# Design, Implementation and Evaluation of a System to Create a Data Set Supporting Research in the EnergieBroker Platform

**Master Thesis**

for the

**Master of Science**

in Computer Science

at RheinMain University of Applied Sciences

by

**Niklas Sauer**

January 11th, 2022

| | |
|---|---|
| **Project Duration** | 6 Months |
| **Student ID** | 1209772 |
| **First Reviewer** | Prof. Dr. Heinz Werntges |
| **Second Reviewer** | Prof. Dr. Robert Kaiser |
| **Supervisor** | Johannes Kaeppel |

## Abstract

# Contents

# Acronyms

| | |
|---|---|
| **SOA** | Service-Oriented Architecture |
| **API** | Application Programming Interface |
| **IP** | Internet Protocol |
| **SRP** | Single Responsibility Principle |
| **DDD** | Domain-Driven Design |
| **ACID** | Atomicity Consistency Isolation Durability |
| **NIST** | National Institute of Standards and Technology |
| **VM** | Virtual Machine |
| **OS** | Operating System |
| **IaaS** | Infrastructure as a Service |
| **PaaS** | Platform as a Service |
| **MAC** | Mandatory Access Control |
| **DoS** | Denial of Service |
| **SELinux** | Security-Enhanced Linux |
| **NSA** | National Security Agency |
| **VCS** | Version Control System |
| **OCI** | Open Container Initiative |

# List of Figures

# List of Tables

# Listings

# 1. Introduction

## 1.1. Motivation

## 1.2. Goals and Scope

## 1.3. Thesis Overview

# 2. Theoretical Framework

## 2.1. Microservice Architecture

Popularized by companies like Amazon, Netflix, Uber, LinkedIn and SoundCloud, the microservice architecture has emerged as a pattern to avoid the problems of conventional monolithic designs [18, p. 847] [23, p. 584]. This section provides an overview to the problems faced in monolithic applications, distinguishes the microservice architecture from a traditional Service-Oriented Architecture and lays out its core principles, while noting some of the newly introduced challenges.

### 2.1.1. The Monolith Problem

A monolith is a software application whose modules cannot be executed independently [7, p. 1]. Hence, a monolith is characterized by requiring to be deployed as a united solution [6, p. 24]. Based on this definition, a set of obstructive characteristics inherent to monolithic applications can be derived:

**Maintainability**

When developing an application with a single large codebase, it naturally becomes harder to maintain and comprehend [7, p. 2]. The latter is especially true for beginners, slowing down their productivity [6, p. 24]. Further, refactoring changes may touch many parts of the software which might lead to a situation in which refactoring is ignored because it becomes too risky [9, p. 35].

**Dependencies and technology lock-in**

Monoliths typically suffer from the "dependency hell" problem where an application's modules depend on conflicting versions of a shared library [7, p. 2]. In cases where this is solved, the likelihood of having to make many changes to update to a newer version of a library again increases with a growing codebase. Next to that, it becomes very difficult to change the technology stack, leading to a lock-in and forcing developers to use the same language in every problem domain [6, p. 24] [7, p. 2].

**Deployment**

Rolling out a new application version requires the complete set of services to be restarted, regardless of whether a service has been altered or not [7,

p. 2]. Similarly, failure of one service leads to downtime across all services [21, p. 970]. The deployment can therefore be viewed as a single point of failure [23, p. 584]. Moreover, deployments are likely sub-optimal due to conflicting resource requirements (e.g. CPU vs. memory-intensive). Developers often have to compromise with a one-size fits all configuration [7, p. 2].

**Scaling**

By combining multiple services into a single process, scaling can lead to resource wastage since the whole application needs to be scaled up even if an increase in traffic stresses only a subset of modules [18, p. 850] [7, p. 2]. Less popular services consume unnecessary (idle) amounts of resources [23, p. 584]. Setting appropriate scaling thresholds also becomes more challenging because different components may have different resource requirements [6, p. 24].

