the field of software development. It is possible that even if a problem is solved optimally by today's standards, two or three years later the ecosystem around software construction has evolved in such a way that there is a superior approach to solving the same problem [Fow16]. In a monolithic system, however, it can be very hard to move from one technology or approach to another one, because the switch would require a vast amount of changes across the whole system. Small and independent microservices provide the possibility to be changed and sometimes even to be fully rewritten with little required effort [New15].

2.2.3 Common Problems and Bad Practices in Microservices

Just like any other architectural approach to software construction, microservices possess some specific challenges [TL18]. Especially since many developers are still only gaining their first experience with microservices we can observe a number of frequently made mistakes.

Wrong Cuts

One of the most common and impactful problems when building a microservice system occurs when the overall domain is wrongly cut. In this case the boundaries between the resulting services have not been correctly identified and behaviour that should be in one service is spread out through several services, creating tight coupling between the different microservices. In this case the resulting system is called a *distributed monolith* and refactoring of such a system is a very costly undertaking [TL18].

Shared Persistency

In this bad practice services are decoupled on the deployment and application level, but access the same data on a shared database [New15]. This in turn leads to the different microservices being coupled again and remedies the promised benefits regarding increased independence [TL18].

Shared libraries

Usually in an attempt of engineers to follow the *Don't Repeat Yourself* (DRY) rule in software development, common functionality and objects are put into libraries which are then used by all services. However, when following a microservice approach this leads to another *bad smell* in such services [New15]. By having a shared library, teams need to coordinate their changes in the shared library. Modifications of the library for one

service are inevitably affecting all the other services and can require an update of every application that uses it [TL18].

Clients call Microservices directly

In monolithic systems clients are able to call the REST API of the system directly. However, if this approach is directly transferred to a microservice system, clients would perform direct calls to a multitude of different applications, which leads to significant drawbacks. Having a client call an endpoint on a service directly makes it difficult to move the given endpoint to a different service. This can be especially problematic when a service needs to be split into several smaller services [TL18].

There is also an additional cost with this approach. Each service needs to be able to authenticate client requests and must adhere to the client's communication mechanism and protocol. If there ever was the decision to change the way clients and servers communicate (for example by switching from REST to GraphQL), every microservice would need to change [Ric18].

Too many standards

Since microservices provide the possibility to use different programming languages and tools for different services, there is a danger of ending up with too many standards [Fow16]. If a company uses too many different programming languages and technologies it becomes difficult for developers to switch between teams or applications. Although this might be fine in the beginning when a stable team is gaining experience with the used technologies during the construction of each service, employing a multitude of technologies can pose a significant threat to the maintainability of applications once developers need to be replaced [TL18].

2.3 Microservice Patterns

Microservice Patterns aim to address many of the common problems that are specific to the MSA. In the following we have a look at the most important patterns in the context of this work.

2.3.1 API Gateway Pattern

Intent

Provides a single point of communication for clients, routes requests to other services in the system, aggregates data from various services and handles cross-cutting concerns like authentication and rate limiting [Ric18] [MW16].

Motivation

In order for a client application to be able to display a single view to the user, it often needs data provided by several microservices. In this case the client must perform several calls to different services, which leads to chatty communication and overhead from a

client's perspective [Ric18]. Especially for clients that operate on a slow or unreliable internet connection, it would be preferable to perform a single call that returns all the necessary data [MW16]. This can be especially problematic for systems with multiple different clients. Due to their different nature in terms of screen size mobile applications might not display as much data as a desktop client. The data returned by an endpoint that both services call must return the conjunction of the fields required by all clients. This leads to drawbacks for mobile clients, because of the additional overhead of unnecessary data transferred [MW16]. One possibility to solve this problem is to create separate endpoints for each client. But doing so would result in coupling services to the clients and microservices would soon become unmaintainable due to the sheer number of different endpoints [Ric18].

In addition to the disadvantages from the client's perspective, connecting clients directly to the services would also be problematic on the server's side. For a service to be invokable by clients over a public network, it must be connected to the outer network, which weakens security and makes changes to the service's endpoints more difficult. When a service is publicly exposed it needs to handle several cross-cutting concerns like authorization, response caching and rate limiting [Ric18]. Furthermore, since clients and services need to be able to communicate via a common protocol, services would need to provide their endpoints via this protocol, even if a different protocol would be advantageous for most other use-cases [Ric18].

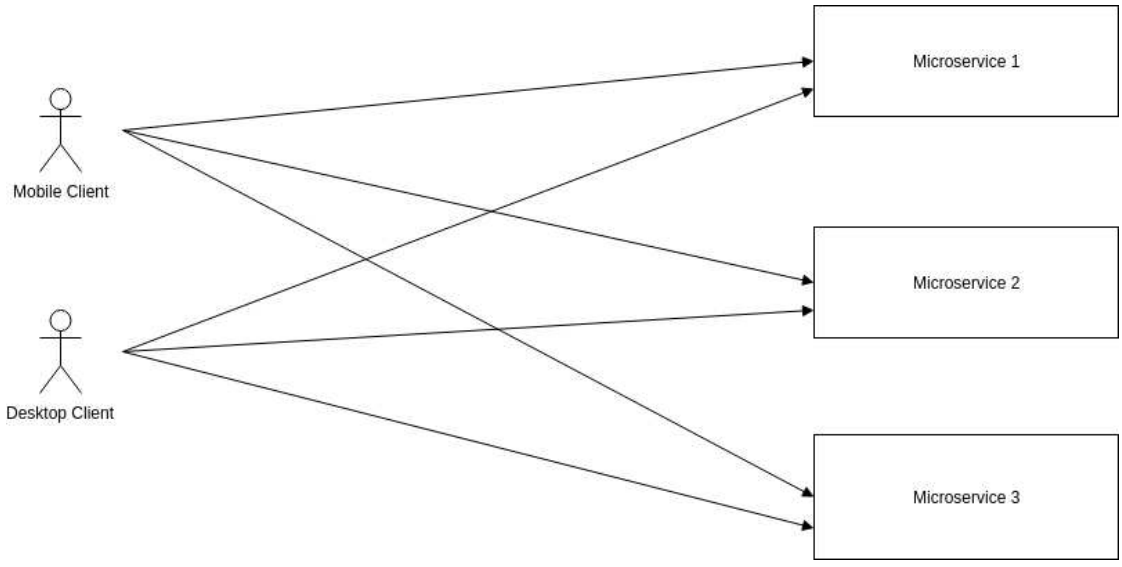

Figure 2.3: Without an API Gateway different clients talk directly to the services of the system.

A common solution in such a case is to use the API Gateway Pattern.

Structure

An API Gateway is a single component in the overall system. Clients are only able to communicate directly with the API Gateway. When the API Gateway receives a request it authenticates the client and looks up where it needs to route the request to [Ric18]. If necessary the API Gateway handles the transformation of the communication protocol between the client and the service that handles the request in the end. These transformations can reach from simple cases where only the format of the request or response needs to be slightly altered to a full translation from a synchronous request over HTTP to an asynchronous request that gets sent over a messaging queue. With the API Gateway pattern it is possible to issue a single request from the client and then fan the request out to several services to aggregate all the necessary data for the client before a response is returned [Ric18].

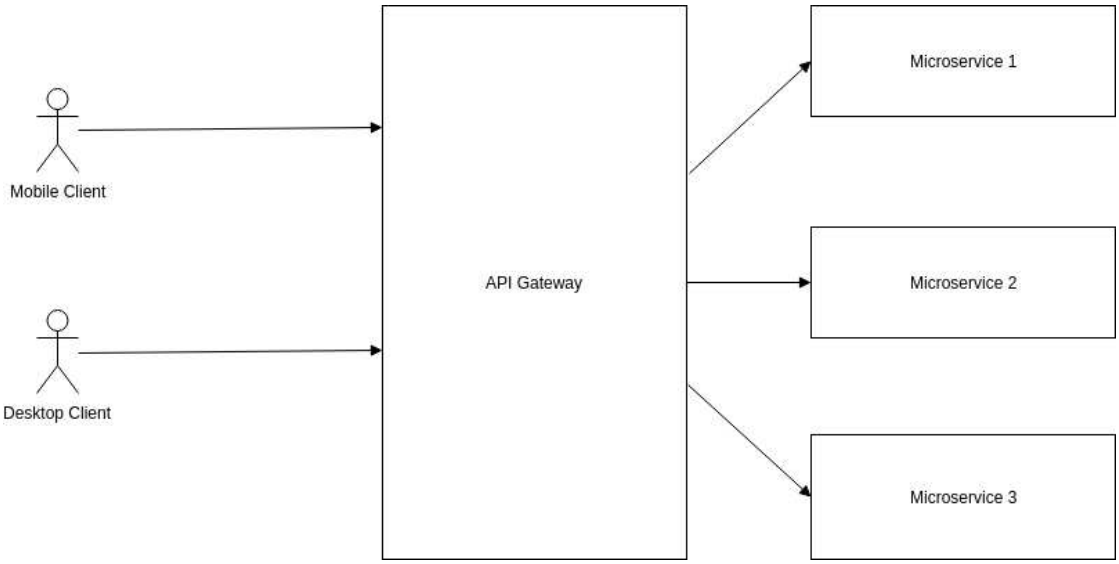

Figure 2.4: Clients communicating with the system via an API Gateway.

A common variation of this pattern is the *Backend for Frontend* (BFF) pattern. Instead of having a single API Gateway for all the different clients, each client gets its own specialized API Gateway. This approach is preferable when aggregation is done in the API Gateway and clients require different communication protocols or use different mechanisms for authentication [Ric18].

Consequences

- An **additional network hop** is required when a client's request always need to pass through the API Gateway component. However, in most cases the cost of this network hop is low due to the fact that typically the API Gateway and the microservices are co-located in the same network.

- The API Gateway is another **single point of failure** and component in a system that needs to be highly available. If this component fails, the whole system is effectively down for the clients. In the case of the BFF variation of this pattern, only a single type of clients would be affected. However, if it is possible to provide a highly available API Gateway, this component could provide fallback behavior for the services it proxies the requests to, which increases the stability of the overall system [Ric18].

- Especially in the case of synchronous requests, but also when asynchronous requests are used, there is a **danger that an API Gateway acts as a bottleneck** in the overall architecture. The architects of the system have to take care that no expensive computations are performed in this part of the system and that the system is load tested before adding additional routes to other services through the Gateway [Ric18].

- **Ownership**. If a shared API Gateway exists, ownership would either be shared between the teams of the different clients or the gateway would belong to a single team. In each case the independence of the teams is negatively affected [Ric18].

- An advantage of this pattern is the **decoupling of clients and microservices** that it provides. Acting effectively as a façade for the clients, the API Gateway is able to translate the client's preferred communication protocol to each service's own protocol and hides the existence and location of the existing microservices in the overall architecture. Providing such a decoupling layer makes it easier to change, to split or to merge existing services. In such a case only the API Gateway would need to be changed and everything happening within the system is transparent to the clients [Ric18].

- It **decreases the cost of communication for clients**. Usually a client would need to call several microservices to gather the data for a single page that is displayed to the user. If the API Gateway performs an aggregation, the client needs to only perform a single call to the API Gateway on the edge of the network. The calls from the API gateway to the different services would then be performed in the faster network of the backend system [Ric18].

- This pattern **simplifies client code** if data from multiple services need to be aggregated. Performing multiple calls that depend on each other and performing the data aggregation makes the client more complex and must be implemented in a multitude of clients. If an API Gateway is used, the gateway can perform this complex task [Ric18].

2.3.2 Circuit-Breaker Pattern

Intent

Immediately reject requests targeted at a non-responsive service to prevent cascading failures in the overall system [FOWa] [MW16].

Motivation

A typical microservice system consists of many services that interact with one another. Oftentimes the invocations between the services are of a synchronous nature, which bears the risk of partial failure. The service receiving the request might be down, overloaded or under maintenance and therefore not able to respond in a timely manner. As a result the client is blocked until it receives a response and if it acts as a server to another client, this blocking state could cascade amongst a chain of services, blocking valuable resources (in many cases threads) and in the worst case leading to a failure of the overall system [MW16] [FOWa].

The goal of using the circuit breaker pattern is to mitigate this problem by aborting requests after a certain timeout and by rejecting requests to services that appear to be unresponsive. A service is typically classified as unresponsive if it is not able to successfully serve a certain percentage of client requests within a timely manner. By rejecting further requests for a given period of time, the target service gets a chance to recover [MW16].

Structure

In its most common form the circuit breaker is a proxy implemented in a library, which the client uses to invoke the target service [Ric18].

Each request from a service passes through the circuit breaker proxy. Depending on the results of recent requests to the given target service, the circuit breaker decides whether to pass the request on or to reject it immediately. If the target service responds successfully, the circuit breaker proxy takes note that the target service is currently available and regards it to be currently in a *healthy* state. In case the request takes too long, the circuit breaker might choose to terminate the request. If a high number of such cases occur, the circuit breaker *trips the circuit* for the given service, setting the service's state to *unhealthy* [Ric18].

In a variation of this pattern the circuit breaker proxy is deployed as a side-car container in the system.

While originally implemented in the form of client libraries, a new kind of circuit breaker implementation occurred in recent years. Services in a system might be written in different programming languages, which makes it difficult to provide and maintain compatible circuit breaker libraries for each language that is used in a system. Instead of relying on such libraries, some companies use light-weight services that are independent proxy applications which handle, amongst other things, circuit breaking. These applications are typically deployed as sidecar containers [IST].

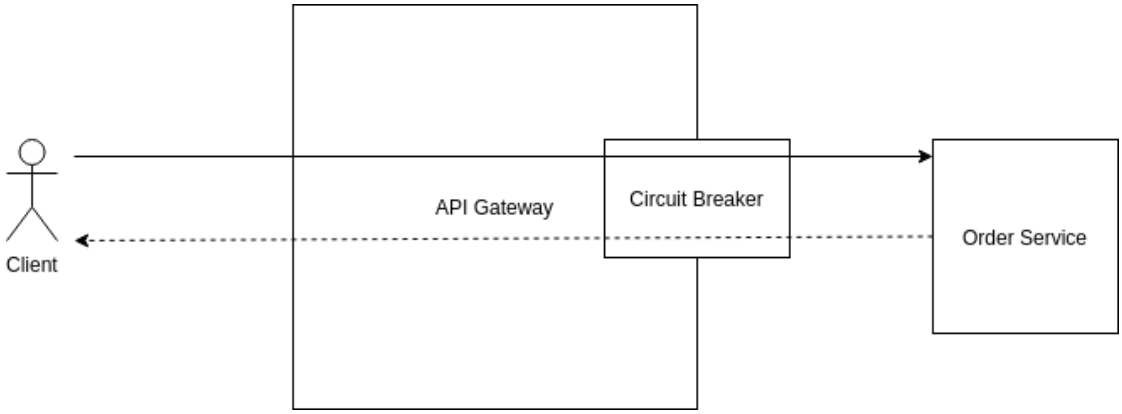

Figure 2.5: Communication via the circuit breaker pattern in a client library [Ric18].

