Migrating the Monolithic Software Application GridVis to a Microservice Based Architecture

By

PATRICK K. WELTER

Department of Mathematics, Natural Science and Computer Science
UNIVERSITY OF APPLIED SCIENCES GIESSEN

A thesis submitted to the University of Applied Science Giessen in accordance with the requirements of the degree MASTER OF SCIENCE.

Referees: Prof. Dr. Franzen, Prof. Dr. Bienhaus

OCTOBER 2017

Abstract

When legacy applications grow in complexity and feature capabilities, the need for scaling and adjustment grows as well. Companies aim for prompt delivery and the potential of outsourcing development infrastructure is becoming and more cost-effective due to enhancing cloud technologies.
When it comes to high scalability and robust software in combination with an optimized delivery pipeline the buzzword *microservice* is hard to avoid. Famous Internet-Service-Companies like Amazon, Google, Facebook and Netflix produce evidence that a microservice-based architecture may meet most of the expectations.
Within the context of this thesis, I'll introduce an approach to migrate a large existing monolithic software architecture to a microservice-based architecture.
After introducing the motivating idea behind a migration and the related challenges, I will comment on recent technologies and concepts that can be used to solve frequent problems within a microservice landscape.
Within the main part of this dissertation, I will introduce a strategy for a migration approach with a corresponding functional extraction process based on a real-life example. The extracted components will be implemented as microservices within a well-fitting set of technologies.
Major difficulties, system-related concepts and the migration approach will be reflected critically, and advantages as well as drawbacks of the possible design decisions will be discussed.
The final passages will outline some possible future prospects regarding the further migration of the system.
This paper is dedicated to software architects and engineers who are driven towards distributed software design. The paper shall give an overview of practical challenges, corresponding problem solutions and possible design decisions while migrating to a microservice-based architecture.

Dedication and acknowledgements

My dedication goes to my family and friends, who always supported me unconditionally in prosperity and adversity. Further, I want to thank all my colleagues, who helped me with knowledge, kindness and humour during the course of my studies. I promise to share your spirit, if and when other people are in need.

Author's declaration

I declare that the work in this master thesis has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: .... DATE: ....

TABLE OF CONTENTS

List of Abbreviations

JVM	Java Virtual Machine
NPM	Node Package Manager
NBM	NetBeans Module
POM	Project Object Model
UI	User Interface
DDD	Domain Driven Design
API	Application Programming Interface
PaaS	Platform as a Service
JAR	Java Archive
SRP	Single Responsibility Principle
WADL	Web Application Description Language
RAML	RESTful API Modelling Language
REST	Representational State Transfer
JSON	JavaScript Object Notation
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
CI	Continuous Integration
SaaS	Software as a Service
SOA	Service-oriented Architecture
XP	Extreme Programming

CAP	Consistency, Availability, Partition Tolerance
DRY	Reporting-Suite Yourself
DNS	Domain Name System
IP	Internet Protocol
TTL	Time To Live
CD	Continuous Delivery
CPU	Central Processing Unit
IO	Input Output
RPC	Remote Procedure Call
RMI	Remote Method Invocation
XML	Extensible Markup Language
SOAP	Simple Object Access Protocol
ESB	Enterprise Service Bus
AMQP	Advanced Message Queuing Protocol
STOMP	Simple (or Streaming) Text Orientated Messaging Protocol
MQTT	Message Queue Telemetry Transport
BC	Bounded Context
MSSQL	Microsoft Structured Query Language
JanDB	Janitza Database
MySQL	My Structured Query Language
WAR	Web Application Archive
V	Voltage
FMC	Fundamental Modeling Concepts
CRUD	Create, Read, Update, Delete
JS	JavaScript

AWS	Amazon Web Services
ELK	Elasticsearch, Logstash and Kibana
URI	Uniform Resource Identifier
UUID	Universally Unique IDentifier
RDBMS	Relational Data Base Management System
HTML	Hyper Text Markup Language
MVC	Model-View-Controller

List of Figures

Chapter 1

Introduction

1.1 Prologue

Digitalization and Industry 4.0 drive the usage of cloud-based cost-efficient software. Customers aim for service-oriented use of software solutions. As a result, more and more software producers have begun to migrate their legacy systems and infrastructure to a more scalable fault tolerant architecture [12]. An architectural pattern opening such a perspective is called *microservice architecture*.

There is plenty of literature and web information focused on common microservice principles and design patterns. This literature frequently comes with short and straightforward examples to demonstrate a specific technology or concept. But companies want to ship complex and feature-rich enterprise software, which is a composition of these ideas. Developers must understand every single technology and individually apply each technology on their project by balancing pros and cons. Since divide and conquer is favourable, the compound set of design principles and technologies are not trivial. Connecting system parts or technologies may become sophisticated. A migration to a microservice system offers dozens of technology choices and engineers should have knowledge about distributed systems.

Many companies maintain and extend an already existing monolithic legacy software system. The maintenance of these systems can be increasingly expensive, since old technical debts still exist. Migration to a microservice-based architecture can increase software quality.

But an architectural evaluation progress throws up several questions like the following:

- *When is a migration of an existing monolithic software application to a microservice architecture reasonable?*
- *How could we start a migration?*

- *Which technologies are best suited for our project?*
- *What are the risks and challenges we will face during a migration?*
- *How do we split up a monolithic software application?*

This thesis contains an empirical investigation dealing with the above-mentioned questions by guiding the reader through a transparent real-life examination of a productive business software called *GridVis*. Since the scope of microservice architectures is way too large, I will mainly focus on implementation instead of deployment and maintenance. My focus basically lies on the extraction of existing functionalities and their implementation. Please consider that I do not claim that the presented techniques work best for any project.

1.2 Definitions

1.2.0.1 Measurement and Measurement Reading

The term **measurement** is defined in relation to the *International System of Units*. This concludes all base and derived units that can be recorded by energy monitoring units (i. e. devices) mentioned in this empirical study.
The term **measurement reading** is defined as a single measurement value in time recorded by energy monitoring units (i. e. devices) mentioned in this empirical study.

1.2.0.2 Device

Physical measurement units capable of collecting certain measurement readings are called **devices**. The company *Janitza electronics GmbH* develops and produces a huge variety of different devices capable of performing a wide range of calculations in order to collect different measurements.