Kalske, Mäkitalo, and Mikkonen, however, acknowledge that the monolith approach might be the correct choice if the codebase is relatively small or the need for fine-grained scaling has not come up [9, pp. 34, 36]. Villamizar et al. add that monoliths are faster to set up [23, p. 589]. Empirically, most organizations start with something big and slowly transition to a decomposed architecture when scaling problems arise [20, p. 113] [23, p. 590]. Yet, it can be argued that an organization should spend more time on the design upfront since it is easy to introduce tight coupling, hereby hindering future refactoring endeavors [9, p. 34].

### 2.1.2. Definition

A microservice-based application is one in which the core functionality has been decomposed into many small units that can be independently developed and deployed [11, p. 43] [21, p. 970]. Each unit, a *microservice*, is modeled around a single, clearly defined set of closely related functionalities that can be used independently over the network through a well-defined interface [1] [19, p. 56] [22, p. 176] [5, p. 30]. This definition implies that microservices:

- use independent codebases, thus can build on different technology stacks
- run in distinct processes, thus can fail and scale independently
- are decoupled but can be used as building blocks to form larger services

[1] Cerny, Donahoo, and Trnka draw a comparison to three Unix ideas: a program should fulfill only one task well, be able to work with other programs and use a universal interface [5, p. 31].

### 2.1.3. Decomposition Techniques

Spitting an application into services should happen along the lines of related processes that can be carried out in isolation. It is not about arbitrarily distributing features across services [19, p. 61]. As in traditional software engineering, the term cohesiveness is used to indicate that a service implements only functionalities strongly related to the concern that it is meant to model [7, p. 2]. Various techniques are cited to determine the breadth of concern:

**Single Responsibility Principle**
: Defines a responsibility of a class as a reason to change and states that a class should only have one such reason [13, p. 36] [20, p. 116]. Is analogously applied to microservices. Leads to a large amount of services.

**Y-axis of scale cube**
: Splits an application into distinct sets of related functions. Each set is implemented by a microservice. In a verb-based approach, sets consist of a single function that covers a specific use case, whereas the noun-based approach creates sets of functions responsible for all operations related to a particular entity [13, p. 36]. Leads to large and medium amount of services, respectively.

**Domain-Driven Design**
: Refers to the application's problem space, i.e. the business, as the domain. This domain consists of multiple subdomains (e.g. product catalog, order management). Each subdomain is represented by a microservice [1, p. 3]. Leads to a small amount of services.

add cita-tion

Irrespective of the technique chosen, the overall goal should be to minimize later interface changes, i.e. to establish proper service contracts [6, p. 26].

### 2.1.4. Service Registry Pattern

The law of conservation of complexity states that the complexity of a large system does not vanish when the system is broken up into smaller pieces. Instead, the complexity is pushed to the interactions between these pieces [13, p. 38] [20, p. 114]. Applied to microservice-based applications, this means that developers need to deal with the challenges innate to distributed systems [6, p. 24] [23, p. 589]. One such challenge is the fact that services can no longer be invoked through language level method calls but rather only through the network. Moreover, given the need for scaling, clients are now required to make requests to a dynamically changing set of service instances. And since it is unfeasible to run these instances at fixed locations [2], a pattern known as the service registry is commonly employed [13, p. 37].

A service registry acts as a database of services, storing the various instances along their locations, i.e. the IP address and port number. Instances are added on startup and removed on shutdown. Keeping this in mind, two types of service discovery mechanisms are distinguished [11, p. 46]:

**Client-side**

To contact a service, clients obtain the locations of all service instances by querying the registry. The client then needs to perform a load balancing algorithm to decide which instance will be contacted.

**Server-side**

To contact a service, clients make a request to the service's load balancer which runs at a well known location. This load balancer queries the registry and forwards the request to an available instance.

In any case, the service registry is a critical component and thus, must be highly available.