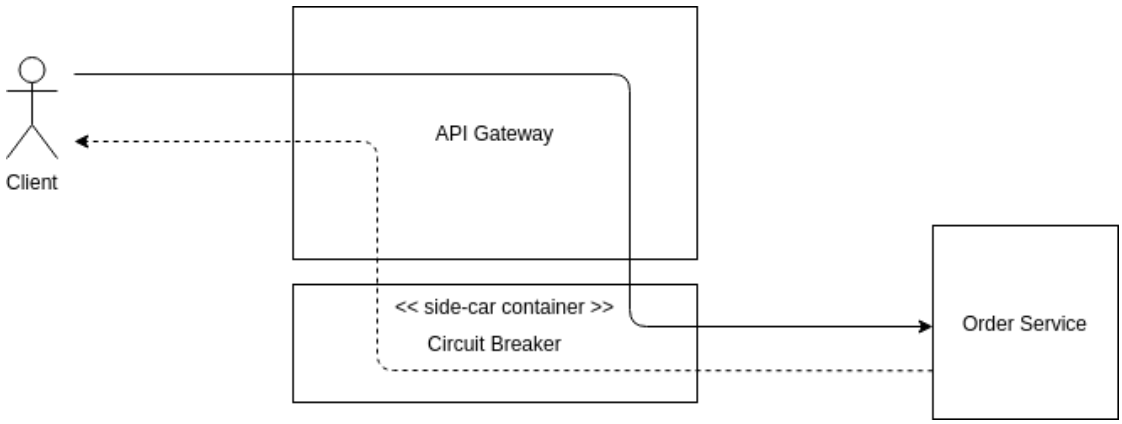

Figure 2.6: Communication via the circuit breaker pattern in a side-car container.

Consequences

- **Increases overall resilience of system** by reducing the amount of resources that are tied up when calls are failing [Ric18].
- **Decreases response times in case of failures**. It is possible to anticipate ahead of time that a call will fail. In this case the fallback behaviour can be triggered immediately without first making another call that will result in a timeout [Ric18].
- When using the circuit-breaker pattern we often face **complexity in the configuration** of the circuit-breaker. For the proxy to determine whether a call succeeded in a timely fashion, it needs information about how long the call is supposed to take in the success case. However, this is far from trivial, because different calls to the same service might behave very differently. A call that uploads a file to a service has very different performance metrics from a request with minimal or no

payload and a complex processing task on one endpoint takes much longer than a quick lookup from a cache that is performed by another endpoint.

- Circuit-breakers can also pose as a **danger to the systems stability if they are misconfigured**. Wrong settings for the timeouts or a wrong grouping of endpoints for one *bulkhead* (a group of endpoints that are disabled when any of them appears unresponsive) can result in overall failure of the service, even if just one of the endpoints produces errors and all other endpoints provided by the service would still be able to fulfill their role in the overall system.

2.3.3 Service Discovery Pattern

Intent

Provide a registry of dynamic service locations in the network [MW16].

Motivation

A typical microservice system consists of many different services and due to its dynamic nature in terms of scaling, the amount of instances of a given service might change at any point in time. If a new instance spins up or an existing instance is taken out of the system, clients need to know where to find the live instances of a given service. It is not possible to simply hard-code IP addresses and ports, as those are always subject to change. Furthermore it would require significant effort to manually set up and update all the required service locations for each service that performs calls to other services [MW16] [Ric18].

Instead, an approach that enables services to dynamically discover live instances of other services is preferable [Ric18].

Structure

At the core of the service discovery pattern is always a *service registry*. The service registry holds information about the registered services, their names and IP addresses [Ric18].

There are two typical variations how this registry can be used in the overall system. In one approach the registry is directly accessible by the services, we call this *application-level service discovery*, in the other approach registration of services in and querying of the registry is handled transparently by the platform. This is known as *platform-provided service discovery* [Ric18].

Application-level service discovery

In this variation each client talks directly to the service registry to register itself and to query the registry for the locations of other services.

In order for a service to be reachable, the registry needs information about the service's location and whether the service is still live. To achieve the former, a service sends a

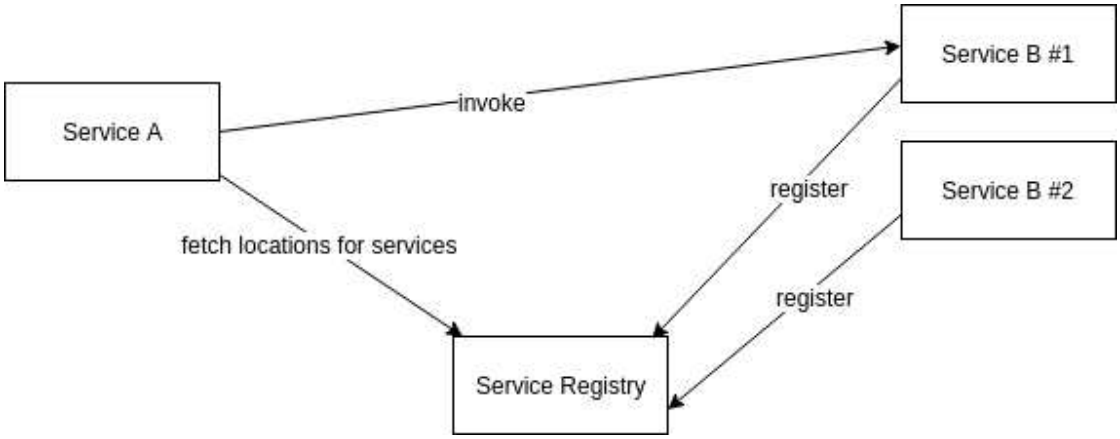

Figure 2.7: Application-level service discovery [Ric18] (modified).

request to the *registration API* of the service registry, known as the *self registration pattern*. In the process of the registration the service also submits a *health check* URL to the service registry, which is used to periodically check for the liveness of the registered service [Ric18].

When a service needs to send a request to another service, it queries the service registry for the target-service's location. Once the locations of the target service's running instances are obtained, and typically cached, the client service performs a load-balanced call to one of these. Performing discovery in this way is called *client-side discovery* [Ric18] [MW16].

Platform-provided service discovery

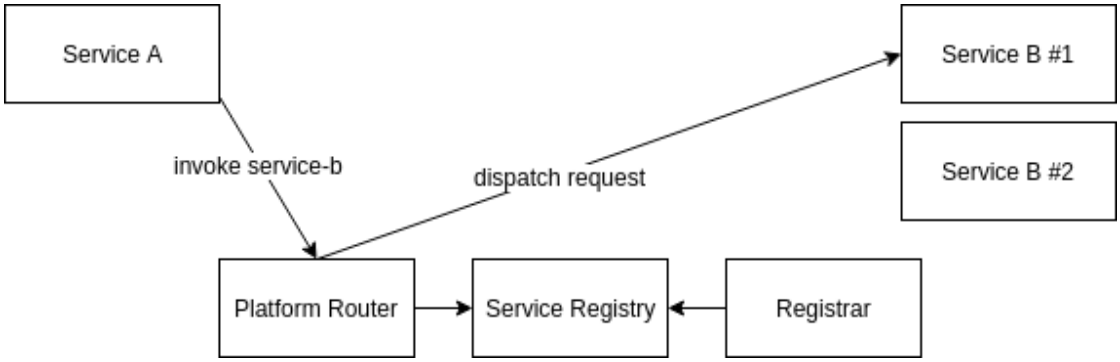

Figure 2.8: Platform-provided service discovery [Ric18] (modified).

In this approach service discovery is handled by the deployment platform (for example Docker and Kubernetes). In addition to a service registry this variation has two additional components: The *platform router* which transparently routes and load-balances requests from the services that only use DNS names to talk to other services, and the *registrar* which registers services in the service registry when a service is deployed to the platform

[Ric18] [MW16]. In contrast to the previous variation this approach uses the *3rd party registration pattern* and *server-side discovery* [Ric18] [MW16].

This approach yields the benefit of keeping the services simple and providing a service-discovery approach that is independent of the language or libraries available to the specific service. A drawback of platform-provided service discovery is that it only works out of the box with services that are deployed in and via the platform [Ric18].

Consequences

- Provides **easier maintenance** of services and **easier scaling**. By not having to worry about the actual location of a service, the number of instances that are up and consequently how to load-balance requests to a service, it becomes far easier to scale and relocate services on demand without any manual changes in other services or the infrastructure setup [Ric18].
- **Increased complexity in setup**. When service-discovery is used on an application level every component needs to be set up to register itself via the service registry and to perform calls via the information provided by the service registry. Despite there usually being many libraries that help with the heavy lifting in this case, changing existing systems might still be a time-consuming task, because every existing service needs to be changed. In the platform-provided approach existing services that are not deployed via the platform require complex solutions in order to be reachable [Ric18].
- **Can create an additional single-point-of-failure**. When the application-level approach is used, the service discovery is an additional single-point-of-failure in the system. If the service discovery goes down, clients might get by for some time by relying on cached data, after some time however no component could invoke any other component anymore [Ric18].

2.3.4 Communication Patterns

One of the many ways to differentiate between the types of communication is to separate between *synchronous* and *asynchronous* calls. These two forms of communication possess very different characteristics and both of them can be useful in a microservice system.

Synchronous remote procedure invocation pattern

Intent

In this pattern a client sends a request and blocks until it receives a response from the invoked endpoint [Ric18] [New15].

Motivation

In many cases a client service needs to send a request to a target service and requires an immediate response. When the cost in terms of latency is relatively low and a failure with

a fallback can be acceptable this pattern can provide the preferred style of communication [Ric18].

Structure

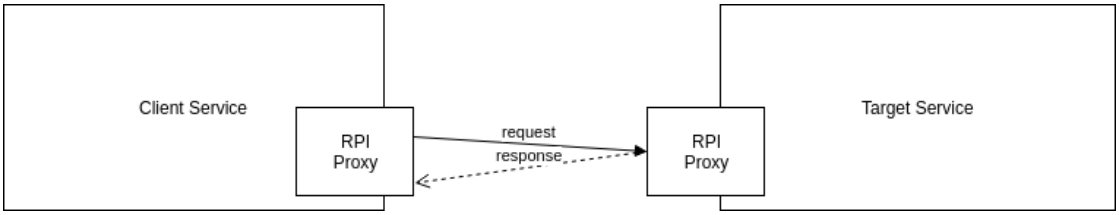

Figure 2.9: Synchronous communication between two services [Ric18].

In this pattern the client service performs its calls to the target service in a synchronous fashion via a *Remote-Procedure-Invocation (RPI) Proxy*. The calling service waits until it receives a response from the target service and only then continues its execution. Widely used protocols for such an invocation are REST and gRPC [Ric18].

Consequences

- Having many synchronous calls in a system, especially if two services invoke each other in a synchronous fashion, creates **strong coupling** between the services. The calling service depends on the availability of the target service and the structure of the response. The target service on the other hand requires a specific request format from the client service [Ric18].

- For the base case it is **simpler** to implement a synchronous call than an asynchronous one [Fow16] [New15]. When moving from a monolithic system the concept of performing a synchronous call is similar to performing a method call to another local service. With this simplicity comes also easier testability, since any endpoint that serves REST calls is easily testable with a variety of existing tools [Ric18].

- Because of the possibility of an outage of the target service in a distributed system, synchronous calls **require circuit breaking and fallback behaviour** if the target service is unresponsive [Ric18].

- For a service to be able to invoke other services a **service discovery approach is needed** [Ric18].

- Due to the nature of this invocation **only one-to-one invocations** are possible. If a service needs to send a request to several or all services, this would result in multiple calls [Ric18].

Asynchronous remote procedure invocation pattern

Intent

A service invokes one or several target services asynchronously through messaging [Ric18] [New15] [Fow16].

Motivation

In cases where no immediate response is required it can often be beneficial to send messages in an asynchronous fashion. As long as the client can be sure that the message will be handled eventually it does not care about who processes it and when the processing actually finishes. Therefore there is no need to block on the client side until a response is received. This also eliminates the need to put complex fallback behaviour into the client if the target service is not available. Furthermore, asynchronous communication and messaging can solve the problem of having multiple interested parties for one message by supporting the *publish-subscribe pattern* [Ric18] [Fow16].

Structure

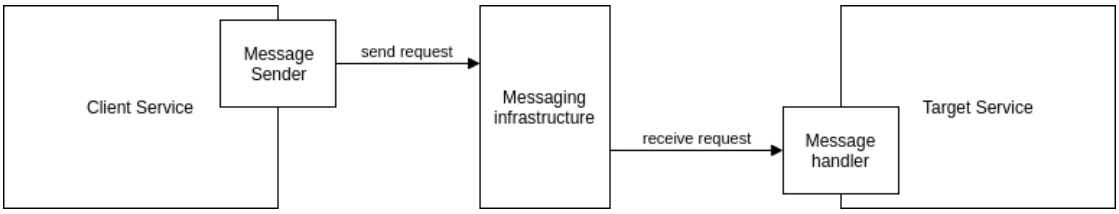

Figure 2.10: Asynchronous communication between two services [Ric18].

In this pattern the client service dispatches a request via its message sender to the messaging infrastructure, which is typically a message broker. Other services then listen to different channels of the message broker and receive messages published to relevant channels. In this case the target service fetches the client's request from the corresponding channel [Ric18] [Fow16].

This approach enables *fire and forget* communication, where a client sends a message and does not require any response and *publish-subscribe*, in which a client sends a message and an arbitrary number of services fetch the dispatched event. A more complex case arises when the client needs a response. Here the client would dispatch the initial message and also act as a target service for the response sent back by the invoked service [Ric18].

Consequences

- When asynchronous communication is used the system becomes more **resilient**. A service is independent of the availability of the services it sends messages to and therefore cases where failures of one service cascade throughout the system are eliminated [Ric18].