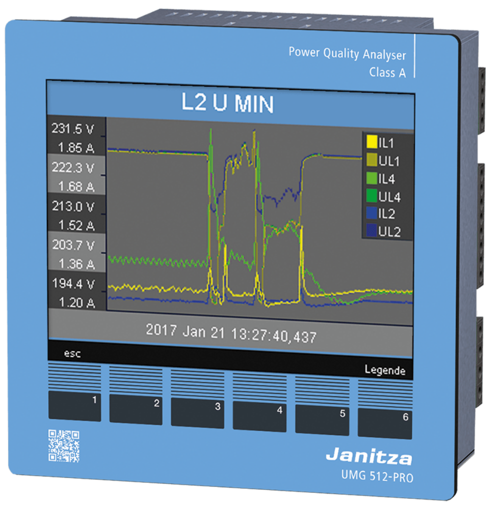


Figure 1.1: The device *UMG 512-PRO (Class A)* by Janitza electronics GmbH. The device was designed in order to collect data about residual current.

1.2.0.3 Online Value and Online Value Type

An **online value** is the last recent measurement reading of a certain device. To avoid misunderstandings, online values are not collected in real-time. Each online value references a specific online value type, consisting of exactly one device reference and one measurement.

1.3 Project Scope: GridVis

1.3.1 About GridVis

The software being considered for migration is called *GridVis*. GridVis is a network visualization software system with a primary goal to provide transparency to energy and power quality measurement (e. g. for detecting voltage drops or flickers). This software comes with a wide variety of features, such as dashboards, reports, key performance indicators, alarms and more. The application is mainly written in Java and has a proper subset of JavaScript modules. In terms of size, GridVis comprises about 1.1 million lines of code.
GridVis was built on top of the *NetBeans platform* and mostly uses the NetBeans Module (NBM) classloader, Maven Project Object Model (POM) and Node Package Manager (NPM) modules to structure code and dependencies.
The GridVis system consists of two main applications sharing a common module set. Both applications are platform independent due to the abstraction of the Java Virtual Machine (JVM). The first application is called *GridVis Desktop*. This is a Rich-Client desktop application built upon

the NetBeans platform. It comes with a UI composed of Swing, JavaFX and the web technologies Hyper Text Markup Language (HTML) and JavaScript (JS). The application is optimized for the configuration of devices and several GridVis features.
The second application is called *GridVis Service*. It runs as an operating system service process. The GridVis Service is conceived as a background program handling GridVis functionalities required at full time. Such tasks might be the collection and persistence of historical measurement readings or the rise of alarms as a warning for suspicious measurement readings.

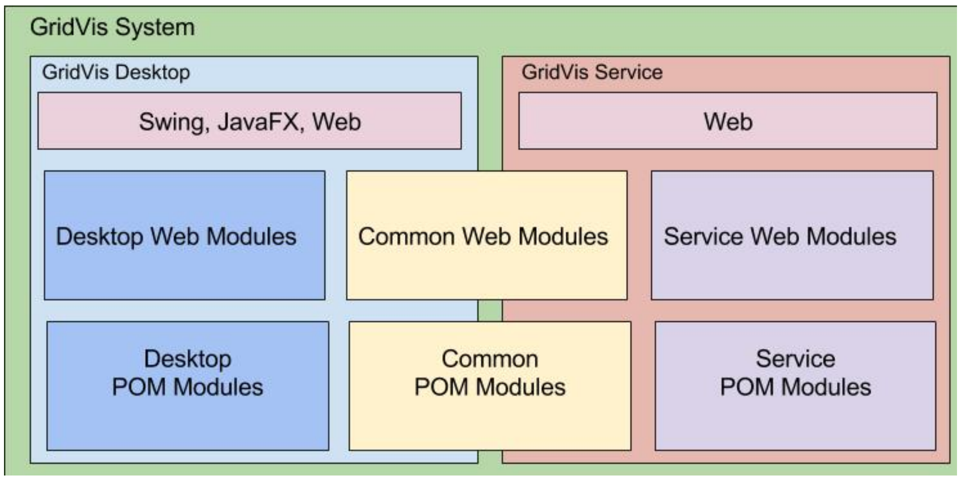


Figure 1.2: Overview of the GridVis system partitioning.

1.3.2 GridVis Operations

In order to collect measurement readings from the energy grid, GridVis communicates with a set of power monitoring units. These units are physical *devices* installed within the customer's energy consumer topology. Each device records a configured set of measurement readings. Depending on the device type, recorded measurement readings are saved to a device's internal storage, where GridVis can access them via bus (Ethernet, Modbus etc.).

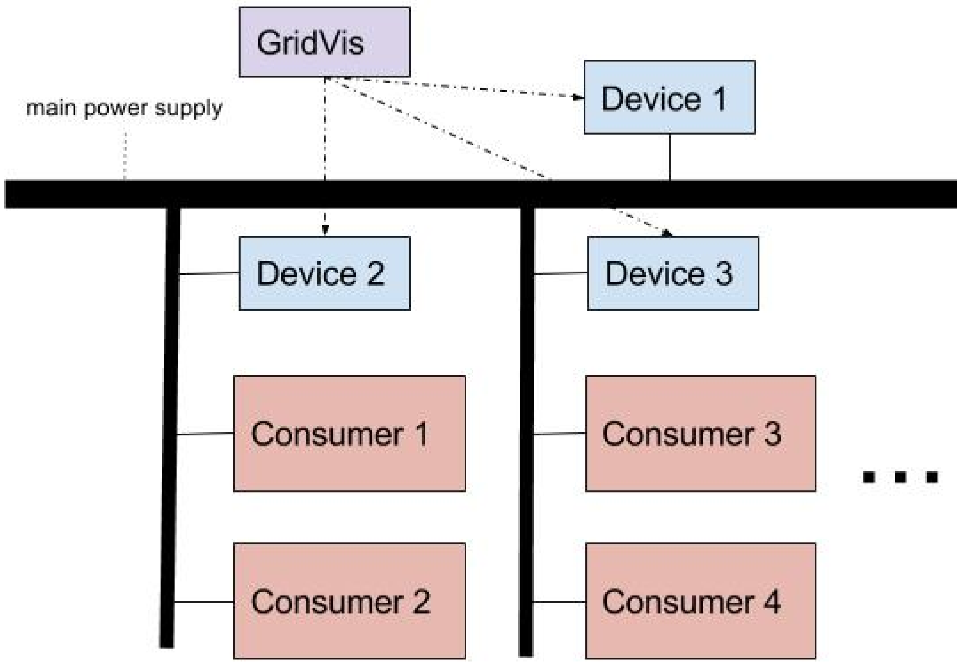


Figure 1.3: Example of a customer's energy consumer topology with connected devices.

GridVis is shipped to the customer as an executable application. The programs (GridVis Service and GridVis Desktop) are installed on the customer's system.

1.4 Motivation for Migrating GridVis

As expected, a well-designed microservice architecture comes with reasonable benefits and drawbacks. This chapter is designated to characterize the project-specific demands of the GridVis system while the common architectural properties will be discussed in chapter 2.1.1.