### 2.1.5. API Gateway Pattern

Depending on the decomposition technique chosen (see subsection 2.1.3), microservices might provide very fine-grained APIs. In turn, this means that clients may need to interact with multiple different services to carry out a high-level business process. To hide this complexity from clients and ensure consistent behavior, a pattern known as the API gateway is employed.

An API gateway represents the single entrypoint for all clients in which some requests are simply proxied while others fan out to and consume multiple services. In the latter case, gateways can be viewed as orchestrators and as such, typically do not have persistence layers [23, p. 585]. They may, however, cache responses. add citation Another important task in orchestrating multiple microservices is managing distributed transactions [3], i.e. ensuring atomicity guarantees for a set of distributed resources [5, p. 32]. Finally, gateways may also deal with generic features such as authentication and authorization or implement the circuit breaker pattern to prevent a service failure from cascading to other services [9, p. 41] [13, p. 37].

[2] In the event of a network partition, a standard recovery process would attempt to restart a service in the healthy partition, thus leading to a new network location for this service.

[3] Distributed transactions are commonly implemented using the two-phase commit protocol. The no-ACID transaction type has also been proposed for this context, which is known as a compensation transaction [5, p. 32].

It shall be noted that gateways incur a performance penalty because they introduce an additional network hop [13, p. 37]. This is generally true for proxying gateways. Orchestrating gateways, on the other hand, have the potential to decrease latency since the various requests being collapsed now already originate from the target network. In both cases, the performance degradation will heavily depend on the system's interconnectedness [7, p. 9].

add citation

### 2.1.6. Database-per-Service Pattern

To keep microservices loosely coupled, a pattern known as database-per-service is employed. This pattern calls for each microservice to have its own database, compared to sharing one across multiple services. Sharing is achieved by making the data accessible via the service's API [13, p. 36] [19, p. 59]. Messina et al. discern between three levels of pattern conformity [13, p. 37]:

**Private tables**
Each service has a set of tables private to that service.

**Private schema**
Each service has a database schema private to that service.

**Private database**
Each service has its own database.

While this pattern contributes to service intimacy, it comes at the cost of having to redefine data models and restate business rules across services [4] [5, p. 30].

### 2.1.7. Delineation from Service-Oriented Architecture

Historically, the complexity of monolithic applications (see subsection 2.1.1) has already been addressed using different Service-Oriented Architecture (SOA) approaches that also decompose a large system into many smaller services. Academia, however, is undecided whether microservices should be considered as a subset or superset of SOAs or whether it constitutes a new, distinct idea [23, pp. 584–585] [5, p. 30]. The systematic mapping study conducted by Cerny, Donahoo, and Trnka in [5] spends a great deal on contrasting the two architectures. A short summary is given in the following.

[4] In Domain-Driven Design, the concept of a Bounded Context describes that services operate with business objects in a specific context and therefore, only need to model a subset of the global object's attributes [5, p. 30].

In both approaches, services cooperate to provide functionality for the overall system. However, the path to achieving this goal is different. This is most obvious when looking at the interaction patterns between the services involved. SOAs rely on orchestration, whereas microservice-based applications prefer choreography. The former expects a centralized business process to coordinate activities across services and combine the outcomes, whereas the latter expects individual services to collaborate based on their interface contracts. Orchestration differs from choreography with respect to where the logic that controls the interactions should reside. The two terms describe a centralized and decentralized approach hereof, respectively.

An important remark that Cerny, Donahoo, and Trnka make, is that orchestration through an integration layer, such as a messaging bus, oftentimes leads to a situation in which the system parties, i.e. the services, agree on a standardized representation of the business objects they exchange. The system ends up with one kind of business object each. This is known as a canonical data modal. The danger being that a change in one of the business objects necessitates changes in all of the services that deal with this object. As a result, deployments in a SOA again happen in a monolithic fashion. In the microservice architecture, such a change can at most propagate to the API gateways (comp. subsection 2.1.5), though this is less likely because gateways integrate services based on their interfaces, not on their models.