- By using messaging, asynchronous communication is more **flexible** and supports in addition to a request response pattern also *publish-subscribe* and *fire and forget* communication [Ric18] [Fow16].
- **Network requests are less transparent** in the code, because instead of synchronous invocations that are very similar to calls within a system, handling request and response communication requires a different approach [Ric18].
- Having a message broker creates a possible **bottleneck or single point of failure** in the system [Ric18]. However, if this danger is addressed correctly, a messaging solution can become more resilient and scalable than its synchronous counterpart [Fow16].
- Communication, especially in the case of request and response communication, becomes more **complex** [Ric18] [Fow16]. On the other hand the resulting complexity tends to be spread out through the system. Instead of a client acting as the central brain of the interaction, each component acts autonomously on the published event [New15].

2.4 Microservice Infrastructure

Distributed systems with a large number of small, different services face very specific challenges in terms of handling those services and the underlying infrastructure. Without recent new technologies in this space we would not be able to create maintainable microservice systems. One of the most important technological concept in this space are *containers* [JNS16].

Containers

Microservices within a system have very different roles and capabilities. In order to fulfill their function they might be written in different programming languages and require vastly different libraries and configuration. Instead of viewing a service merely as its application-level code, it makes more sense to regard a service as a fully functional bundle with all its required configuration [JNS16].

One of the previous ways to achieve such a bundling was to create virtual machines that contain everything a service needs to be up and running without any manual configuration. Technologies like *Vagrant* were built to simplify the process of building and maintaining such portable virtual machines, but were still too heavy-weight when dealing with numerous services.

To solve the problems of virtual machines, new technologies like *Docker* were developed. In Docker, software with its libraries and configuration files is bundled into *containers*. These containers are then executed via operating-system-level virtualization [SP17]. From a user's perspective the result is very similar to having a virtual machine image, but there are some important differences between the two technologies.

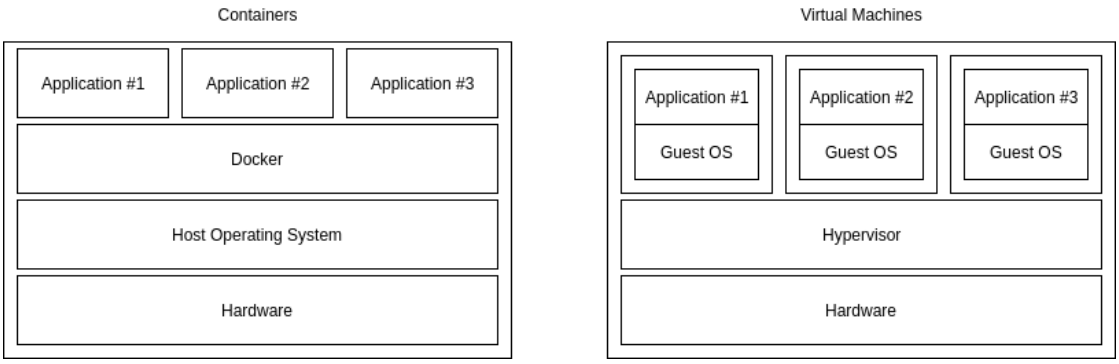

Figure 2.11: Layers of Containers and Virtual Machines [CON] (modified).

Both virtual machines and containers isolate the resources within their images, but virtual machines apply virtualization on the hardware level, whereas containers are virtualized at the operating-system level. Because of this difference containers abstract the application layer, and they are able to run on the same machine and share the operating system kernel, which greatly reduces the amount of memory that is required by each container [SP17].

Virtual machines on the other hand are abstracting the hardware of the underlying system. By using a hypervisor on the actual hardware of the system, each virtual machine runs under the illusion that it has exclusive access to the system's hardware. This approach however comes at the cost that each virtual machine needs to have its own virtual hardware layer which leads to overhead in terms of memory and cpu consumption [SP17].

2.5 Related Work

In the following section we present existing scientific work that we found to be related to our evaluation. First we have a look at research related to building specific systems via a microservice architecture. Following these we discuss articles related the general construction of microservice systems. The last topic we investigate is *Function as a Service* and *Serverless Computing*. We present each paper briefly, summarize the results and discuss how they relate to our work. In the end we give an overview of our findings.

2.5.1 Microservice Systems

In ***A microservices architecture for collaborative document editing enhanced with face recognition*** [GTI+16] a system similar to ours is built, where the goal of the authors is to create a collaborative document editing system via microservices. The system they are building consists of four different services:

- A web based user interface served by a Node.js server.

- The *Collaboration Server* which is responsible for operational transformation, which is written in Node.js and supports communication with the client via websockets.
- The *Chat Server* which enables users to chat and offers history functionality. This server is based on Java and communicates directly with clients via websockets.
- The *Face Recognition Server* which offers face recognition on images via a REST API written in Java.

The authors leverage the possibility provided by microservices to use several technologies depending on the needs of the specific service and aim at achieving a higher level of flexibility in terms of technologies, scalability and extendability of the system. After building and evaluating the system they found that it performed well in terms of synchronization with 80 users performing collaborative editing in the system. However, they also state that having a direct websocket connection from the web clients to different microservices is far from ideal and it would be preferable if there was only one open websocket. This is especially the case once the system is extended with several other services. Another finding is that microservices introduce some duplication in the overall code, which is outweighed by the advantages of microservices.

These results are consistent with what we found during the research and the construction of our system. Especially when it comes to adding functionality to the system and the possibility of scaling certain parts of the overall application, the effects of using a microservice architecture are very similar in both their and our work. However, the focus of our work is different. Instead of implementing a prototype with an emphasis on providing sophisticated functionality and a fully usable system, we are focusing on the implementation process based on an architecture that meets the standards of a production ready system. This becomes especially clear when the authors mention that their system's clients communicate with services directly and via several websocket connections instead of using an API Gateway.

Lihonga — A Microservice-Based Virtual Learning Environment [SK18] presents a microservice system that provides a virtual learning environment with a strong emphasis on the system's architecture. In this application the clients (an Android app and an iOS app) both communicate via an API Gateway with the system. Communication between the services behind the API Gateway happens in an asynchronous fashion via a message broker. This broker is connected to the different microservices and enables the architecture to provide the overall functionality of the system. According to the authors asynchronous communication provided them with a higher level of flexibility and the possibility to decouple services. The findings in this work are that a microservice architecture offers the possibility to perform iterative development on the system more easily and to choose technologies that fit the given functionality and each microservice best. Since the authors of the work focus on the architecture of the system and aimed at creating a system that is already extensible, their architecture is similar to what we identified as best practices for microservice systems and therefore the use of an API

Gateway, Docker and messaging for asynchronous communication can also be found in our systems.

Dynamic Web Service Based Image Processing System [HAJ08] introduces a distributed image processing system based on SOA. The system proposed in the work classifies algorithms for image processing into *Image Enhancement algorithms*, *Deblurring algorithms*, *Image transformations*, *Colour space conversion functions* and proposes to split algorithms into services that encapsulate common analysis functionality amongst algorithms like *Edge Detection*, *Image Segmentation* and *Feature measurement functions*. The goal of this extraction is to provide services that make up the atomic step of an image transformation and can be arbitrarily combined to achieve the desired result. It is therefore possible to create new transformations by connecting existing services and to extend transformations by plugging in additional processing steps via additional services. The system then exposes its services as web services to the public. These services can be used by a client that supplies an image and chooses transformations that shall be applied to this image. After submitting the task the client receives a unique identifier for the given task, while the task is analyzed by the rule engine and distributed to the responsible services. During the computation intermediate results and in the end the finalized images are pushed back to the client.

Despite their implementation also being composed of separate services for image processing there are significant differences to our work. While the authors use SOA and aim at creating a distributed system of reusable image processing components that provides flexible transformations via a complex rule engine, we are focusing on creating services that provide atomic image transformations with a lightweight protocol.

2.5.2 Microservice System Construction

Development and evaluation of MicroBuilder: a Model-Driven tool for the specification of REST Microservice Software Architectures [TDK+18] presents the architecture of a tool to generate microservice systems from a given specification. In the second part of this article the tool is evaluated in two different ways. The first evaluation is based on a comparison of the number of lines of code written in both a MicroBuilder specification that is used to generate the target project and the lines of code required for writing the system manually. Each of the resulting services required only about 10% of the lines of code when written via their domain specific language, leading to a total number of 152 lines for the system when using MicroBuilder and 1802 lines of code when the code was written manually. For the second evaluation 15 participants in the study built the same system according to a specification by using MicroBuilder and were required to answer a questionnaire about their experience using the system. The questions and evaluation are primarily focused on the usability of MicroBuilder. According to the findings the participants perceived using such a tool positively.

This work is especially of interest in relation to our work, because the use of a domain specific language (DSL) to generate the system via a tool is very similar to what we do

in our second approach using *GoKit*. While our tool does not provide a DSL, it provides the possibility to generate most of the boilerplate used in the system via a command line tool. The results of the paper's and our evaluation are however very different. While the authors find that using a generic generator for microservices drastically reduces their effort related metric, our result show no significant difference in effort to an implementation without such tools.

We find that there are two main reasons for these differences:

1. **Lines of Code (LoC) vs Time Spent to measure effort**. Our results might be more in line if we had also used LoC as a metric. However, our argumentation is that writing a smaller amount of code, which is more complex and therefore takes the same amount of time as writing less complex code, does not necessarily result in any advantage. This is especially the case when the resulting amount of code in the application, that is after the code generator is run, results in the same or a higher amount of code that needs to be maintained.
2. **Simple Synchronous vs Asynchronous Communication**. While the sample application using MicroBuilder exclusively uses synchronous calls over HTTP, the nature of our application requires asynchronous communication, which is neither supported out of the box by MicroBuilder nor GoKit. The need to customize and replace large parts of the generated code resulted in an increase of effort. We would also argue that the sample application used for the MicroBuilder evaluation shows the limits of such approaches, because microservice systems that offer the possibility to buy, order and pay for products, such as theirs, greatly benefit from asynchronous communication.

Apart from these differences our assumption is that we might have had similar results if both works were using similar metrics for building the same system.

Constructing a Service Software with Microservices [WF18] describes a systematic approach to constructing layered microservice systems. In their approach the authors propose a microservice architecture which not only splits microservices by the *Bounded Contexts* in the domain of the given system, but also by performance characteristics of services and by classifying services into certain types that correspond to specific layers. The layers in the example are the *Domain Microservice Layer*, *Shared Microservice Layer*, *Utility Microservice layer*. According to the paper this approach could increase the concurrency of the system and improve performance and reusability. If we take into account that our system is different from most microservice systems in the aspect that we do not have different domain objects, but split our services by the actual function that is performed on an image, we can find numerous similarities in the architectures of the two systems. Instead of services for the domain, we have a *Image Service Layer* and the other two layers would map to our infrastructure, storage and gateway services.

Workload Characterization for Microservices [UNO16] presents an investigation of the performance impact when moving from a monolithic system to microservices.

The hypothesis of this work is that although microservices originate from SOA, both approaches possess very different performance characteristics, because microservices use simpler, but more expensive, communication mechanisms like REST HTTP calls and each microservice needs to run in its own container.

In order to evaluate this impact the authors ran performance tests via the Acme Air Benchmark [ACM] for both the monolithic and the microservice implementations of the system in Java and Node.js. This combination of languages was chosen because Node.js uses a single threaded model with asynchronous execution, while Java uses a multi threaded model. They found a performance penalty of 79.2% for the Node.js version and of 70.2% for the Java version. The two main reasons for this result are an increase of the number of CPU cache misses and the increased network load especially in virtualized networks between Docker containers.

This publication is of interest to us because in our work we also evaluate the use of microservices for a system that could as well be implemented in a monolithic fashion and performance plays an important role in image processing systems. It is therefore possible to argue that this increased cost could be an argument against creating such a microservice system. However, despite not having a direct comparison between a monolithic version and the different microservice implementations of our system, we found that in the case of image processing the time spent in the image transformation outweighs all other processing tasks for a given request. This is the main reason why we could not find any differences between the various implementations. Therefore we assume that as the transformation of an image for a given task is always done in a single service, a system that performs such a CPU intensive task will not show differences in performance compared to its monolithic counterpart.

2.5.3 Function as a Service and Serverless Computing

Serverless programming (function as a service) [CIMS17] presents key characteristics, use cases, challenges and open problems in the space of serverless computing. The authors state that this approach is well suited for *bursty, compute intensive* workloads. For this reason image processing is mentioned as one of the typical use cases for this approach and an example is provided in which serverless functions receive events about new images being uploaded to a storage server (in this case an amazon S3 bucket). Upon receiving such an event the serverless function performs its image transformation on the given file in the bucket. The proposed technique is similar to what we are doing in the third approach which we are evaluating with OpenFaaS.

A Review of Serverless Frameworks [KS18] evaluates different frameworks for serverless computing. Serverless promises zero administration, infinite elasticity and minimal cost by offering a pay-as-you-use model. This leads to a decrease in administration effort for the devops and the ability to handle unanticipated loads well. According to the authors typical use cases are image processing, video processing and scientific computing.

One of the disadvantages of serverless computing is that providers try to lock-in clients. In order to avoid possible disadvantages, serverless frameworks are being developed. The cited work presents two types of frameworks that help with this problem: *abstraction frameworks* that abstract away the underlying serverless platform and provide a unified interface to each provider, and *provisioning frameworks* which are mini serverless platforms that can be deployed on any cloud provider.

For their evaluation the authors chose several aspects to identify the most promising serverless frameworks:

- *FaaSification*: How well is the mapping of existing code to functions in the framework supported? Are lots of adaptions required or can existing code be used as is?
- *Development*: Are multiple languages supported? Is there a *Function Development Kit* to help with the development of functions?
- *Deployment*: Is it easy to deploy functions?
- *Testing*: How well are existing testing strategies like unit and integration testing supported?
- *Execution*: Are multiple ways to execute a function supported? Are the provided execution mechanisms flexible enough?
- *Monitoring*: Is monitoring supported?
- *Security*: Does the framework support authentication and authorization?

By evaluating existing frameworks based on these criteria they found that there is currently no framework that clearly outperforms all or most other frameworks in each area. There are however significant differences for the specific aspects.

The criteria used by the authors are highly relevant to our work, because we are also using a serverless framework, OpenFaaS, in our third implementation approach. Despite our framework not being evaluated in this paper, it is important to take the presented evaluation aspects into account.