1.4.1 Principles

Migrating a system brings plenty of trade-offs to choose from. Some decisions can be made relatively easily, because the necessary information is available, whereas other decisions are quite hard to make, since information is incomplete or difficult to obtain. Therefore, a predefined framework consisting of standard principles and practices eases decision-making a lot.
Strategic goals speak to where the company will be moving in the long term. For example, a strategic goal can be to decrease the time to market for new features.
Architectural Principles are rules helping to align practices to a higher goal. For example, the delivery teams have full control over their software life cycle in order to release features faster.
Practices, a set of detailed, practical guidance for performing tasks, help us to ensure that principles are carried out. Due to their technical nature, practices change much more often than

principles do. A practice could be to implement default inter service communication via Hypertext Transfer Protocol (HTTP) and REST [29, pp. 17 ff.].

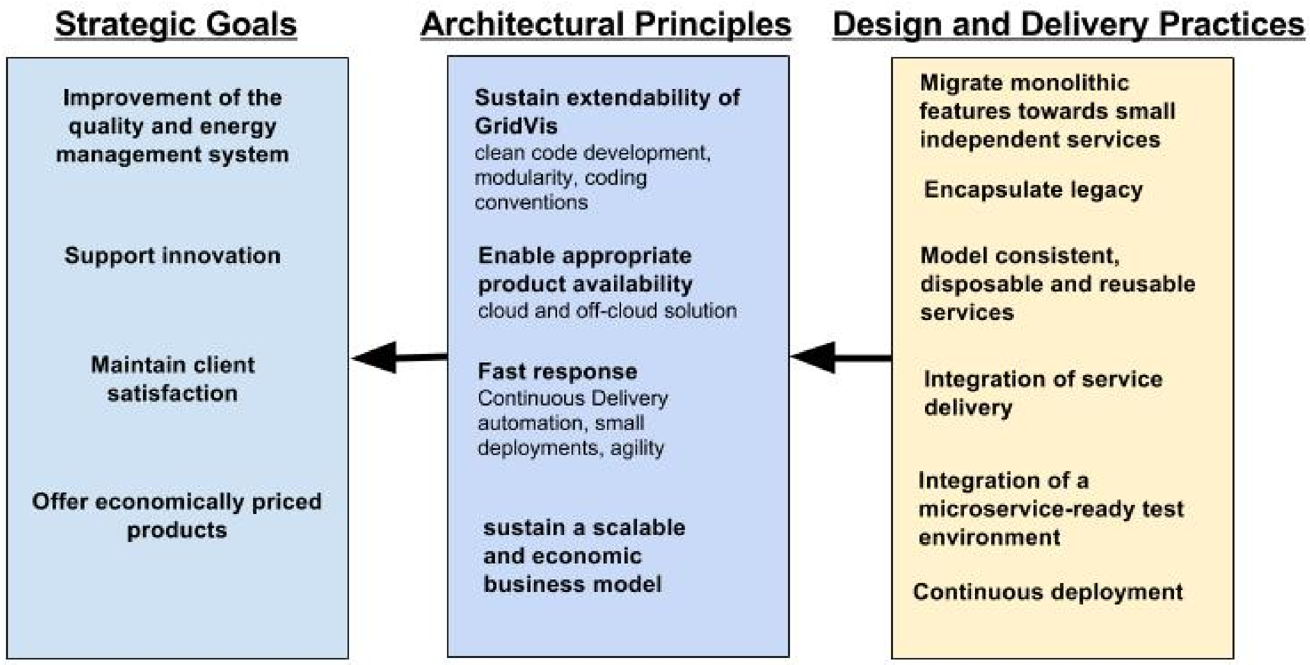

Strategic Goals
Improvement of the quality and energy management system
Support innovation
Maintain client satisfaction
Offer economically priced products
Architectural Principles
Sustain extendability of GridVis
clean code development, modularity, coding conventions
Enable appropriate product availability
cloud and off-cloud solution
Fast response
Continuous Delivery automation, small deployments, agility
sustain a scalable and economic business model
Design and Delivery Practices
Migrate monolithic features towards small independent services
Encapsulate legacy
Model consistent, disposable and reusable services
Integration of service delivery
Integration of a microservice-ready test environment
Continuous deployment

Figure 1.4: Broadly defined strategic goals, architectural principles, design and delivery practices of Janitza electronics GmbH and the GridVis project

Fig. 1.4 outlines how the microservice architecture can help Janitza electronics GmbH to achieve the company's strategic goals: The architectural principles reflect the system's properties and behavior, while the design principles and practices outline what developers need to perform in order to accomplish these architectural principles.
Microservices can help to sustain the maintainability and expandability of GridVis. They offer a good opportunity to create a scalable economic business model.
While migrating, it is reasonable to encapsulate the legacy code of GridVis in order to achieve better control of change impacts. Well-designed microservices are logically consistent, disposable and reusable. They help a company react quickly to customers' demands, since independent teams can deploy them as they want without waiting periods. In addition, an outdated microservice can be replaced more easily.
A microservice-based system needs an appropriate test environment within its Continuous Integration (CI) pipeline. Otherwise, mistakes may be detected too late, which slows down customer response time.
The customer requires different types of access to GridVis, because he might maintain an energy management system without internet access. This requirement implies that microservices need to be integrated in some sort of shipment solution.

As you may have noticed, all these practices rest upon principles ensuring some of the strategic goals of the company.
Within the next chapters, I will go into more details showing some advantages and disadvantages of a GridVis system migration.

1.4.2 Segregation of Domain Features

The principle of segregation within the GridVis system itself is applied in terms of POMs. Broadly speaking, as each Maven module consists of multiple nested sub modules, most of the GridVis internal module compositions shape a single feature with a corresponding API: modules for device communications, reporting, graphs etc. This encapsulation encourages a migration to a microservice system due to an easier utilization of Domain Driven Design (DDD). The principles and adoption of DDD will be explained in chapter 2.4 while the usage of these principles will be outlined in chapter 3.2.4.