Lastly, albeit obvious, the biggest drawback of SOAs and their centralized orchestration model must be stated. Having a centralized business process integrate all services again entails having a single point of failure.

## 2.2. OS-level Virtualization

Virtualization describes the act of creating a virtual version of something [4, p. 2]. In the context of computing, it specifically refers to hardware resources such as CPU, memory, storage or network devices to create complete virtual instances of computer systems [17, p. 21]. This section sheds light on the motivation behind virtualization, gives a brief overview of the traditional approach hereto and focuses on a more recent and lightweight alternative in the remainder.

### 2.2.1. The Need for Virtualization

Server consolidation attempts to maximize resource utilization, while also reducing costs through energy savings [5] [24, p. 233] [8, p. 2]. This is achieved by using fewer

update footnote figure a citation

[5] About 10% of the world's energy consumption stems from data center operations [16, p. 1]. If resources can be used more efficiently, carbon emissions could be lowered.

physical servers to host the same number of applications. But without an isolation layer there are no guarantees that an application from one user will not interfere with that of another [24, p. 233]. Isolation and multi-tenancy are exactly those traits promised by virtualization technologies [17, p. 21]. Coupled with a software layer to provision resources on demand, virtualization yields the elastic multi-tenant model embodied in cloud computing [10, p. 203] [2, p. 81] [15, p. 24].

### 2.2.2. Comparison to Hardware Virtualization

In traditional hardware virtualization, a so called hypervisor makes siloed slices of hardware available in the form of Virtual Machines (VMs) by emulating the underlying physical resources. Each of the VMs (*guests*) running on the physical hardware (*host*) comes with its own full-fledged OS. Two types of hypervisors can be discerned [12, p. 2] [8, p. 1] [14, pp. 386–387]:

**Type 1 (bare-metal)**
: Operates directly on top of the host's hardware, not requiring a host OS.

**Type 2 (hosted)**
: Operates as a software layer on top of the host's OS.

Although virtualization through emulation enjoys great popularity [6], it comes at the cost of efficiency. On the other hand, the overhead introduced by OS-level virtualization can be considered almost negligible [14, pp. 386, 392].

Whereas hypervisor-based virtualization provides strong isolation guarantees between systems, OS-level virtualization only strives to isolate processes. Such an isolated process is known as a container [7]. Here, the isolation mechanisms are provided as kernel features that establish an abstract and protected view on the OS, making two containers unaware of each other or any of the other processes running on the host [12, p. 2] [8, pp. 1–2].

Even though both technologies enable a safe multi-tenant model of hardware by confining parts of the application infrastructure, Pahl ascribes them to different use cases, arguing that VMs are about hardware allocation and management, while containers are tools for delivering software. He further compares them to the concepts of IaaS and PaaS, respectively [15, p. 24]. Eder and Merkel et al. see the potential for the two technologies to complement each other since the resource footprint of

[6] Hardware acceleration for hypervisor-based virtualization has even been incorporated into commodity processors [24, p. 233].

[7] Containers are known as jails or zones in the FreeBSD and Solaris OSes, respectively [8, p. 2].

containers is minimal and its security profile is still slightly worse than that of VMs [8] [8, p. 6] [12, p. 2]. add citation

Lastly, it shall be noted that the shared kernel approach of containers means that processes being isolated need to be compatible with the host's kernel and CPU architecture [14, p. 386] [8, p. 2]. In other words, it is not possible to e.g. run Windows containers on Linux hosts or deploy x86 containers on ARM because no emulation is taking place. On the flip side, the shared kernel approach allows kernel security patches to be applied without having to modify a container. This is not the case for a VM, where the underlying machine image would have to be rebuild first [8, p. 3]. Moreover, sharing a kernel allows container-based solutions to achieve a higher density of virtualized instances when compared to hypervisor-based solutions [14, p. 386] [10, p. 204]. Containers are also a magnitude faster to start and stop since they are essentially just processes that have to be spawned and terminated, whereas the OS of a VM needs to be fully booted and shut down [12, p. 2] [8, p. 2]. Similarly, being a process means that containers do not occupy any resources when they are not executing. VMs will idle until they are shut down [12, p. 2].