We find that especially *FaaSification*, *Development* and *Monitoring* need to be supported by the framework that we use. Having the possibility to easily map existing code to functions in the framework results in a speed-up in development time which is one of the factors that we measure in our work. We find that OpenFaaS supports this very well by automatically wrapping a provided function to work with its internal structures. If this was not the case our findings in terms of effort might have been different. *Development* is important to us because in image processing we can not implement every image processing functionality by ourselves, but need to use existing tools and libraries which are not always written in the same language. OpenFaaS supports this by offering the possibility to configure Docker containers that run the actual functions. This containerization enables

us to use multiple languages and to install relevant libraries for image processing in the corresponding containers. Furthermore, our evaluation requires the resulting system to provide monitoring, which is also supported by OpenFaaS.

Eventually ***Function-as-a-Service X Platform-as-a-Service: Towards a Comparative Study on FaaS and PaaS*** [AJFOG17] compares microservices deployed via a PaaS approach to Function as a Service based on the resulting performance, scalability and cost. In their experiment the authors use a prototypical system of two services, one responsible for writing data to a database, the other one for retrieving data from the database and returning it as JSON to the client. They find that despite both approaches providing sufficient scalability to handle different workloads, a deployed function with more memory resulted in a better correlation of resources and performance than adding additional memory to a service deployed via a PaaS approach. However, they faced problems with timeouts due to *cold starts*, which describe the phenomenon when a deployed function responds only very slowly to the first invocation. *Cold starts* appear on some of the FaaS platforms and can be especially problematic for synchronous invocations.

In terms of cost the findings were that FaaS is more efficient if the workloads are varying and PaaS is preferable when the workload is stable. This is because in a FaaS environment the client pays per number of invocations. While the FaaS infrastructure is capable of handling thousands of requests in seconds if needed, there is little or no cost while no requests hit the services. As a result it can be more cost efficient to use such an approach instead of running and paying for a large amount of instances to handle eventual bursts of requests. On the other hand dedicated servers are more cost efficient if they are fully utilized.

In our work the results regarding scalability are quite similar. We faced no issues with scaling between the different approaches and find it important to note that the results are dependent on the type of FaaS platform that is used. In our case OpenFaaS uses containers as a wrapper for the given functions and it is possible to specify that at least one container must always run for a given service. This eliminates the cold start problem, but would result in disadvantages regarding cost. It is also important to note that the importance of cold starts and the time needed to scale services is highly dependent on the type of invocation. While the findings in the paper are relevant to synchronous requests, cold starts are less significant for asynchronous requests that do not require an immediate response but only need to be handled eventually. In the end the authors mentioned the need for further proof of concept implementations of systems with a FaaS approach, the third implementation in our work can be viewed as such an implementation.

2.5.4 Summary

We found several articles about the construction of microservice systems. While each publication presented a successful implementation and in many cases found advantages of such an architecture, the focus of the works varied. On one end of the spectrum we saw articles that presented proof of concept implementations of specific functionalities

in a system without going into detail about the architecture and the resulting quality attributes for the overall system. At the opposite side we found articles with a stronger emphasis on an idiomatic microservice architecture. Especially in cases when the authors presented findings about such a system's structure we see a significant overlap with our findings and design and are therefore confident that our system contains no large errors in its design.

The articles about the construction of microservices showed us different approaches to create microservice systems. When we had a look at the work presenting a DSL for microservices and the impact of using a DSL when building similar system, we felt that this is most likely the publication with the most similarities to our work. However, we found that measuring the implementation effort can be done in a variety of ways, possibly leading to very different results.

In the end we looked into *Functions as a Service* and learned that the serverless approach can be especially useful for resource intensive computations and for varying load on the system. There are however differences between the platforms and frameworks for this approach. We will later see that this approach is also very promising for an image processing system.

Following this chapter we will look into the design considerations of our system. There we will see how the findings of this chapter translate to an architecture for our system.

CHAPTER 3

Design

In this chapter we will first present the requirements for our system. On the basis of these we will discuss general architectural decisions. Finally, the last part will give a short overview of the main technologies used for our implementations.

3.1 Requirements

This work started with a basic set of given requirements, which were mainly concerned with the usage of the system and its behaviour from a client's perspective. However, due to the fact that the application must be created using a microservice architecture, additional non-functional requirements in terms of the architecture arose. They are primarily related to the creation of a system that is maintainable and matches the needs and best practices of a microservice architectural style.

From a functional viewpoint the overall goal is to create a microservice system for image processing tasks. The resulting application must offer a given set of common image transformations:

- optimization of a given image
- cropping an image to fit a given aspect ratio
- performing face detection and the extraction of a portrait on an image
- taking a full-page screenshot of a website
- extracting the most significant image from a given website

Since the focus of the work lies on the integration of the service layer, we are not required to write the code for the image transformations on our own. Instead existing open source libraries to perform the actual transformation logic can be used.

Each transformation has to be implemented as a separate service to provide the possibility to easily scale the system. Furthermore each service must be packaged in a Docker container and these containers should be run via Docker Swarm [DOC] to provide redundancy and easy scaling. In order to have information about the performance and health of the system and its services each service must expose metrics concerning its current health status. For time-consuming tasks the system must be able to process images in an asynchronous manner and images must be stored on an object storage server.

From a client's perspective, the system must be invokable over HTTP via *JsonApi*. Clients must have the possibility to upload images, trigger transformations on the images and to download the resulting image after the processing.

Regarding the triggering of transformations we identified the need of clients to be able to trigger these on a large set of images in a short time. Therefore the system must be able to handle bursts of load and must not become unresponsive once it reaches its limits due to the high amount of system resources used during such transformations. In such cases it is especially important that no requests get lost and each image transformation finishes eventually.

If the load on the system for a certain task gets too high it should support a basic automated mechanism to start additional instances of the corresponding service to adapt to the current load.

3.2 Architecture

Designing, the architecture of a microservice system is a complex task with many possible pitfalls. On one hand there is a unique combination of requirements that must be fulfilled, while on the other hand we have a set of general best practices, strengths and constraints that come with a given architectural style. When faced with architectural decisions there is often no single right answer. Instead each possibility comes with its own advantages and disadvantages.

3.2.1 Core Architecture Requirements

We identified the following properties as being the most impactful on the constraints and requirements for our architecture.

Clients talk to the system over HTTP via *JsonApi*

While communication between clients and a system via RESTful APIs is often viewed as the default communication mechanism, we were explicitly asked to use the *JsonApi* specification for our system. Despite the specification's advantages, we found that there are currently several competing standards for sending JSON over HTTP. Therefore we regard the use of *JsonApi* as a property that might be subject to change in the future

and the architecture of the system should offer the possibility to easily change the way clients talk to the system.

Image processing is a resource heavy and slow process

In many applications the most common tasks are *Create, Read, Update, Delete* (CRUD) operations. In such cases the resulting functionality heavily relies on reading and writing of data, with little time spent on actual computations. In this regard our system heavily differs. Transforming an image is a computationally complex task and therefore most of execution time for each request is spent on CPU operations. This results not only in a relatively long time to process a single request, but also in limitations on the parallelization of tasks. While the bottleneck in CRUD tasks is usually IO on the system and tasks can therefore be handled asynchronously, image processing quickly takes up all of the system's computing power and a high level of parallelization would result in a general slowdown and unresponsiveness of the overall system.

Images are large in file size

Typically the different services within a microservice system communicate via messages that have a small payload. Our system differs in this regard, because images play a central role in the requests and responses produced by each component. Due to their size we regard it as crucial to minimize the load on the network by having as few transfers of images between applications as possible.

Requests will come in bursts

Clients of our system need to perform transformations on a large amount of images in batches. This aspect is especially important when we take the previous two points into account. The system must be able to deal with high numbers of requests that are very expensive regarding the system's resources.

Request types are not evenly distributed

Despite the system offering several different image transformations, it is unlikely that the amount of requests for each task will be evenly distributed. A particular client might choose to request a large number of images processed at the same time. This same client might however not be interested in performing all of the other available image transformations on these or other images at the same time. We further assume that not all image transformations are of the same interest to clients and therefore that the distribution will also not even out across multiple clients.

3.2.2 Decisions

API Gateway

We already identified the use of an API Gateway as a best practice in the chapter chapter 2. For our requirements this pattern offers a number of benefits. The use of an API Gateway gives us the possibility to expose a specific API format, in our case

JsonApi, to the client, while keeping the system's internal communication formats flexible. This service will act as the publicly reachable component and acts as a façade for the underlying system. While we do not require some of the typical functionality offered by this pattern, we can make use of the fact that all requests flow through the API Gateway by monitoring the current rate of requests of each image processing task on this component. Having such a focal point in the request flow should make automatic scaling easier.

Asynchronous communication behind the API Gateway

Using asynchronous communication within the system promises a higher degree of flexibility. Our system performs very resource intensive tasks and we do not want the services to become unresponsive. To achieve this we chose to decouple the API Gateway from the services that perform the image processing. With asynchronous communication we are able to easily queue up tasks, while the microservice that performs the image transformation is able to spend all of its available resources on its current task. Once a specific service is finished with the computations it can then fetch the next piece of work from the queue.

Because of the large amount of time that is spent on the processing of an image, asynchronous communication allows us to save system resources. Instead of having to keep a connection open between all components that are involved in a processing task, we are able to simply dispatch messages to the queue.

Furthermore, asynchronous communication enables us to shut down all instances of a given image transformation if no requests of one type are currently present. While with synchronous communication we would always need at least one running instance for each service to accept a possible request at any time, asynchronous communication enables us to scale the amount of instances to zero, queue up a possible request and then launch an instance for the request on demand.

No direct transfer of images

The large file size of images and the fact that we were already using a storage server lead to the decision that we only allow image transfers via the storage server. Instead of passing all image data along with the actual request, we chose a typical approach by only using references to images via identifiers on the storage server. This avoids unnecessary load on the network layer and in the queuing system and the problem that such large payloads are not properly supported by *JsonApi*.

In our approach the client first makes a request to upload the image to the storage server and all subsequent requests only contain a reference of the image on the server. If a service needs to perform an operation on the image, it fetches the file from Minio [MIN].

3.2.3 Alternatives

Reactive approach for task creation via the storage server

In systems that use storage servers with special functionality an alternative approach can be found for similar use cases. Some of the implementations for storage servers offer the possibility to emit events to the overall system if a new file is placed in a bucket or folder.

In such a case we could for example create a separate bucket for cropping images. When the API Gateway receives a request for cropping an image, it would then send a request to the storage server to move the corresponding image file into the *crop* bucket. With the arrival of a new file in the bucket, the storage server's trigger would automatically emit an event that a new file needs to be processed. One of the services offering these capabilities would then handle the task.

In simple cases and when the storage server that is used supports this behavior, this might be the preferred option. However, we decided against this option for several reasons. Whether or not this functionality is supported strongly depends on the technology that is used for the storage server, leading to a vendor lock in if such behavior is used. The fact that our processing tasks also take dynamic properties, like the width and height to crop to in this example, would also make this approach more complex or impossible. Furthermore we need to support tasks like analyzing a web page and downloading a screenshot of the given web location, which does not use an input image that we could put into such a bucket.

3.3 Technologies

In this part we want to give a short overview of the technologies that we used to build the system, their roles and characteristics for a microservice architecture.

3.3.1 Go

Golang is a relatively new programming language that was developed by Google with cloud systems in mind. Apart from providing an easy-to-learn and concise syntax, the designers of the language aimed at creating a performant language, similar to C++, that still offers some of the modern amenities like garbage collection. In addition to its performance, Golang comes with the advantage that no interpreter like the *Java Virtual Machine* (JVM) is required to run a Go program. This is achieved by compiling the binary to machine code for the target system. Abstaining from interpreters that consume a lot of resources offers the possibility to create very light-weight containers for microservices [AH17].

It is therefore of little surprise that many of the modern tools in cloud systems, such as Docker and Kubernetes, are built using Golang. However, the language's young age results in immature tooling and a lack of stable frameworks and libraries when it is used to build larger systems.

3.3.2 JsonApi

Endpoints that are exposed to clients by our system are implemented using the *JsonApi* specification. The goal of the specification is to provide a clear and standardized structure for responses of APIs, which is supposed to make it easier to implement an API on the server side and to consume it on the client side. The community around *JsonApi* creates and maintains both client and server libraries that can be used to easily integrate endpoints that adhere to the specification.

3.3.3 Docker

We use the popular container technology *Docker* to package each of our services with its required libraries into deployable bundles. This is especially useful in our system for the different image processing services which require vastly different configurations and libraries to be installed [JNS16].

3.3.4 Docker Swarm

In order to manage the running containers in our system we use *Docker Swarm*, which is a popular tool provided by the makers of Docker. With Docker Swarm we are able to create a description of all the required services in our system, their connections, virtual networks and their configuration for different environments. The resulting services can then be deployed to a single or even multiple machines using a simple command [DOC].

3.3.5 Prometheus

Prometheus is a popular open-source monitoring solution that does not rely on distributed storages, offers a flexible query language and supports multiple graphing and dashboarding solutions [PROa]. In contrast to other monitoring tools, the Prometheus server is used to scrape the metrics each service exposes. Such targets can either be found via service discovery or via static configuration. Due to the tool's popularity there exists a multitude of libraries for different programming languages and components that work seamlessly with the Prometheus server. These differences to traditional monitoring tools lead to a higher level of reliability when the environment is dynamic, making Prometheus a good fit for modern microservice architectures [PROa].

This technology plays an important role in the monitoring and for the automatic scaling in our system.

3.3.6 Alertmanager

One of the components that can be connected to the Prometheus server is *Alertmanager*. This application can be configured to receive and handle alerts that are produced by Prometheus or other monitoring servers [Ale].

In our system we use Alertmanager to dispatch scaling commands to the application when the load on services does not align with the number of running instances for the given type.

3.3.7 Minio

Minio is an open source storage server that offers an Amazon S3 compatible API for storing data and files. While Amazon S3 is a popular choice for large production systems in the cloud, Minio offers the advantage to use such an environment locally for development purposes and to easily switch over to Amazon S3 because of the compatible API [MIN].

We use Minio to store all image files our system operates on.

3.3.8 Faktory

Faktory is a work server that provides a repository to manage tasks produced by other applications in queues. Other services can connect to the Faktory server as workers to fetch and process tasks [FAK].

We use Faktory to store the image processing tasks that we receive from clients. Each image processing service is a worker that fetches tasks from the specific queue on the Faktory server.

3.3.9 OpenFaaS

Currently most of the major players in the cloud space provide solutions for serverless computing [CIMS17], to deploy and to run single functions in the cloud. Similar to the approach taken with Minio we use OpenFaaS to host our own serverless functions framework [OPEa].