1.4.3 Economic Efficiency for Customers and Suppliers

Cloud business models can be natively applied on microservice-based systems due to high availability and optimized utilization of resources compared to other known web service architectures. Such business models result, for example, in *pay on demand* licenses [31].
There is no need for the customer to install and maintain the software and its related infrastructure.
Running and maintaining a system can become labour-intensive. For example, a GridVis user needs to set up a database. Depending on the customer's project size, measurement readings over a long period of time may require large disc space capacities. This results in an intervention, like manually adding disc space or data compression in order to prevent the system from failing. Another task might be picking the appropriate starting parameters like the JVM heap size of the GridVis Service application.
These examples are just a proper subset of operations that must be performed to run such an extensive system. In terms of technical depth, most of the required tasks can only be executed by well-educated engineers, which influences such a system's economic efficiency.
A cloud solution, on the other hand, encapsulates complex infrastructure dependencies. The customer does not need to care about disc space or environment and runtime parameters. Instead of installing and maintaining every piece of infrastructure by himself, the customer books a certain amount of support in terms of Software as a Service (SaaS) and the system allocates all necessary resources. Scaling services in a cloud environment helps the software producer to react appropriately to customers' demands. For example, additional microservices can be dynamically started to handle demand peaks on a specific system feature. As soon as a functionality demand decreases, microservices not required any more can be shut down in order to save resources.
Like R. RV points out in [33, p. 2], microservices are supposed to increase business efficiency in

terms of agility and delivery speed while at the same time reducing costs and project turn around time.

1.4.4 Resolving Inconvenient Dependencies

As software grows and complexity increases, unforeseen dependency conjunctions can emerge during further development. Cross-dependencies were linked up or module extractions are absent due to pending refactoring or design decisions. Such legacies can sneak into the software context and decrease code quality in terms of maintenance and expandability.
A migration to a microservice architecture can dramatically lower the coupling of domain specific software components, as we will see later on in this document. In the common view, microservices are designed as small units. Each unit encapsulates a domain-specific feature on its own [29, pp. 2 f.].
As an example, a web shop application could consist of a registration service, a news letter service, checkout service, etc.
This design approach removes the need for very complex module dependency management, as is required with large monolithic software architectures. This benefit is mainly owed to messaging, as described in chapter 2.1.2.1. However, it is important to avoid the idea that the microservice approach will solve all problems we are facing in dependency management on monoliths, since the complexity simply shifts from dependency management into Application Programming Interface (API) design and API versioning.

1.5 Challenges for Migrating GridVis

1.5.1 Legacy Base

The re-engineering of legacy code is a chance, but is also work-intensive due to advanced evaluation processes. Each domain functionality and feature within GridVis needs to be assessed on its own regarding code decomposition, data modelling and communication behavior.
Integration tests in particular must be rewritten based on the new system architecture. New bugs related to communication infrastructure and data consistency will occur. New data persistence technologies need to be understood and integrated correctly. Developers will need to rethink their algorithms and applied design patterns. Moreover, new test infrastructure layers have to be added to the system in order to ensure messaging and network fault behavior.
For larger projects, this set of problems can not be faced in a large single refactoring process, but only with an iterative approach.

1.5.2 Customer Application Access

Not all GridVis customers wish to change their energy management infrastructure towards an external cloud solution. Some customers intend to integrate a cloud-native GridVis system into their own cloud environment.
Other customers aim for a non-cloud solution, which involves running the application on their native host systems. This implies a need for a comprehensive installer, containing all necessary microservices and the architecture-specific infrastructure.
Altogether, product delivery is one of the most critical requirements during the migration evaluation. Many aspects should be considered, including runtime performance on a given data throughput in relation to computation resources, the size of the single artefacts (Java Archives (JARs), microservice infrastructure instances, etc.) and the corresponding installation effort.

Chapter 2

Technologies and Platform

2.1 Microservice Architectures

2.1.1 About Microservices

The term microservice architecture was discussed quite often in many academic sources. This chapter summarizes the main idea behind microservices.

Microservice architecture is an approach to structure and modularize software. The evolution of microservices has mainly been driven by the disruptive digital innovation trends in modern business and technology [33, p. 1]. Many companies, like Amazon, Netflix and eBay divided their monolithic systems to functional partitions each performing a single task. These companies were driven by prevailing issues with their monolithic applications. Many organizations followed this approach and finally, evangelists termed this pattern microservice architecture [33, pp. 5f].
The main idea behind microservices is structuring software into multiple applications that run as simultaneous processes forming a distributed system, instead of structuring software modules into a single application, where each application accomplishes a domain specific purpose, also known as Single Responsibility Principle (SRP). This approach is basically derived from the UNIX philosophy by Doug Mcllroy :

> Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features" [32].

While splitting a large scale software application, the organization of development teams is affected as well [39]. When it comes to finding the *right* size, service boundaries are oriented against business boundaries. One microservice should reflect one piece of the domain function-

ality. This approach grants the transparency to determine where the code for a given domain functionality can be found. The limitation of each service to a business functionality avoids each service growing too large with all the resulting consequences [29, p. 2].
Each microservice is a separate entity which might be individually deployed as an isolated service, e. g., as a Platform as a Service (PaaS) system or on a native operating system [29, p. 3].
The relationship between microservices should be as independent as possible. Within its chapter *strategic design*, the DDD enlists possible relation classes based on communication behavior, which can be reflected on microservice relations.
The reader might wonder if the microservices approach is not the same as Service-oriented Architecture (SOA). Well, this is basically a terminology decision, with companies interpreting and implementing SOA in different ways. In [29, p. 8], Newman points out that the microservice architecture is a specific approach for SOA, pretty much in the same way as Extreme Programming (XP) and Scrum are specific approaches for Agile Software Development. A more detailed distinction can be found in [33, p. 33] by R. RV.

2.1.2 Discussion: Benefits and Challenges

2.1.2.1 Collaboration behavior

Since a microservice-based system consists of multiple independent services, communication is unavoidable. But how should services collaborate? Shall they use synchronous or asynchronous communication?
Synchronous communication implies a call to a remote server, which is blocked until the remote server has answered.
Asynchronous communication encounters no blocks, since the caller does not wait for the operation to complete before returning.
Basically, synchronous communication can be easier to deal with. For example, it is obvious, if a call has succeeded or not and source code tends to be more readable. On the other hand, long-term jobs, applied on the remote side, have to be held within an open connection, and this causes performance loss.
If low latencies are needed, asynchronous communication might be better. In addition, the UI even remains responsive if the network is laggy due to optimistic rendering [29, p. 42].
Each communication mode aligns with its own idiomatic collaboration style. Synchronous communication mostly aligns with *request/response*, where the client initiates a request and waits for the response, but it can also be accomplished in an asynchronous way, where the client kicks off an operation and registers a callback.
The second collaboration style is *event-based* communication. In this case, a system component communicates *what has happened* on its side and expects other components to react correspondingly. This approach is highly decoupled, since the client that emits an event does not need to know who or what will react to it. New event consumers can be added without applying a single

change to the producer [29, pp. 42 f.].

2.1.2.2 Technology Heterogeneity

Each microservice can be implemented in its own separated and isolated way to fulfil its demands. Like Netflix presented in their technology blog, developer teams can pick the right tool for each job. Each technology may be replaced by a more adequate alternative while providing a certain amount of infrastructure. [24]
As an example, in a social network application, user interactions could be stored in a graph-oriented database to reflect the highly interconnected nature of a social graph. Posts from a certain user could be stored in a document-oriented data store whereas blob stores would be more suitable for saving profile pictures.