### 2.2.3. Linux Kernel Containment Features

As previously hinted at, containers rely on the host's kernel to sandbox processes from each other. In Linux, the set of containment features include but are not limited to:

**Chroots**

`chroot()` (from *change root*) changes the root directory of the calling process and all of its children [9]. This is used to restrict a container's view on the filesystem. However, it cannot be considered a security feature because there are multiple intentional escape hatches [8, p. 3]. update citation

**Namespaces**

Namespaces provide one or more processes with private and restricted views towards certain global system resources [14, p. 387]. Changes to resources within a namespace are only visible to processes that are members of that namespace. The namespace type (e.g. `ipc`, `net`, `pid` or `user`) indicates which kinds of resources are being isolated. As an example, this feature allows update citation processes within a container to have identifiers that are already in use on the

[8] A good example of combining VMs and containers is given by the Google Kubernetes Engine. Here, a cluster of hosts is created on the basis of VMs which then exclusively runs software in the form of containers.

[9] Some people trace the inspiration for containers back to `chroot()`, which was originally introduced with Unix 7 in 1979 [2, p. 82].

host system or within other containers [8, p. 3]. The network namespace can provide a container with its own network device and virtual IP address, whereas the user namespace would be used to ensure that a container's user database is separated from that of the host which means that the container's root user privileges cannot be applied on the host [12, p. 1].

#### Control groups

Control groups, usually referred to as cgroups, provide resource accounting and limiting for a set of processes [14, p. 387] [12, p. 1]. Even though they are not mandatory for process isolation, cgroups can ensure that one container cannot starve another (e.g. in a DoS attack), as well as to help realize cost-accounting multi-tenant environments [8, p. 4].

#### Mandatory Access Control

Mandatory Access Control (MAC) describes a concept in which access to or operations on a particular resource is granted based on different authorization rules (*policies*). On Linux, the two prevalent implementations hereof are AppArmor and SELinux [10]. In the context of containers, MAC is useful as that it can restrict the process to its minimal requirements, thus further reducing the attack surface against the host and other containers [8, p. 4].

add cita tion

### 2.2.4. Docker

The concept of container-based virtualization has existed long before Docker came into view in 2013 [11]. Yet, it was Docker, Inc. who made containers popular by creating a toolkit that is greater than the sum of its parts [12, p. 1]. The following presents three areas of developer concern in which Docker shines and through which it has become a synonym for containers [12]:

add cita tion

add cita tion

#### Code packaging

A study found that less than 50% of software can be successfully built or installed. This is due to issues such as the "dependency hell" problem (see subsection 2.1.1), imprecise documentation or code rot. Executing code assumes the ability to create a compatible environment [3, p. 72]. Docker addresses this challenge with the concept of container images. Such an image bundles an application with all of its dependencies up to, but excluding, the kernel

add to glossary

[10] Security-Enhanced Linux (SELinux) was originally developed by the NSA to address threats of tampering and enable the confinement of damage caused by malicious applications.

[11] In 1998, FreeBSD came up with an extended version of `chroot()`, called jails. This capability further improved with the release of Solaris' zones in 2004 [2, p. 82]. In Linux, kernel namespaces were discussed as early as 2006 [4, p. 1].

[12] Boettiger makes a good case how Docker can also help to make research reproducible and more easily extendable [3, p. 71].