The requirements of this chapter, the decisions that followed from them and the actual technologies lead us to the concrete implementation part of the system.

CHAPTER 4

Implementation

This chapter presents details on the actual implementation of the system. It is organized into three different parts, one for each implementation approach we used. First we will have a look at the *Implementation without a Framework*, then at the *Implementation with a Microservice Framework*, which will be brief because of its numerous similarities to the previous approach and finally at the *Implementation with a Function as a Service Framework*.

4.1 Implementation without a Framework

4.1.1 Components

The resulting system is composed of a number of different services. Figure 4.1 provides a full overview over all of them. In the following we want to have a brief look at the different services and their roles within the system.

API Gateway

In chapter 3 we already discussed the need for the API Gateway pattern in our system. Therefore we implemented such a service using *GoLang* [GOL]. It provides a single entrypoint for clients to communicate with the system and translates requests and responses between the external communication protocols of clients and the internal formats.

This is achieved by offering REST endpoints to clients. All of these endpoints follow the *JsonApi* specification to provide a consistent API format for outside requests. Each such request is translated by the API Gateway into a message for the *Faktory Server*. After dispatching the message to the queue the API Gateway replies to the client that his request has been successfully accepted and will be processed.

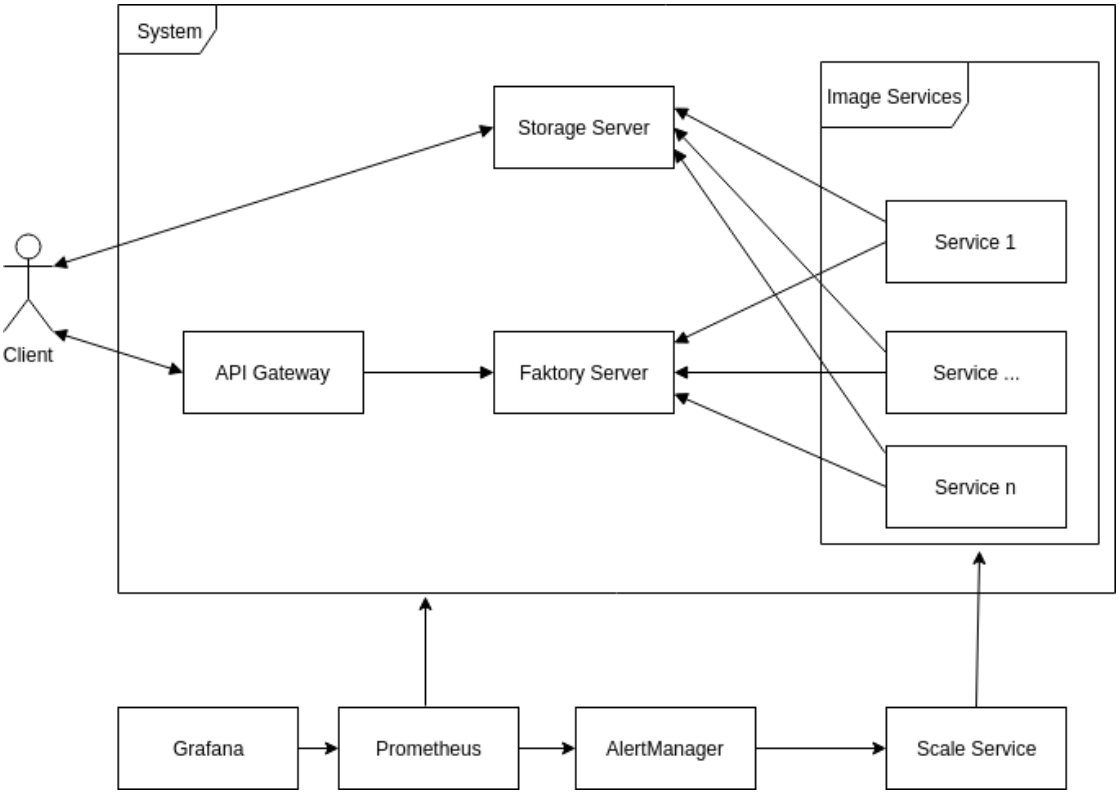

Figure 4.1: Components of the System.

Faktory Server

The *Faktory Server* is a third-party service that we run in our system. It acts as our message broker and provides multiple persistent queues for messages.

Our *API Gateway* dispatches messages to the queues of this service and the image processing services fetch available tasks from this server's queues. A task is marked as completed once a service fetched the task and acknowledges that it successfully finished the processing. If a service fails to perform a task it fetches from the queue, the task will be made available for processing by another instance. Therefore the *Faktory Server* has reliable information about the number of tasks currently in the system.

Image Services

These are the services that are able to perform the transformation logic on image files. Each *Image Service* is written in *GoLang* and provides exactly one type of image transformation. The actual logic for image processing is never written directly in the *Image Service*. Instead each service either invokes libraries or other applications co-located on the same host.

The current system contains five different image processing services and depending on

the number of pending requests there might be one or several running instances of each service.

Services fulfill their task by following a clear series of steps:

1. Fetch an open task from the *Faktory Server*.
2. If necessary, load the image specified in the task from the *Storage Server*.
3. Perform the image processing logic.
4. Upload the resulting image file to the *Storage Server*.
5. Acknowledge the successful processing of the task to the *Faktory Server*.

Storage Server

The *Storage Server* is another third-party component that we use in our system. It stores all image files that are handled within our system and provides an API to upload and download files.

This service is used by clients to upload the input images and to download the resulting images. In contrast the image processing services download input images and upload resulting images to this server.

Prometheus

The *Prometheus Server* is a third-party component that we configured to collect the metrics which we expose from the other services in our system. This service is configured with alerting rules which we use to monitor the load on the system for automatic scaling.

Alertmanager

This is a third-party component that is able to handle and dispatch alerts provided by the *Prometheus Server*. We configured the component to dispatch the alerts to the *Scale Service* with the use of webhooks.

Scale Service

Is a service written in *GoLang* that receives messages about alerts in the system from the *Alertmanager*. The actual logic to perform automatic scaling is implemented by us in this service.

Grafana

A third party service that offers the possibility to manage and create the visualization of metrics provided by *Prometheus*.

4.1.2 Flow of Processing

In general there are two large interaction flows between the components in our system. The first one being when an image processing request is handled by the system. The second largest interaction between components happens when the systems scales services up or down depending on the current load.

4.1.3 Image Processing Flow

The largest and most important processing flow in our system takes place when a client sends a request for an image transformation. This flow involves numerous interactions between components and nearly every component in the system is involved in the time between the upload of the initial image and the client finally downloading the result. Below we want to have a look at the detailed steps of this flow and how the different services play together to provide this functionality.

Optional step zero: Image Upload

Many of the services in the system require an input image to perform their image transformation. Therefore, if a client wants to request such a transformation it must first upload the image to the storage server. Once the image is successfully uploaded, the client receives an identifier that acts as a reference to the newly uploaded image.

However, there are cases in which no initial upload of an image is necessary. Tasks such as downloading the most significant image of a website or to take a screenshot of a website do not require an input image. In such cases this step can be skipped.

Step one: Client request is processed in the API Gateway

In the first step of the image processing flow a client issues a request for an image task to the API Gateway's endpoint. This request contains all information necessary to process a task. Listing 4.1 shows an example request to crop an image with the identifier *example.jpg* to *50* % of its width and *10* % of its original height.

```
curl -X POST \
  http://api-gateway-host/crop \
  -H 'Content-Type: application/json' \
  -H 'cache-control: no-cache' \
  -d '{
    "data": {
        "type": "crop_task",
        "attributes": {
            "image_id": "example.jpg",
            "width": 50,
            "height": 10
        }
    }}'
```

Listing 4.1: Client HTTP request to submit a new task.

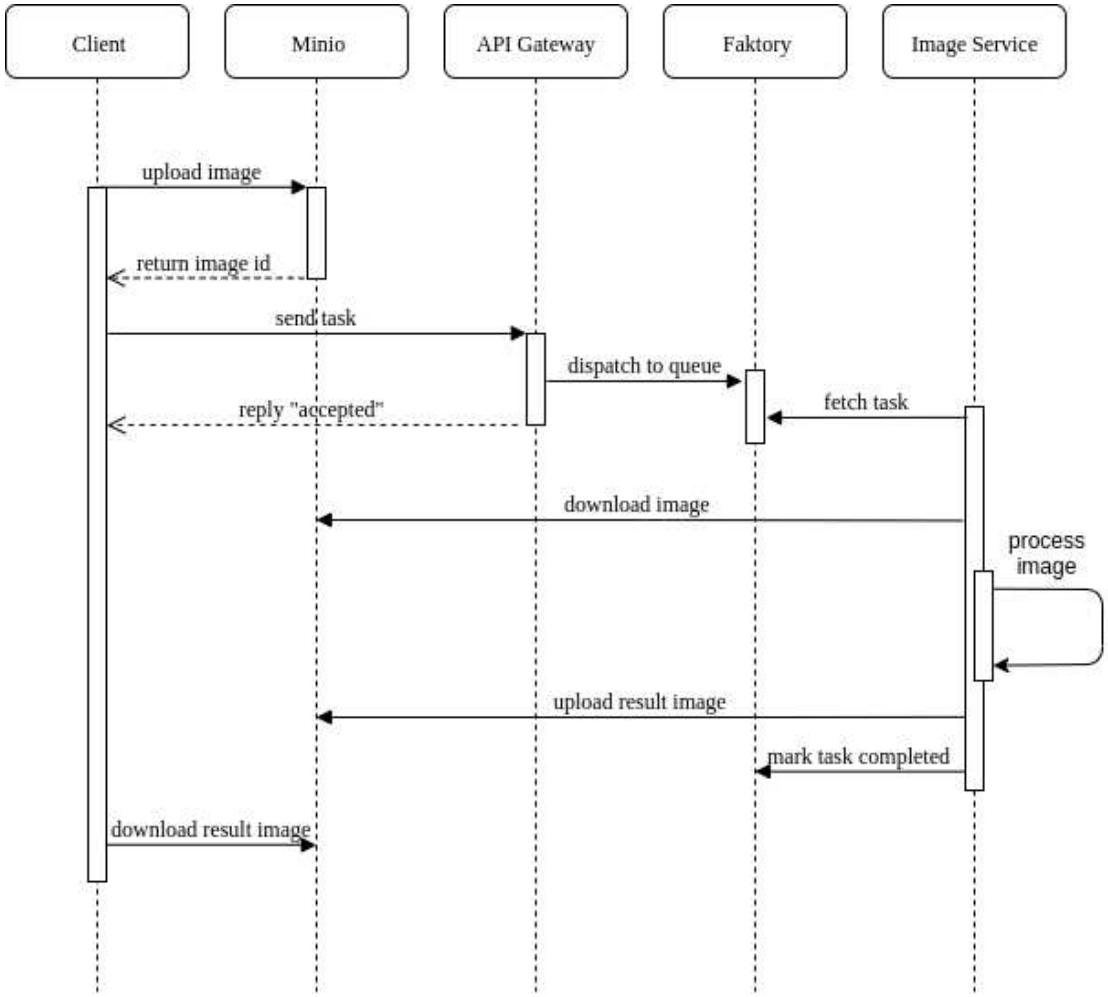

Figure 4.2: Interactions between services during image processing.

```
1 func (faktoryServiceImpl) PublishToFaktory(taskType string,
2                                            jsonTask string) error {
3   client, _ := faktory.Open()
4   job := faktory.NewJob(taskType, jsonTask)
5   job.Queue = taskType
6   err = client.Push(job)
7   return err
8 }
```

Listing 4.2: Code to submit a task to the Faktory Server's queue (simplified).

When the API Gateway receives such a task, it first needs to forward the task into the system. During the dispatching of the task, the API Gateway first enriches the request with a unique identifier. This enriched request is then serialized and passed to one of the

Faktory Server's queues as shown in Listing 4.2. For each possible image transformation type there is exactly one matching queue.

Once the request was successfully dispatched into the system, the API Gateway returns a response to the client, informing it of the unique identifier of the task and that the request has been accepted and will be processed.

```
< HTTP/1.1 201 Created
< Date: Mon, 14 Oct 2019 18:04:59 GMT
< Content-Length: 139
< Content-Type: text/plain; charset=utf-8
<
{
   "data":{
      "type":"crop_task",
      "id":"04d11ba2-4912-4e0b-b396-4d77d2bf4302",
      "attributes":{
         "height":10,
         "image_id":"example.jpg",
         "width":50
      }
   }
}
```

Listing 4.3: Example HTTP response from the system after a client submitted a new task.

It is important to note that only very little work is done in this step and the connection to the client can be closed nearly instantaneously. This behaviour provides a sharp contrast to an option in which client requests would be handled synchronously. If we used synchronous requests here, the connection from the client through the API Gateway into the system would stay open throughout the whole processing flow.

Once this step is complete the task resides on the Faktory Server's queue and is ready to be picked up for processing by the image services.

Step two: Image Service processes the request

Since every service listens to its corresponding queue on the Faktory Server, it detects the presence of a new open task and loads this task from the queue for processing. Depending on the type of the task, the service might need to fetch the input image from the storage server via the image identifier provided with the request.

Once these preparatory actions are completed, the service processes the image. Depending on the type of the task, the input parameters and the hardware used, this step can take from milliseconds to several minutes. Once the operation is completed the output image is available. This resulting image is then uploaded on to the Storage Server. After this step the Image Service notifies the Faktory Server that it successfully completed its task and starts to listen on the queue again.

Step three: Client downloads final image

Finally the client can retrieve the result. This is achieved by requesting the image with the task identifier the API Gateway issued in step one.

4.1.4 Auto Scaling Flow

The second large interaction within the system takes place when the system performs automatic scaling on its image processing services.