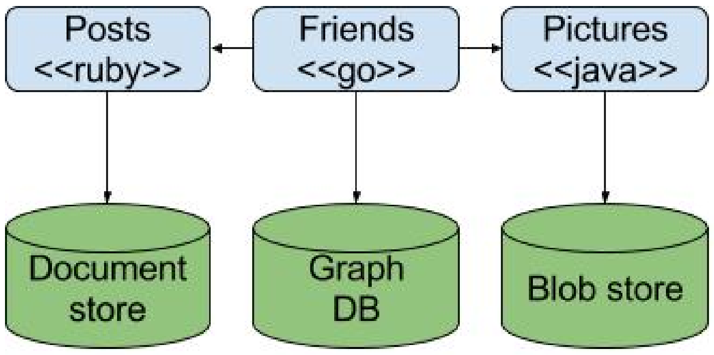


Figure 2.1: Microservices (blue) implemented in different languages using different data storage technologies (green) [29, p. 4].

When trying out new technologies, like introducing a new programming language, the biggest barrier is the risks associated with it. This is more or less because new technologies like programming languages, database systems or frameworks have a large impact on the system. Within a microservice-based architecture, these risks are limited to a single service, which can be replaced more easily. This approach helps companies to take advantage of new technologies while limiting risks [29, p. 4].

2.1.2.3 Service Contracts

Microservices should fulfil a well-designed service-contract. JavaScript Object Notation (JSON) and REST are universally accepted for service communication. There are many techniques that can be used to design JSON/REST contracts, including Schema, Web Application Description Language (WADL), Swagger and RESTful API Modelling Language (RAML) [33, p. 11].

2.1.2.4 Loose Coupling

Most microservices accept events as input while sending events as output. In order to gain higher levels of technical independence, messaging, REST and HTTP are commonly used for interactions between microservices. But be ware of using these standard protocols without enhancing technologies. On one hand, interoperability is added to each service, which raises decoupling, but on the other hand, there may be a lack of feature support such as message recovery, etc. [33, p. 11].

2.1.2.5 Service Reuse

Each microservice is a coarse-grained reusable unit within the system accessed by mobile devices, desktop channels, other microservices or other systems. This kind of segmentation enables composability through service orchestration or service choreography [33, p. 11]. A small sized service application can be reimplemented fast and cleanly, because its scope should only include a slight set of functionalities [29, p. 8].

2.1.2.6 Statelessness

Well-designed microservices are built by using a share-no-state approach. If required within a microservice, state maintenance is normally stored in a database or in-memory [33, p. 12].

2.1.2.7 Discoverable

Microservices make use of location transparency to be quickly deployable, dynamic and automated. In order to enable information exchange through communication, a microservice self-advertises its existence and makes itself available for discovery. As soon as services die, they automatically take themselves out from the microservice ecosystem [33, p. 12].

2.1.2.8 Resilience

As earlier mentioned in chapter 1.3.1, GridVis consists of multiple functional components running on a different level of importance. As an example, the impact of an undelivered alarm notification could be more crucial than the disability of report creation.
Within the current GridVis version, a severe error occurring within the reporting module will possibly affect the entire system. Microservices can mitigate this problem in terms of a trade-off: It is possible to isolate errors and prevent them from cascading towards the system at the price of new sources of failure like network issues or host failures.

2.1.2.9 Scaling

With the increasing requirements of existing monolithic architecture, scaling becomes difficult. For example, with rising data throughput, it will become necessary to start several instances of the monolithic application in parallel.
However, a monolithic application also has to scale between functionality and code complexity, which is often achieved by modularization. But within monolithic applications, parallelization cannot be achieved at module level. Usually, a monolithic application has to be entirely replicated and unused parts will be replicated without being used. In an increasing number of instances, the drawback of unnecessary replication gets more importance, which finally reflects on the system's resource usage.
Another problem is that larger web-scale applications must be able to cope with an increasing amount of persistent data.
The scaling cube (Fig. 2.2) is a widely used illustration of various approaches for partitioning a monolithic application into microservices.

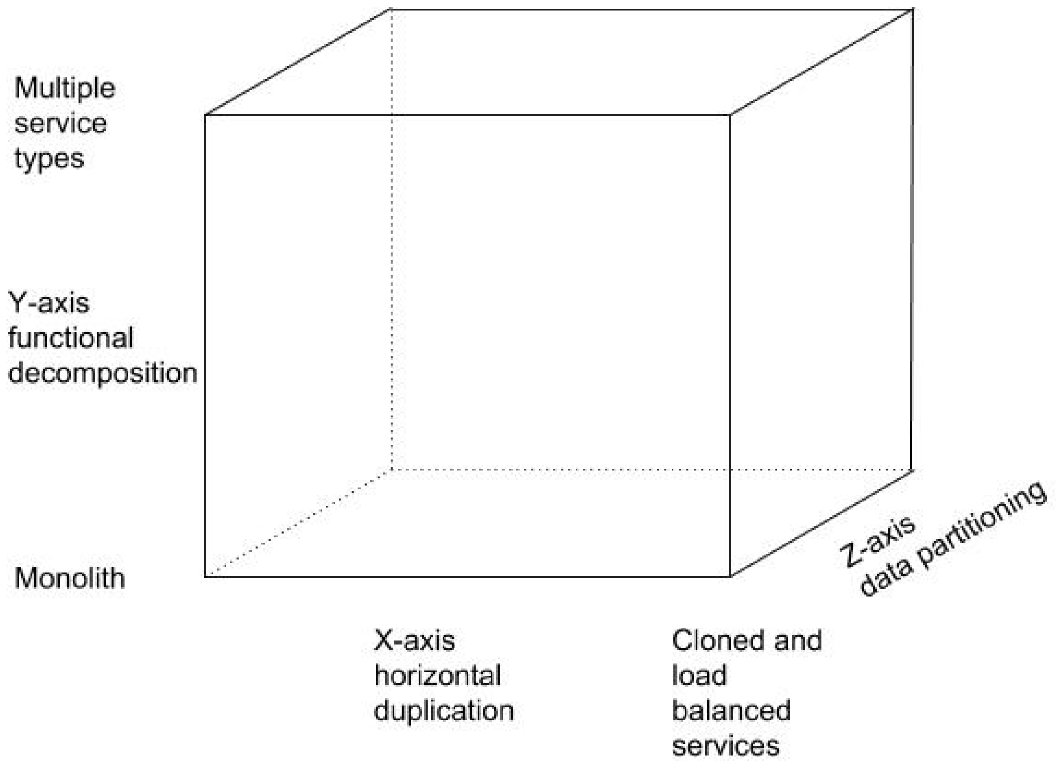

Multiple service types
Y-axis functional decomposition
Monolith
X-axis horizontal duplication
Cloned and load balanced services
Z-axis data partitioning

Figure 2.2: The scaling cube referring to microservice architectures [26].