[12, p. 1]. It is essentially the filesystem bundle made available to a process that is then isolated as a container (comp. Chroots on page 9). To further simply things, Docker allows users to imperatively describe how such an image shall be built. This plain text file of steps to be taken is known as a Dockerfile and is ideally suited for use with a VCS, i.e. it can be checked in along a repository and evolve with the application [3, p. 74]. As a final innovation in this area, Docker calculates the filesystem differences between each of the steps in a Dockerfile and treats each as a distinct layer of the image. This enables developers to both version and extend images [13] [12, p. 1].

**Code portability**

By packaging an application along with its dependencies into a single image, Docker paves the way for image-based deployments that offer the freedom of "develop once, deploy everywhere" [14] [10, p. 203]. In this context, a container can be regarded as a running instance of an image. However, it is unlikely that such a container by itself will run in the same way, if at all, across platforms due to differences in e.g. networking or storage [3, pp. 74–75]. This problem is known as runtime consistency [10, p. 203]. To enable consistent behavior, Docker ships with a container runtime that abstracts many of the platform peculiarities [15] [3, p. 75].

**Code reuse**

As previously stated, Docker allows one image to extend another. To make this process more straightforward, Docker offers a public registry, the Docker Hub, to and from which users can up- and download versioned, binary copies of images. Eder notes that having such a social aspect in the area of virtualization was unheard of before. At the same time, he warns that because images do not get updated automatically, there is a large amount of images containing security vulnerabilities. For instance, over 30% of official images, i.e. those created by official project developers, contain high priority vulnerabilities [8, pp. 6–7].

Driven by the risk of lack of interoperability, Docker and other leaders in the container industry established the Open Container Initiative (OCI) in 2015 to standardize aforementioned container image formats and runtimes [17, p. 23]. update citation

[13] At runtime, all image layers will be merged into a single representation of the filesystem. This is known as a union mount [15, p. 26].

[14] While image-based deployments can also be achieved with VMs, they are not as portable nor lightweight because they contain the complete toolchain for running an OS, including device drivers, the kernel and init system [10, p. 203] [8, p. 2].

[15] Of course, target platforms will have to provide some sort of process isolation features (see subsection 2.2.3). Because this is not the case on macOS and Windows, Docker runs inside a Linux-based VM on those platforms [12, p. 5].

## 2.3. Container Orchestration

## 2.4. Agile Software Development

### 2.4.1. CI/CD

### 2.4.2. DevOps

### 2.4.3. GitOps

# 3. Concept

## 3.1. Overview

## 3.2. Functional Requirements

## 3.3. Non-Functional Requirements

# 4. Architecture

## 4.1. Component Design

### 4.1.1. Hardware

### 4.1.2. Backend

### 4.1.3. Maintenance and Support Plan

## 4.2. Component Interaction

## 4.3. Component Mapping

## 4.4. Design Rationale

### 4.4.1. Modeling as Cloud-Edge Computing Problem

# 5. Implementation

# 6. Testing

# 7. Conclusion

# 8. Summary

# 9. Outlook

# Bibliography


[1] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. “Microservices architecture enables devops: Migration to a cloud-native architecture”. In: *Ieee Software* 33.3 (2016), pp. 42–52.

[2] David Bernstein. “Containers and cloud: From lxc to docker to kubernetes”. In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84.

[3] Carl Boettiger. “An introduction to Docker for reproducible research”. In: *ACM SIGOPS Operating Systems Review* 49.1 (2015), pp. 71–79.

[4] Antonio Celesti et al. “Exploring container virtualization in IoT clouds”. In: *2016 IEEE international conference on Smart Computing (SMARTCOMP)*. IEEE. 2016, pp. 1–6.

[5] Tomas Cerny, Michael J Donahoo, and Michal Trnka. “Contextual understanding of microservice architecture: current and future directions”. In: *ACM SIGAPP Applied Computing Review* 17.4 (2018), pp. 29–45.

[6] Namiot Dmitry and Sneps-Sneppe Manfred. “On micro-services architecture”. In: *International Journal of Open Information Technologies* 2.9 (2014).