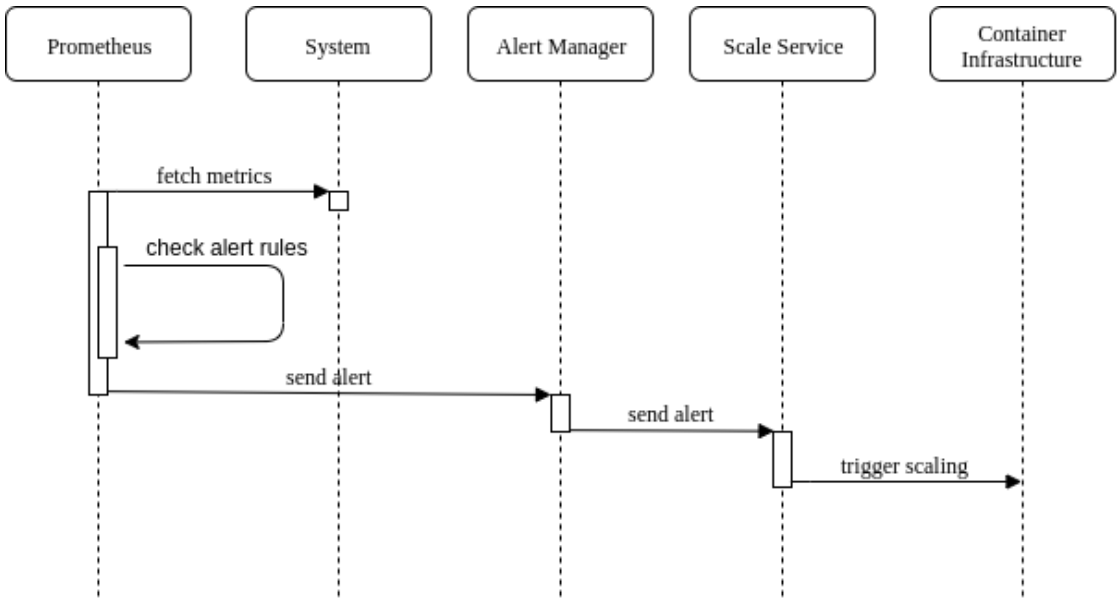

Figure 4.3: Interactions between services during auto scaling.

Step one: Prometheus detects an alert according to rule file

The *Prometheus Server* acts as a starting point for this interaction. While this service is configured to constantly collect metrics about the system and the number of open tasks, we also added alerting rules to detect whether we have too many or too few instances running for the current load.

Such rules can easily be specified in *yaml* [YAM] files by using the *Prometheus Query Language* [PROb]. Listing 4.4 demonstrates two simple rules for firing alarms to scale the number of instances of a service up or down.

```
groups:
  - name: default
    rules:
      - alert: ManyCropTasksPending
        expr: sum(up{job="crop_service"}) * 10
              < sum(crop_tasks_pending) + 1
      - alert: TooManyCropInstances
        expr: sum(up{job="crop_service"}) * 5
              > sum(crop_tasks_pending) + 5
```

Listing 4.4: Example Prometheus alert rules for up and down scaling.

If the *Prometheus Server* detects that any of the expressions defined in the alerting rules are fulfilled, the corresponding alert is set to the *firing* state and passed on to the *Alertmanager*.

Step two: Alertmanager dispatches the alert

When the Alertmanager receives an alert it uses its configuration file to determine how to handle the alert. As shown in Listing 4.5, all of our alerts are configured to be passed on to the *Scale Service* via the generic webhook configuration.

```
route:
  receiver: scale
  group_wait: 10s
  group_interval: 10s
  repeat_interval: 10s

receivers:
  - name: scale
    webhook_configs:
      - url: 'http://scale-service:8085'
```

Listing 4.5: Alertmanager configuration to use our Scale Service as webhook target for alerts.

Step three: Scale Service performs scaling

As shown in the previous step, the Scale Service implements the endpoints to act as a webhook receiver for alerts. Depending on the type of alert this service receives, it picks the corresponding service in the system and scales the number of instances accordingly.

4.2 Implementation with a Microservice Framework

The second implementation, which uses a framework, is very similar to the implementation without a framework. This is especially true when it comes to the architecture of the system and the interactions of components.

The use of this framework did hardly affect the system's architecture, because the framework itself operates only on a lower level within single services. From an architectural point of view the use of the framework is therefore transparent.

Within each service we tried to follow the conventions provided by the framework. However, we found that most of the benefits that the framework would have provided were not applicable to our system for two reasons.

1. The framework only supported a limited set of very well established technologies. In the case of the *Faktory Server*, as our means to performs messaging in the system, *GoKit* [GOK] did not provide an existing adapter. In such cases we implemented our own compatible version that implements the interfaces used by *GoKit*. While this will allow us to easily switch to another solution that is supported by the framework, we had to go through additional effort to integrate the messaging provider of our choice.

2. *GoKit* follows an approach where only a very small set of decisions is provided by the framework. It acts as an unopinionated set of libraries. For our use case we only required very few of these libraries since *GoLang* already provided a sophisticated standard library with sufficient capabilities for our system.

4.3 Implementation with a Function as a Service Framework

The concept of serverless functions works on a very high level of abstraction, which clearly shows in the implementation using *OpenFaaS* [OPEa]. For this implementation we had to loosen some of our requirements. In the case of an approach that has the goal of replacing the need to implement and handle one's own infrastructure we had no other options, but to use the technologies that are already provided by OpenFaaS. In the following we present the implementation with OpenFaaS from the user's and the developer's point of view and will then look into the details of how OpenFaaS achieves its tasks.

User & Developer View

From a developer's point of view this implementation is by far the simplest of the three options investigated in this thesis. A developer's conceptual view of the system is illustrated in Figure 4.4. There is no need to implement any services other than the five functions for image processing. Even those were drastically reduced to a single file of

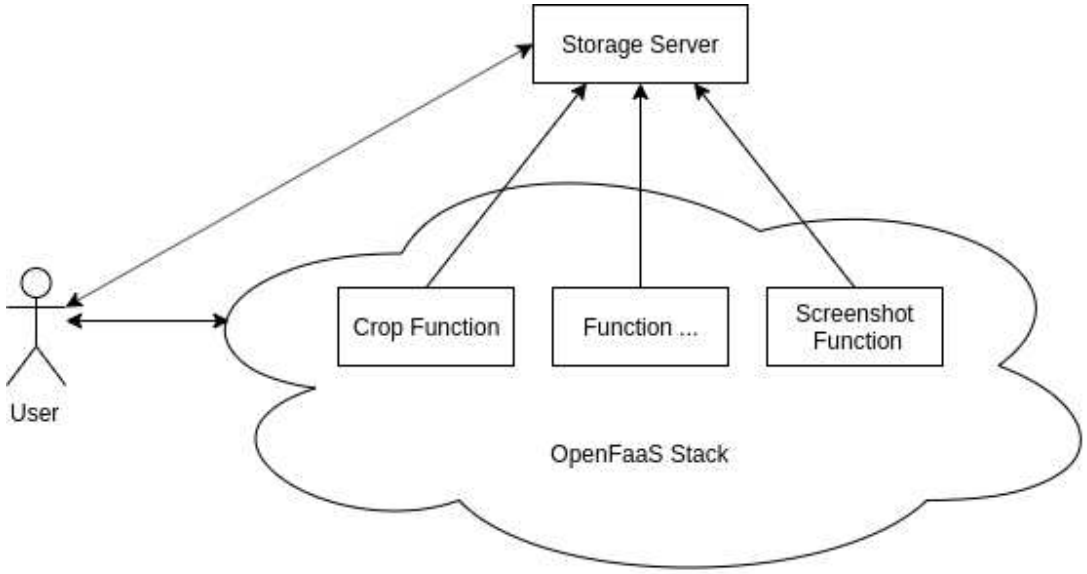

Figure 4.4: Client retrieves processed image.

about 80 lines of code each. This file only contains the innermost logic for the actual handling of a task. Listing 4.6 shows a slightly adapted version of the code from our system to integrate an image transformation into the OpenFaaS stack.

```
type Task struct {
  ID      string `jsonapi:"primary,crop_task"`
  ImageId string `jsonapi:"attr,image_id"`
  Width   int    `jsonapi:"attr,width"`
  Height  int    `jsonapi:"attr,height"`
}
func Handle(req []byte) string {
  task, err := unmarshalTask(req)
  handleTask(task)
  return ""
}
func unmarshalTask(req) Task, error {
  task := new(Task)
  task.ID = uuid.New().String()
  _ := jsonapi.UnmarshalPayload(bytes.NewReader(req), task)
  return task, nil
}
func handleTask(task *Task) error {
  // download image, transform, upload result
}
```

Listing 4.6: Code example of OpenFaaS service (simplified).

However, abstracting away a lot of the complexity also results in the framework acting as a black box. In our case this lead to difficulties with some of our image transformations, because they are dependent on the installation of other programs and libraries on the same system. Since OpenFaaS only provides one standard (Docker-based) GoLang image, we had to create additional templates in OpenFaaS. Once we solved this issue, we were able to create new base images that contained all the libraries that we needed for a given image transformation. However, handling such special cases was more difficult than implementing the service logic itself and cancelled out some of the advantages of the simple cases.

Detail View

In this part we want to have a look at the internal implementation of OpenFaaS for two different reasons. The first is that some architectural decisions within the framework might have an impact on how it could be used and for which use cases the framework might show certain strengths and weaknesses. The second reason is that we can use the internal architecture of OpenFaaS to some extent as a reference implementation for our architecture in the first two approaches.

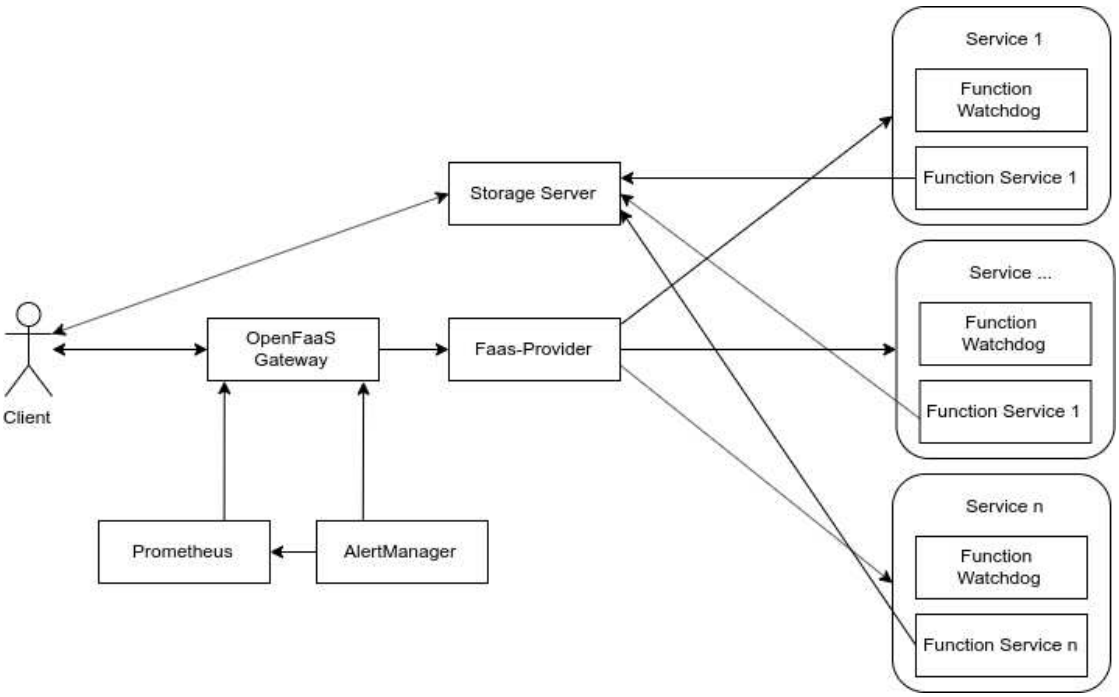

Figure 4.5: Detailed view of internal components in OpenFaaS and our system [OPEa] (modified).

When we have a look at the overall high-level architecture of OpenFaaS in Figure 4.5 we cannot help but noticing a strong similarity to our own architecture:

- The **OpenFaaS Gateway** takes the role of our *API Gateway*. Additionally it contains the logic for processing alerts from the *Alertmanager* and therefore maps also to our *Scaling Service*.
- The **FaaS-Provider** offers the middleware functionality in the system. In cases when OpenFaaS is used asynchronously the FaaS-Provider acts as a queuing system and uses *NATS Streaming* [NAT] to achieve this. This part maps to the *Faktory Server* in our system. It is interesting to note that the FaaS-Provider can be used both for synchronous as well as asynchronous communication.
- The **Services** within OpenFaaS are another interesting part of the framework. They can either be regarded very similar to the other implementations or as drastically different. While the actual source code provided by the developer is minimal and looks only like a simple function call, there is more to these services. Behind the scenes OpenFaaS is performing a lot of the heavy lifting to make the communication work. This is achieved by packaging each function with the OpenFaaS *Function Watchdog* [OPEb]. This watchdog provides a minimal HTTP server for the service, takes care of request and response translation between other services and the function and exposes health metrics, as illustrated in Figure 4.6. If we look at a service in OpenFaaS as the actual container that is running in the stack, the only differences to the services from the previous approaches are implementation details.

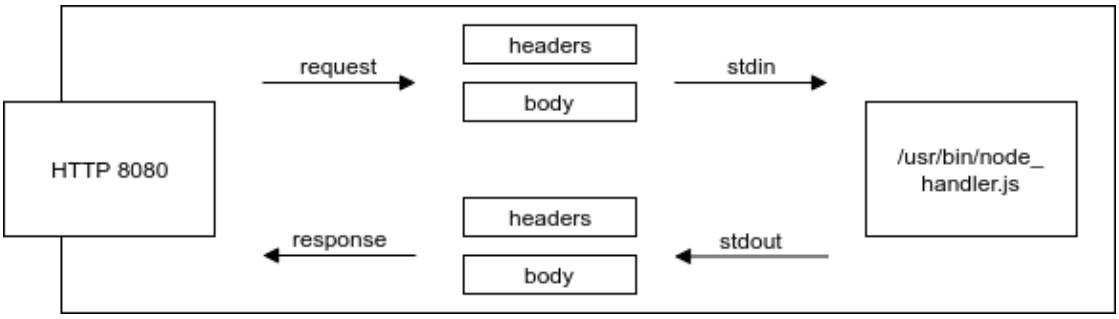

Figure 4.6: Request and response translation by the OpenFaaS Watchdog [OPEb] (modified).

- Just like in our implementation OpenFaaS depends on **Prometheus** for monitoring and in combination with **Alertmanager** provides its automatic scaling functionality.

Now that we know how the three implementations work, we would like to have a deeper look into the effects of their differences in the following chapter.

CHAPTER 5

Evaluation

In this chapter we evaluate the consequences of the three different implementation approaches. The first half of the chapter presents qualitative metrics, where we have a look at the differences in implementation effort and performance of each prototype. This is followed by the qualitative evaluation, where we discuss advantages and disadvantages of each approach and how well each of them supports current best practices.