X-axis scaling, also known as horizontal scaling, describes scaling by running multiple identical copies of a single application. This approach requires a load balancer handling requests and forwarding each of them to a specific service instance.
Within z-axis scaling, each server runs an identical copy of code, but each service is only re-

sponsible for a part of the entire data set. The Cassandra database, for instance, supports data partitioning within its data modelling concept.
While x-axis and z-axis scaling improves system capacity and availability, it adds development complexity as well. In order to deal with increasing application complexity, y-axis scaling can be applied. This is comprised of functional decomposition and splits a monolithic application towards more fine-grained services [26].

Real world example: The online fashion retailer Gilt adopted microservices to combat issues with scale and throughput.
In 2007, Gilt built their system as a monolith in Rails. In 2009, this system failed by not being able to cope with huge loads of data. This incident was caused by the system scaling factor. In order to increase the capacities of a single functionality within a monolithic system, everything needed to be scaled simultaneously. On the other hand, implementing smaller (single responsibility oriented) services helped to limit scaling only to required functionalities, allowing Gilt to run other parts of the system on less powerful hardware. This approach enabled better cost-effectiveness. By splitting up their system, Gilt became able to deal with its traffic spikes. In 2015, the system ran about 450 microservices [29, p. 6].

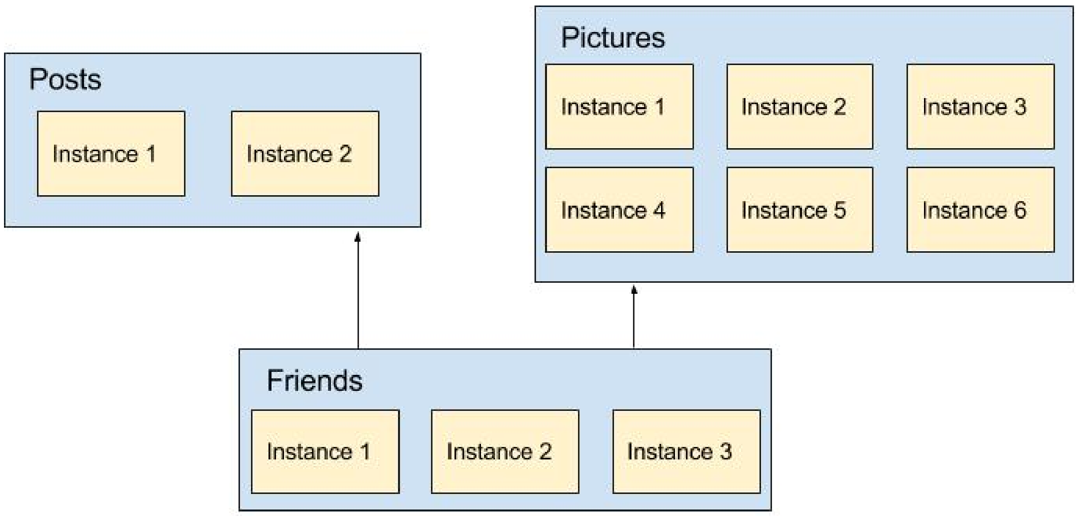


Figure 2.3: Horizontal and vertical scaling per functionality (blue) with redundant microservice instances (yellow) [29, p. 6].

2.1.2.10 Deployment

Each deployment of a large monolithic application like GridVis normally includes a lot of changes and happens infrequently. This is due in part to the high risk of deploying such a system. For instance, a data model change could affect multiple modules. Each module may contain some new unforeseen behaviors and errors. As a result, changes build up between releases. The bigger the delta between releases, the higher the risk that something will go wrong.

Microservices, on the other hand, can be deployed independently. Since they encapsulate a specific functionality, new feature requirements are usually within the scope of only a few single services. This allows developers to deploy faster. Problems can be isolated quickly to an individual service, providing the possibility to apply a rollback in time [29, p. 6].
When deploying microservices, CI is indispensable. The migration from a monolithic application requires adapting the build environment appropriately. It is advantageous to manage each microservice together with its tests and have it assigned to its own repository. As outlined in Fig. 2.4, each push should trigger a corresponding build job only responsible for this single microservice. It is important to adjust the build pipeline to gain full performance for the microservice architecture [29, p. 106].

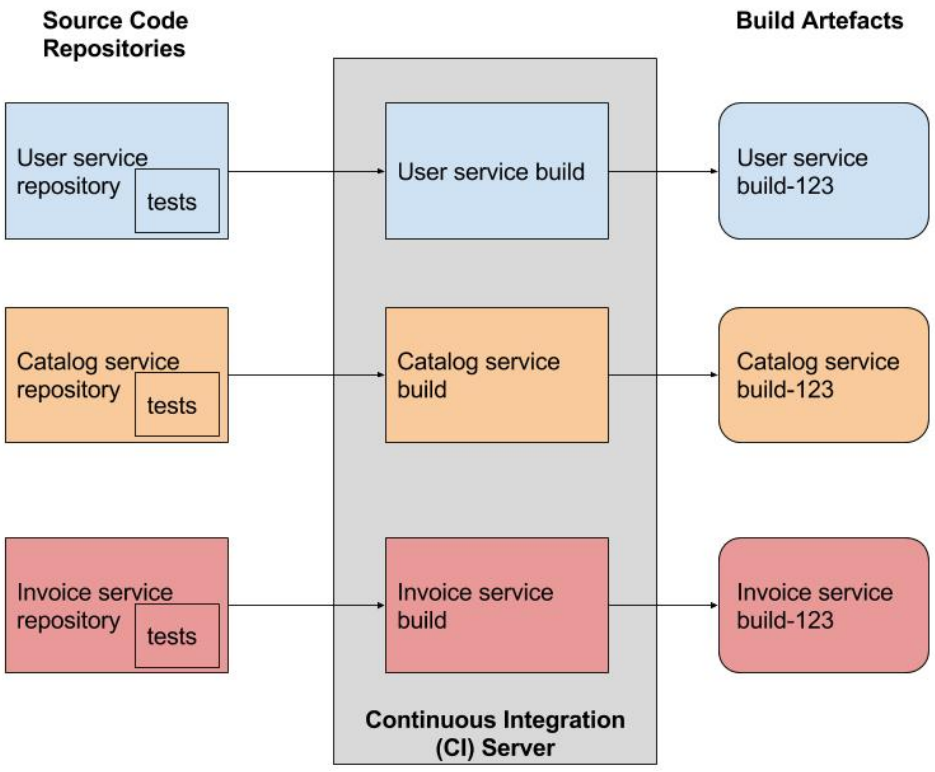


Figure 2.4: Model of multiple build jobs (coloured blue, orange and red) for individual microservices. Each microservice contains its tests within its source code repository. A code check-in triggers an individual build job at the CI server. Each build produces a single artefact for production [29, p. 107].