[7] Nicola Dragoni et al. “Microservices: yesterday, today, and tomorrow”. In: *Present and ulterior software engineering* (2017), pp. 195–216.

[8] Michael Eder. “Hypervisor-vs. container-based virtualization”. In: *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)* 1 (2016).

[9] Miika Kalske, Niko Mäkitalo, and Tommi Mikkonen. “Challenges when moving from monolith to microservice architecture”. In: *International Conference on Web Engineering*. Springer. 2017, pp. 32–47.

[10] Hui Kang, Michael Le, and Shu Tao. “Container and microservice driven design for cloud infrastructure devops”. In: *2016 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2016, pp. 202–211.

[11] Asif Khan. “Key characteristics of a container orchestration platform to enable a modern application”. In: *IEEE cloud Computing* 4.5 (2017), pp. 42–48.

[12] Dirk Merkel et al. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux journal* 2014.239 (2014), p. 2.

[13] Antonio Messina et al. “A simplified database pattern for the microservice architecture”. In: *The Eighth International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA)*. 2016, pp. 35–40.

[14] Roberto Morabito, Jimmy Kjällman, and Miika Komu. "Hypervisors vs. lightweight virtualization: a performance comparison". In: *2015 IEEE International Conference on Cloud Engineering*. IEEE. 2015, pp. 386–393.

[15] Claus Pahl. "Containerization and the paas cloud". In: *IEEE Cloud Computing* 2.3 (2015), pp. 24–31.

[16] Mathijs Jeroen Scheepers. "Virtualization and containerization of application infrastructure: A comparison". In: *21st twente student conference on IT*. Vol. 21. 2014.

[17] Vitor Goncalves da Silva, Marite Kirikova, and Gundars Alksnis. "Containers for Virtualization: An Overview." In: *Appl. Comput. Syst.* 23.1 (2018), pp. 21–27.

[18] Vindeep Singh and Sateesh K Peddoju. "Container-based microservice architecture for cloud applications". In: *2017 International Conference on Computing, Communication and Automation (ICCCA)*. IEEE. 2017, pp. 847–852.

[19] Davide Taibi and Valentina Lenarduzzi. "On the definition of microservice bad smells". In: *IEEE software* 35.3 (2018), pp. 56–62.

[20] Johannes Thönes. "Microservices". In: *IEEE software* 32.1 (2015), pp. 116–116.

[21] Leila Abdollahi Vayghan et al. "Deploying microservice based applications with kubernetes: Experiments and lessons learned". In: *2018 IEEE 11th international conference on cloud computing (CLOUD)*. IEEE. 2018, pp. 970–973.

[22] Leila Abdollahi Vayghan et al. "Microservice based architecture: Towards high-availability for stateful applications with Kubernetes". In: *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE. 2019, pp. 176–185.

[23] Mario Villamizar et al. "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud". In: *2015 10th Computing Colombian Conference (10CCC)*. IEEE. 2015, pp. 583–590.

[24] Miguel G Xavier et al. "Performance evaluation of container-based virtualization for high performance computing environments". In: *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE. 2013, pp. 233–240.

# Glossary

**cloud computing**

National Institute of Standards and Technology (NIST) internationally accepted definition of cloud computing calls for resource pooling where provider's computing resources are pooled to serve multiple consumers using multi-tenant model with different physical and virtual resources dynamically assigned and reassigned according to consumer demand [2, p. 81].

**scale cube**

Describes a three-dimensional model for scaling an application. X-axis scaling refers to running multiple instances of an application behind a load balancer. Y-axis scaling splits an application into multiple, distinct services. Z-axis scaling partitions the data to be processed across a set of application instances [13, p. 36].

# Appendices

# A. Concept

# B. Architecture

## B.1. Representational State Transfer (REST)

## B.2. Certificate-based Authentication

# C. Conclusion