5.1 Quantitative Evaluation

For the *Quantitative Evaluation* we use two distinct metrics. First we will have a look at the *Implementation Effort* and then in *Performance* at metrics regarding the performance of the resulting system for each approach.

5.1.1 Implementation Effort

To measure the implementation effort we were logging each change we made with a description of the change and the affected components. By doing so we created a detailed log of the time effort put into each implementation. During the evaluation we then classified each action by its description, affected parts of the system and the code changes into different categories to allow a more detailed analysis. The results of this evaluation are illustrated in Table 5.1.

While we can only observe a small difference in the overall development effort between the first two approaches, the third approach shows a clear difference. The implementation using *OpenFaaS* took only *45%* of the time compared to the implementation without a framework and *48%* of the time to the implementation with a framework. To get a better

Task Type	Without Framework	With Framework	With FaaS
Development	79.21 (69% of total)	72.36 (66% of total)	34.30 (65% of total)
Operations	20.61 (18% of total)	21.56 (20% of total)	9.20 (17% of total)
Knowledge	10.52 (9% of total)	10.35 (9% of total)	5.78 (11% of total)
Testing	4.10 (4% of total)	6.10 (5% of total)	3.75 (7% of total)
Total Time	114.44	110.37	53.03

Table 5.1: Time in hours spent on development per task type.

understanding of the cause of these differences, we need to analyse them separately by type. In the following we separate the effort into four different types of tasks:

- **Development**. Changes in the source code of the application.
- **Operations**. Tasks related to the infrastructure of the application.
- **Knowledge**. Tasks related to research about how a specific technology is used.
- **Testing**. Verification of the correctness of the system's behaviour.

Development

The measured effort for this type of task is similar for the first two approaches. Using the third approach, however, shows a significant decrease in development time. To understand these differences better we need to split the task type *Development* into two subcategories:

- **Development - Service Core**, which classifies changes within the image processing services. This category involves both the processing logic and handling of images, as well as connecting each service to the surrounding infrastructure. For a service to be fully functional, it needs to be able to communicate with the *Faktory Server* to fetch its tasks, serialize and deserialize tasks and inform the system about a successful or unsuccessful completion.
- **Development - Service Communication**, which classifies changes related to services that are responsible for the communication within the system. A typical example for this is the *API Gateway* in our system. Effort spent on developing these services is counted in this category.

After a reclassification of *Development* we end up with Table 5.2, which makes the differences clearer. While the development time spent within services is still lower for the third approach, we can observe that the differences in the *Services - Core* subcategory are at least not as drastic as the differences in the *Service - Communication* category.

Task Type	Without Framework	With Framework	With FaaS
Service - Core	50.78	51.16	32.66
Service - Communication	28.43	21.2	1.63
Development	79.21	72.36	34.30

Table 5.2: Time in hours spent on development per service task type.

The differences for the *Service - Communication* category are easily explained. While the first two approaches required an implementation of additional services such as the *API Gateway* and the *Scale Service*, *OpenFaaS* already provided such services and the corresponding functionality via the internals of the framework.

For the *Service - Communication* category we would like to point out some differences between the first and the second approach. While the overall time spent is nearly identical, the effort in these approaches was different in nature. Despite the integration of the image processing library taking the same amount of effort, we observed differences when connecting the services to the overall system. The first approach, without a framework, presented the highest level of flexibility, which lead both to certain benefits and disadvantages. On the one hand it enabled us to choose easy and well fitting solutions to perform parts of the communication, but on the other hand the lack of a given structure and standards lead to certain pitfalls and the need to refactor the code several times until an acceptable solution was found. The second approach offered less flexibility and required additional effort to implement the communication mechanism in a way that is compatible with the given framework. While the effort was spent differently in these approaches, these factors evened out in the end. In the third approach *OpenFaaS* already provided useful abstractions for the communication and apart from the translation of requests to the library calls, little work had to be done here.

Operations

This part describes the effort for infrastructure-related changes. It includes the creation of *Docker* containers, scripts to build and start the system and for setting up the monitoring solutions.

The first two approaches are nearly identical in this case, because they use the same infrastructure. The use of the *GoKit* framework was in this case transparent. However, the case is different when it comes to the third approach. *OpenFaaS* already provides all the necessary infrastructure out of the box and in a simple case there would have been no effort at all for this category, but in our case we had to adapt and extend the existing *Docker* images to work with the image processing libraries.

Knowledge

Here we compare how the time was spent on getting familiar with the technologies and on planning of implementation details.

In the first approach this part was mainly concerned with research on possible libraries and how to structure the services. The lack of constraints in this approach lead again to additional effort and responsibility for the developer.

In the second approach the set of technologies was already dictated by the framework, but additional effort had to be taken to figure out what and how the framework supports certain tasks.

We faced a similar, even more pronounced situation with the third approach. While many of the details were hidden behind the framework, every action that had to be performed required steps that were specific to *OpenFaaS*. These spanned from the installation of the framework on the host system, over the required signatures of exposed functions, to how to deploy a function and to start the system.

Testing

We could not observe any differences in this category, since all approaches offered very similar APIs to the client.

5.1.2 Performance

In this section we compare the performance of the resulting systems in all three approaches. Our tests are mainly concerned with the performance of the system and not the performance of the different image processing libraries. Therefore we chose just one type of image transformation, cropping an image, which we used across all our tests. Each test run was executed ten times and all docker containers, including infrastructure-related ones, were shut down and restarted after each execution. The processing times in this section are always the averages over all runs we performed for the given test configuration. Furthermore we used a fixed upper bound of CPU time and memory that each docker container could consume from the host system. By setting the sum of these limits to a value that is lower than the overall resources on the host machine, we aimed to isolate the performance of the tests from external factors on the host system.

Results

When the performance tests were run with a single instance of the image processing service we could

An analysis of the results of our performance tests with a single running image service in Tables 5.3, 5.4 and 5.5 shows that there are no significant differences in the processing time for each approach. During our analysis of these results we found that the execution of the image transformation, as a long running task, leads to an insignificance of minor

Requests	Processing Time (seconds)	Processing Time per Task (seconds)
10	2.81	0.28
100	27.29	0.27
250	68.28	0.27
500	131.78	0.26

Table 5.3: Implementation without Framework performance test with one running service.

Requests	Processing Time (seconds)	Processing Time per Task (seconds)
10	2.89	0.29
100	27.47	0.27
250	68.37	0.27
500	135.14	0.27

Table 5.4: Implementation with Framework performance test with one running service.

Requests	Processing Time (seconds)	Processing Time per Task (seconds)
10	2.96	0.30
100	29.33	0.29
250	72.21	0.29
500	139.11	0.28

Table 5.5: FaaS performance test with one running service.

optimizations in the different approaches. Therefore other factors, such as the performance of related services, play no significant role as long as those service do not become a bottleneck in the system.

The results of adding a second running instance of the image processing service confirms this. We can observe in Tables 5.6, 5.7 and 5.8, that by adding a second instance of the *Crop Service*, we effectively process the requests twice as fast.

This is an interesting result, because it lets us assume that we can achieve linear scalability by adding additional image processing services. However, with a high number of image processing services the storage server will become the bottle-neck of the system, because of the heavy input and output load. Once this point is reached additional measures have to be taken to perform scaling on the storage server itself. We assume that there are several ways to easily scale this service with simple strategies such as partitioning the images over several storage servers by key or by image type.

Requests	Processing Time (seconds)	Processing Time per Task (seconds)
10	1.34	0.13
100	12.17	0.12
250	30.17	0.12
500	61.33	0.12

Table 5.6: Implementation without a Framework performance test with two running services.

Requests	Processing Time (seconds)	Processing Time per Task (seconds)
10	1.34	0.13
100	12.58	0.13
250	31.03	0.12
500	61.12	0.12

Table 5.7: Implementation with a Framework performance test with two running services.

Requests	Processing Time (seconds)	Processing Time per Task (seconds)
10	1.34	0.13
100	13.2	0.13
250	32.24	0.13
500	66.41	0.13

Table 5.8: FaaS performance test with two running services.

5.2 Qualitative Evaluation

In this section we want to give an overview of the differences from a qualitative perspective. Here we discuss our experience during the implementation of the prototypes and compare their strenghts and weaknesses. Finally we evaluate whether a microservice architecture is a suitable choice for an image processing system.

5.2.1 Implementation without Framework

Implementing the system without the use of a framework shows very pronounced strenghts and weaknesses. This approach clearly provides the highest level of flexibility. No matter which surrounding and integrating technologies are chosen, writing the system from scratch allows to create a specifically tailored system. However, this flexibility comes with a high premium. Choosing this approach requires a large amount of code that needs to be written and even more important, requires a lot of planning. With the possibility to freely choose the libraries and the structure of the system, come nearly endless possibilities and a multitude of decisions. Because of the longer evaluation and

decision making process, this can not only slow down development, but also increases the chance that early decisions need to be reverted. Such refactorings of parts that strongly influence the overall structure of the system can then be very costly in terms of development time and effort. Another disadvantage of this freedom is that the resulting structure of the software is completely custom to the application and onboarding of new developers can be costly.

With this approach we found ourselves several times in situations where the lack of a given structure resulted in repeated refactorings of the same parts of the system. The main reason for this was a constant increase in understanding of the specific challenges and needs in such a system obtained during the development process. Despite following the general best practices for microservice systems, we lacked experience with a similar image processing system. Additionally we had little experience with the *Go* programming language and its surrounding ecosystem, which resulted in considerable effort to implement even trivial functionality that can often be provided by a framework and might easily be implemented by someone with more experience in the programming language. However, we found that this approach supported very specific requirements, like the use of the *Faktory Server* very well.

Therefore, our conclusions are that this approach is well suited if a high level of flexibility to integrate with non-standard systems is required. In practice this is often the case when an existing system is migrated to a microservice system. With an already existing infrastructure that uses a very particular set of technologies which may not be well supported by current frameworks, a migration to a microservice system can be easier if the system is written without any frameworks. However, in this case a detailed understanding of the needs of the resulting system and experience of the developers in the domain and with the technologies are required.

In contrast, if there are no existing services, infrastructure and strict technological requirements for the inner workings of the system, the other approaches will in many cases be a preferable choice.

In conclusion we would state this approach if there is already significant existing infrastructure and an existing system with a set of non-standard technologies that need to be supported.

Implementation with a Microservice Framework

Our experience was very different in the second approach, where we used a microservice framework. Typically such frameworks provide parts of the *boilerplate code* and a predefined set of libraries that work well with each other. In many cases developers speak of frameworks as being more or less *opinionated*, which describes the case when certain decisions are already made by the maintainers of a framework and only a limited set or a single way of achieving certain results is supported by the framework. This resulted both in advantages and disadvantages in our case, as being provided with a predefined structure and a set of libraries for the services greatly reduced the need to evaluate and

restructure parts of the application. We also assume that using a predefined structure improves the maintainability of the overall system if the system is maintained by multiple developers.

However, parts of the structure provided by the framework introduced complexity into otherwise simple parts of the system, as our image processing services by themselves were rather small and the complexity of the services was mostly in parts very specific to image processing. Another drawback that we experienced in this approach was that frameworks, such as *GoKit*, are tailored for typical use cases and support only a limited set of integrating technologies. In our case this was especially problematic when we needed to use the *Faktory Server* as a means to manage and distribute image processing tasks. Integrating a technology that is not supported by a framework can be a difficult and time consuming undertaking.

From an architectural point of view the use of a framework was transparent, as the framework only operates on the level of a single service. Whether a service was created with a framework or not, is not visible if the microservice itself is regarded as a black box.

In conclusion we would state that the use of such a framework can be justified if a large number of services needs to be created and maintained by several developers, the system performs *typical* tasks and the technologies that must be used are already supported by the framework.

Implementation with a Functions as a Service Framework

The *Function as a Service* approach is promising for a system that is mainly concerned with image transformations, as such a system revolves primarily around the execution of single tasks, a typical usecase for *FaaS*. The approach offers considerable flexibility within the functions, which contain the image processing code, while eliminating most of the boilerplate code needed for a microservice system.

In our experience, using *OpenFaaS* made the implementation and especially the design of the system easier and therefore drastically reduced the amount of time needed to implement the system. In contrast to the other two approaches, where we had to create and design the services around the actual image processing applications by ourselves, this part was already provided by the framework. Additionally, the implementation of the image services themselves required less work, since we only had to create the functions.

However, with *OpenFaaS* we needed to loosen some of the requirements regarding the surrounding system. While we used *Faktory* to manage the tasks in the other approaches, this was not possible with *FaaS*, as the messaging solution is part of the framework itself and can not be replaced.

In conclusion we experienced many benefits with *Functions as a Service* and would use this approach in all cases where the requirements and constraints by an already existing system allow us to do so.

5.2.2 Microservice Advantages & Disadvantages in Image Processing Systems

Since all three implementations are microservice systems in the context of image processing, we experienced some advantages and disadvantages of this combination across all approaches.

One of the first disadvantages that we experienced was related to the upload and transfer of images. There seems to be no standard way of creating requests that contain images for systems that offer a REST API. The content type for such an API has to be *application/json*, which is not suitable for files. Therefore many approaches with different pros and cons to try to work around this issue exist. Additionally the transfer of image files within the system is costly when it comes to network load. This leads to complex solutions to minimize the amount of image transfers between services, while a monolithic system would completely avoid this problem.

REST APIs, which are considered the default communication approach for microservice systems are also not perfectly suited to represent endpoints for image processing. While an image processing system is focused at invocations, the concept of REST maps best to representing data resources. A REST API offers endpoints under flexible paths that provide information about the type and relations of the objects that are handled at a given endpoint. However, the ways to represent actions in such an API are limited by the HTTP verbs *GET*, *POST*, *PUT*, *DELETE*. This leads to the unnatural abstractions where the API offers endpoints to create image processing transformations instead of more naturally invoking an action directly.

On the other hand we also identified several advantages of using the microservice architectural style, when it comes to image processing. Since image processing requires a large amount of system resources and most of the time to handle a request is spent in the part of the system that performs the actual image transformation, it can be advantageous to isolate this code in a separate service. Separate services for image processing then offer the possibility to be efficiently scaled on demand. This can be especially useful in terms of resources when viewed in contrast to a monolithic system in which an additional instance of the whole application must be deployed if one of its capabilities reaches the resource limits.

By comparing the execution times between a system with a single instance of a service and one with multiple instances, we also found that increasing the number of running instances of a service provides an easy mechanism to provide parallelization of the execution of tasks.