2.1.2.11 Testing

Due to finer grained services, the microservice architecture should improve deployment duration. But secure deployment can only be achieved in combination with a decent test coverage. The use of a well-composed test suite within the CI pipeline enables faster and more secure deployment,

because tests ensure system stability when source code changes are applied. There are multiple types of tests each targeting a specific problem set, see Fig. 2.5.
Unit tests are written to ensure the stability of small functional pieces within the software, such as functions or methods. Normally, developers are responsible to write them. Unit tests run very quickly, even in complex projects, while covering a huge part of the complete system. Frequently, tested functionalities require some external dependencies, e. g. a database call. These dependencies need to be replaced first, before a unit test can be applied.
A replacement can either be accomplished by using a *mock* or a *stub*. A mock simulates a function call with a determined result, whereas a stub simulates a microservice with limited capabilities. For example, the stub could answer over a specific communication channel with a constant value.
Integration tests are used to assure the correct interaction behavior between certain components and neither use stubs or mocks nor any system specific information. They can perform tests in relation to business logic, for example, performing operations like a customer does.
Unit and integration tests target deeper layered components within the system. UI tests, on the other hand, verify components via their UI. While performing UI actions, like clicking, dragging etc., they treat the application like a black box and validate the occurrence of expected behavior within the UI. A UI test could validate, for example, whether a displayed table on the UI was correctly extended by the item most recently entered.
Although testing the UI may result in large code coverage, UI tests are still fragile, since UI changes will break them, even if the logic behind stays untouched.
Last but not least, manual tests can be performed to ensure software quality. Manual tests are exploratively applied in order to find security, performance or domain specific deficits [39, pp. 221 f.] .

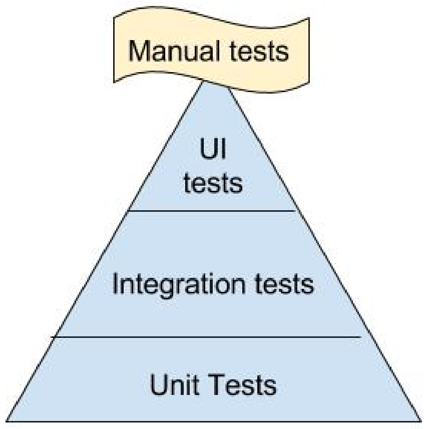


Figure 2.5: Optimal test pyramid: The test pyramid's base consists of unit tests, which are implemented most often. They assert that logic and algorithms work as expected. The second level consists of integration tests, though it is not recommended to implement as many integration tests as unit tests, because their implementation and execution take more time. UI tests break quite often and their implementation is complex. As a result, UI tests are the least frequent automated tests within the pyramid. And finally, manual tests are located at the top of the pyramid. They are quite expensive and should be applied as little as possible [39, p. 221].

Distributed systems have more error sources than monolithic systems. Certain system functionalities may require multiple calls between interacting microservices before returning a result. Integration with distributed system specific technologies, like load balancers, discovery, etc., has to be tested as well.

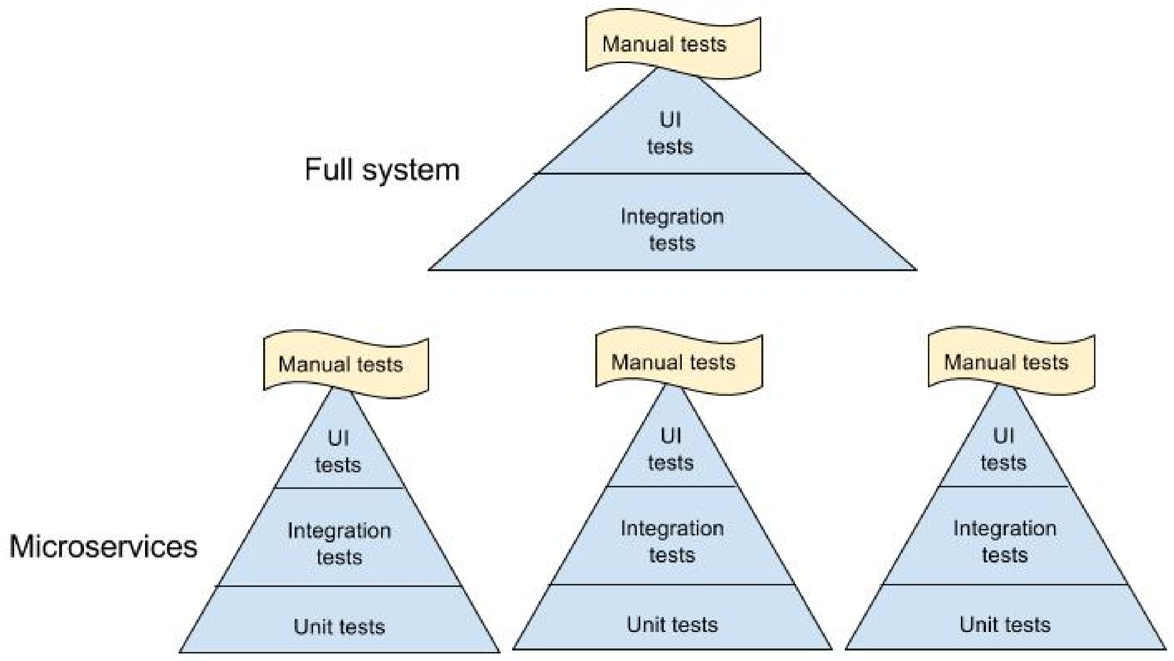


Figure 2.6: Full microservice system test coverage consisting of a test pyramid for each microservice and integration, UI and manual tests for full system stability [39, p. 226].

Alongside a test pyramid for each single microservice, the full system test coverage consists of integration, UI and manual tests, as displayed in Fig. 2.6. In order to ensure resilience, single microservices should be able to offer their functionalities even if depending microservices fail. Functional tests of a single microservice can be applied by using stubs or mocks for any dependency in order to avoid the execution of a complete distributed system.
Nevertheless, integration tests with other microservices are necessary before bringing a new version of a service into production. Each integration test may only be started in combination with a single update or new service instance. This means that system-wide integration tests have to be iteratively applied. The downside of this approach is the slowed-down release frequency caused by the duration of system integration tests. For instance, if a full system integration test takes 1 hour, 8 microservices can be deployed during a single working day. As a consequence, parallelizable tests within services should cover as much communication logic as possible [39, pp. 226 ff.].