In addition to scalability, advantages of microservice systems in terms of maintainability are applicable to image processing systems. When patterns like *Service Discovery* and *API Gateway* are used, services can be added, removed and changed on demand. We experienced this during the implementation phase of the three prototypes. Once we completed the basic infrastructure and a single image processing service for each approach, we could easily add additional services without modifying other parts of the system.

Additionally the use of an API Gateway would make it easily possible to add capabilities like rate-limiting or to change the protocol that is exposed to clients, as it offers the single interface between clients and internals of the system.

5.3 Critical Reflection

Despite trying our best to achieve results which are as general and objective as possible we realize that this is not always possible and that there are parts in our approach that influence the results. In the following we mention the most significant factors.

While there are cases in which a system is built without any preexisting infrastructure, it is common that some parts of the infrastructure already exist before the project is started. In many cases this happens when a company migrates from a monolithic system to microservices. In such cases approaches that come with their own infrastructure might not be suitable and quantitatively only the effort we measured for implementing the services themselves applies in this case.

We also found that the results for the second and third approach are dependent on the actual framework that is used. There are many different frameworks and providers to choose from to build a system with a framework or to deploy serverless functions. There are also significant differences between these options and a piece of functionality that is provided by a framework or by an FaaS provider might make a big difference in the results.

Furthermore, since we created three similar systems we had to deal with a learning effect. We tried to mitigate this effect by isolating tasks which are identical in the approaches and correcting the time effort by setting it to the same value for each approach.

Eventually, there are always several ways to separate the parts of a single system into different microservices. In our case we chose to create a separate service for each image processing task. A possible alternative would be to create a general image processing microservice that is able to fulfill all possible image processing tasks and register its capabilities at the API Gateway.

Now that we have presented and evaluated the results of this thesis we want to briefly present the conclusions we draw for our work and point out possibilities for future work.

CHAPTER 6

Conclusions

This chapter summarizes the results we obtained during this thesis. In the first part we present the conclusions we draw from the combination of the research, implementation and evaluation phase. Thereafter, in the second part, we discuss possible future work that can be based on what we learned during this thesis.

6.1 Conclusions

In this work we saw that there is no one clear answer to which approach for creating a microservice system is best suited for image processing systems. Whether using no framework, a framework or the serverless functions as a tool for the implementation is the most beneficial, depends on the specific needs in a given situation. The main factors contributing to this decision are the required amount of flexibility, the presence of already existing parts of the system and the specific tools and frameworks that are used. Therefore a rigorous planning and evaluation phase should precede the actual decision on and implementation of such a system.

We saw that image processing systems pose additional challenges when implemented as a microservice system. Image transfers within the system and the resource intensive computations during image transformations are both challenges that need to be addressed at all times during the design of such a system and lead to a need for more complex and resilient design. Therefore communication mechanisms such as asynchronous communication via messaging solutions are favourable over the simpler forms such as synchronous JSON requests via REST and the amount of image transfers has to be kept to a minimum within the system. However, if the principles of microservice systems are correctly applied, we can expect the resulting architecture to yield benefits such as improved maintainability, a clear separation of concerns and better scalability.

In case one decides against the monolithic approach, our findings strongly suggest that a *Function as a Service* approach, if possible in the specific situation, suites the use case of image processing very well and can be beneficial in terms of maintainability and scalability while reducing the required development time needed.

6.2 Future Work

While this work focuses on a comparison between implementation approaches for a microservice architecture, the questions whether or not and when such an architecture is a good fit for image processing systems remains partly unanswered. Additional research would be needed that gives insights on conditions when a microservice architecture would be favorable over a monolithic system. Especially in the case of image processing and resource intensive operations there is currently no information available that helps with this decision.

In our work we only provided an exemplary overview over each implementation approach, but as we can observe in our results there is a multitude of options to choose from when an application is implemented via framework or with FaaS approaches. Depending on the chosen technology each approach might show significant differences by itself. Additional and more detailed research would be required to evaluate characteristics of the different approaches when building an image processing system.

Despite the availability of research on cloud system auto-scaling techniques [LBMAL14], further research is needed regarding systems that perform tasks such as image processing. Such long running and computationally expensive tasks have a strong influence on the feasibility of scaling approaches and would require a separate analysis. It would be beneficial to have findings on possible auto-scaling techniques in such a domain available and conditions that imply the usage of a certain technique.

To minimize the network load in our system we aimed to keep the necessary image transfers to a minimum. However, this still leaves room for improvements. Ideally transfers of large files and their location could be handled transparently. Especially when the system is spread out over several machines, sophisticated strategies regarding the physical location of files would be necessary to keep the network load as low as possible. Therefore, we see a need for further research on efficient mechanisms to transfer and store large files in distributed systems.

Eventually, when it came to creating the diagrams for this thesis, we found that there is no standard approach available to visualize microservice systems. The existing work on microservices uses different variants of UML diagrams to represent such systems and therefore we too used an approach that resembles the majority of what we found in existing papers. During our research we observed differences in how the same concept is visually presented across different works. Therefore, we see a need to either research, collect and classify existing patterns in the visualization of microservices or to create a visualization language that can be used in the context of this topic.

List of Figures

List of Tables

url breakurl [breaklinks]hyperref

Bibliography

[ACM] Acme air sample and benchmark. https://github.com/acmeair/acmeair. Accessed: 24.08.2019.

[AH17] Mina Andrawos and Martin Helmich. *Cloud Native Programming with Golang: Develop microservice-based high performance web apps for the cloud with Go*. Packt Publishing Ltd, 2017.

[AJFOG17] Lucas F Albuquerque Jr, Felipe Silva Ferraz, RF Oliveira, and SM Galdino. Function-as-a-service x platform-as-a-service: Towards a comparative study on faas and paas. In *ICSEA*, pages 206–212, 2017.

[Ale] Prometheus alertmanager github. https://github.com/prometheus/alertmanager. Accessed: 21.09.2019.

[APP] Software architecture - the monolithic approach. https://medium.com/@shivendraodean/software-architecture-the-monolithic-approach-b948ded8c333. Accessed: 04.09.2019.

[BCK12] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.

[BOU] Bounded context. https://martinfowler.com/bliki/BoundedContext.html. Accessed: 25.09.2019.

[CIMS17] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serverless programming (function as a service). In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2658–2659. IEEE, 2017.

[CON] Containers vs. virtual machines. https://blog.netapp.com/blogs/containers-vs-vms/. Accessed: 15.09.2019.

[Con68] Melvin E Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

[DOC] Docker swarm. https://docs.docker.com/engine/swarm/. Accessed: 21.09.2019.

[EG02] Fritjof Boger Engelhardtsen and Tommy Gagnes. Using javaspaces to create adaptive distributed systems. In *Proceedings of Workshop and EUNICE Summer School on Adaptable Networks and Teleservices*, pages 125–130, 2002.

[Eva04] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[FAK] Faktory. `https://github.com/contribsys/faktory`. Accessed: 21.09.2019.

[FOWa] Circuit breaker. `https://martinfowler.com/bliki/CircuitBreaker.html`. Accessed: 16.08.2019.

[FOWb] Microservices. `https://martinfowler.com/articles/microservices.html`. Accessed: 07.09.2019.

[FOWc] Monolith first. `https://martinfowler.com/bliki/MonolithFirst.html`. Accessed: 15.07.2019.

[FOWd] Monolithic architecture pattern. `https://microservices.io/patterns/monolithic.html`. Accessed: 15.07.2019.

[Fow16] Susan J Fowler. *Production-ready microservices: Building standardized systems across an engineering organization*. " O'Reilly Media, Inc.", 2016.

[GOK] Go kit - a toolkit for microservices. `https://gokit.io/`. Accessed: 14.12.2019.

[GOL] The go programming language. `https://golang.org/`. Accessed: 13.12.2019.

[GTI+16] Cristian Gadea, Mircea Trifan, Dan Ionescu, Marius Cordea, and Bogdan Ionescu. A microservices architecture for collaborative document editing enhanced with face recognition. In *2016 IEEE 11th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 441–446. IEEE, 2016.

[HAJ08] T Hemalatha, G Athisha, and S Jeyanthi. Dynamic web service based image processing system. In *2008 16th International Conference on Advanced Computing and Communications*, pages 323–328. IEEE, 2008.

[IEE] Ieee xplore digital library. `https://ieeexplore.ieee.org`. Accessed: 29.08.2019.

[IST] Istio/circuit breaking. `https://istio.io/docs/tasks/traffic-management/circuit-breaking/`. Accessed: 09.09.2019.

[JNS16] D. Jaramillo, D. V. Nguyen, and R. Smart. Leveraging microservices architecture by using docker technology. In *SoutheastCon 2016*, pages 1–5, March 2016.

[JPM+18] Pooyan Jamshidi, Claus Pahl, Nabor C Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[Jur10] Matjaz B. Juric. Wsdl and bpel extensions for event driven architecture. *Information and Software Technology*, 52(10):1023 – 1043, 2010.

[Kil16] Tom Killalea. The hidden dividends of microservices. *Communications of the ACM*, 59(8):42–45, 2016.

[KS18] K. Kritikos and P. Skrzypek. A review of serverless frameworks. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 161–168, Dec 2018.

[LBMAL14] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592, 2014.

[LS09] Olga Levina and Vladimir Stantchev. Realizing event-driven soa. In *2009 Fourth International Conference on Internet and Web Applications and Services*, pages 37–42. IEEE, 2009.

[LWO07] Einar Landre, Harald Wesenberg, and Jorn Olmheim. Agile enterprise software development using domain-driven design and test first. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 983–993. ACM, 2007.

[Mah07] Zaigham Mahmood. The promise and limitations of service oriented architecture. *International journal of Computers*, 1(3):74–78, 2007.

[MI00] Edward F McDonough III. Investigation of factors contributing to the success of cross-functional teams. *Journal of Product Innovation Management: An International Publication of the Product Development & Management Association*, 17(3):221–235, 2000.

[Mic06] Brenda M Michelson. Event-driven architecture overview. *Patricia Seybold Group*, 2(12):10–1571, 2006.

[MIN] Minio. `https://min.io/`. Accessed: 21.09.2019.

[MSD] Database per service. `https://microservices.io/patterns/data/database-per-service.html`. Accessed: 09.09.2019.

[MST] Google trends search for microservices. https://trends.google.com/trends/explore?date=today%205-y&q=Microservices. Accessed: 10.08.2019.

[MW16] Fabrizio Montesi and Janine Weber. Circuit breakers, discovery, and api gateways in microservices. *arXiv preprint arXiv:1609.05830*, 2016.

[NAT] Nats - open source messaging system. https://nats.io/. Accessed: 22.10.2019.

[New15] Sam Newman. *Building Microservices*. O'Reilly, 2015.

[NZT96] Michael G. Norman, Thomas Zurek, and Peter Thanisch. Much ado about shared-nothing. *SIGMOD Rec.*, 25(3):16–21, September 1996.

[OPEa] Openfaas. https://github.com/openfaas/faas. Accessed: 21.09.2019.

[OPEb] Openfaas watchdog. https://docs.openfaas.com/architecture/watchdog/. Accessed: 23.10.2019.

[PROa] Overview | prometheus. https://prometheus.io/docs/introduction/overview/. Accessed: 21.09.2019.

[PROb] Prometheus query language. https://prometheus.io/docs/prometheus/latest/querying/basics/. Accessed: 29.10.2019.

[PVDH07] Mike P Papazoglou and Willem-Jan Van Den Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB journal*, 16(3):389–415, 2007.

[PZA+17] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai Josuttis. Microservices in practice, part 1: Reality check and service design. *IEEE Software*, 34(1):91–98, 2017.

[RIC] Monolithic architecture pattern. https://microservices.io/patterns/monolithic.html. Accessed: 03.09.2019.

[Ric18] Chris Richardson. *Microservices Patterns - With examples in Java*. Manning Publications, 2018.

[SBA] Space-based architecture and microservices in xap. https://dzone.com/articles/space-based-microservicesgigaspaces-xap-blog-the-i-1. Accessed: 07.09.2019.

[SHA] Database sharding. http://www.agildata.com/database-sharding/. Accessed: 07.09.2019.

[SHO] Deconstructing the monolith. https://engineering.shopify.com/blogs/engineering/deconstructing-monolith-designing-software-maximizes-developer-productivity. Accessed: 06.09.2019.

[Sil16] Alan Sill. The design and architecture of microservices. *IEEE Cloud Computing*, 3(5):76–80, 2016.

[SK18] José Quenum Samuel Kapembe. Lihonga — a microservice-based virtual learning environment. In *2018 IEEE 18th International Conference on Advanced Learning Technologies (ICALT)*, pages 1–4. IEEE, 2018.

[SOA] Soa manifesto. http://www.soa-manifesto.org. Accessed: 13.08.2019.

[SP17] V. Singh and S. K. Peddoju. Container-based microservice architecture for cloud applications. In *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pages 847–852, May 2017.

[Sto86] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[SW14] A. Syromiatnikov and D. Weyns. A journey through the land of model-view-design patterns. In *2014 IEEE/IFIP Conference on Software Architecture*, pages 21–30, April 2014.

[TDK+18] Branko Terzić, Vladimir Dimitrieski, Slavica Kordić, Gordana Milosavljević, and Ivan Luković. Development and evaluation of microbuilder: a model-driven tool for the specification of rest microservice software architectures. *Enterprise Information Systems*, 12(8-9):1034–1057, 2018.

[TL18] Davide Taibi and Valentina Lenarduzzi. On the definition of microservice bad smells. *IEEE software*, 35(3):56–62, 2018.

[UNO16] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. Workload characterization for microservices. In *2016 IEEE international symposium on workload characterization (IISWC)*, pages 1–10. IEEE, 2016.

[VAMD09] M.H Valipour, B Amirzafari, K.N Maleki, and N Daneshpour. A brief survey of software architecture concepts and service oriented architecture. In *2009 2nd IEEE International Conference on Computer Science and Information Technology*, pages 34–38. IEEE, 2009.

[WF18] Feng-Jian Wang and Faisal Fahmi. Constructing a service software with microservices. In *2018 IEEE World Congress on Services (SERVICES)*, pages 43–44. IEEE, 2018.

[XWQ16] Zhongxiang Xiao, Inji Wijegunaratne, and Xinjian Qiang. Reflections on soa and microservices. In *2016 4th International Conference on Enterprise Systems (ES)*, pages 60–67. IEEE, 2016.

[YAM] The official yaml web site. `https://yaml.org/`. Accessed: 29.10.2019.