2.1.2.12 Organizational Alignment

Microservice architecture does not only apply on software structure, but also on organizational structure. Melvin Conway observed in [15]:

> Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structures.

This statement is often quoted as Conway's law. Conway's law found in studies like [25], where *MacCormack et al.* looked at a number of different software systems and determined the relations between strongly and loosely coupled organizations and their produced software. In [10], Microsoft found that software-specific metrics like code complexity, code dependencies etc. associated with organizational structures proved to be the most statistically relevant measures.
Like Amazon and Netflix understood early on, smaller teams can work much more efficiently if they own the whole life cycle of the system they manage. Amazon recognized that the more teams are working on the same system, the slower development evolves, and Netflix learned from Amazon's approach. Netflix designed the organizational structure for the system it wanted: small, independently working teams [29, p. 192].
In the context of microservices, a single team should *own* a certain amount of services. In this scope, owning involves finding and using appropriate technologies, development, testing, bringing into production and maintenance, as shown in Fig. 2.7.
Normally, a team owning a microservice may change its code as it likes, but it still has to ensure it does not break consumer contracts. This implies an exclusion of a technical-oriented team structuring approach, like having a team for back-end, business logic and front-end development, as shown in Fig. 2.8 [29, p. 194].
In the same way that microservices need to communicate with other microservices, teams owning specific services have to communicate with other teams owning different services in order to find convenient APIs. If only a small number of features is involved, the development of new features is more efficient, since among teams, communicating is less frequently necessary.

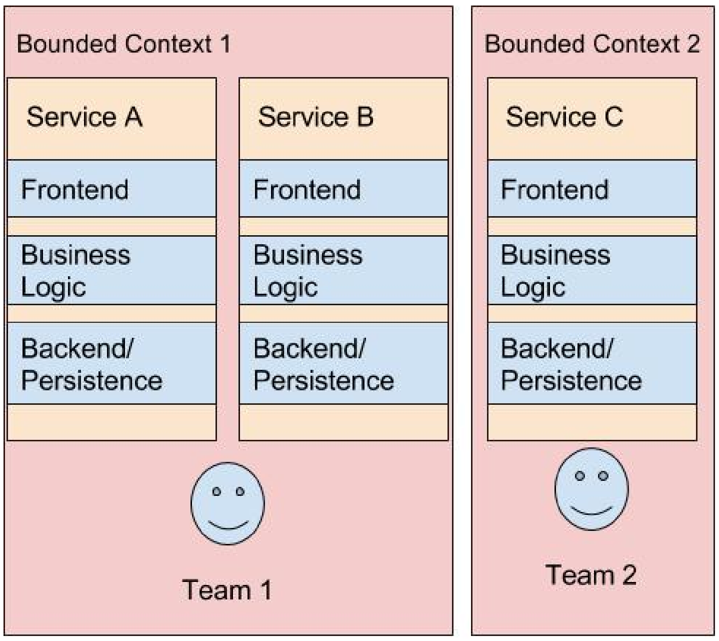


Figure 2.7: An example of a functionally partitioned team organization for developing microservices. One team works within its own bounded context. For example, bounded context 1 could include customer-specific functionalities while bounded context 2 could include billing specific functionalities. As billing and customer functionalities need communication, bounded context 1 and bounded context 2 need to communicate as well. This results in communication of team 1 and team 2. Since service A and service B are strongly coupled, and because both belong to bounded context 1, the team does not need to arrange for interfaces outside of its bounds.

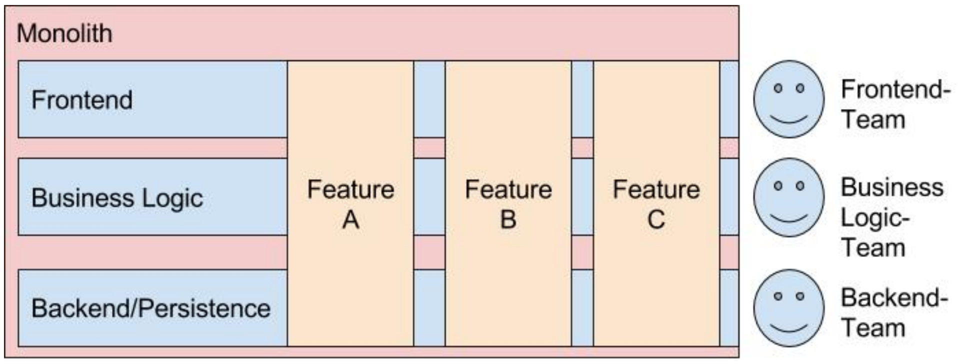


Figure 2.8: An example of a technically partitioned team organization for developing a large monolithic application. One team is assigned to a given form of technology: frontend, business logic and back-end. This approach ensures a high amount of technical knowledge within a single team [39, pp. 40 f.].

2.1.2.13 Code Reuse

Reporting-Suite Yourself (DRY) is a clean code principle and is a best practice in professional software development. The intention of DRY is to avoid duplicating system *behavior and knowledge*, which enhances source code reusability and code extendibility.
Unfortunately, when developing microservices, one thing developers should avoid at all costs is to couple a service and its consumers too closely, because then changing the producer could cause an unnecessary change to all its consumers. Sometimes, shared code can create this coupling [29, p. 59].
General advice found through several sources and experiences, [29, p. 59] and [39, p. 116], is to accept an adequate amount of code redundancy within multiple microservices, because the disadvantages of too much coupling are worse than the problems resulting from code duplication.

2.1.2.14 Consistency, Availability and Partition Tolerance Trade-off

A microservice system is a distributed system by its nature. The most common trade-off we will face while utilizing networked shared-data systems is described within Eric Brewer's Consistency, Availability, Partition Tolerance (CAP) theorem. The CAP theorem has been mathematically proved: It is not possible to achieve all three terms of *consistency, availability* and *partition tolerance* **at full extent** within networked shared-data systems.
An appropriate description of the terms can be found in [29, p. 232], where Sam Newman writes:

> Consistency is the system characteristic by which I will get the same answer if I go to multiple nodes. Availability means that every request receives a response. Partition tolerance is the system's ability to handle the fact that communication between its parts is sometimes impossible.

The problem addressed by the CAP theorem can be understood best by thinking of two partitioned nodes. If one node is updated, its new state will cause the nodes to become inconsistent. If consistency should be obtained, one side of the partition must act as if it were unavailable, which will sacrifice availability. If nodes communicate, they are able to achieve consistency and availability, but the network can fail and partition tolerance is violated [13].
A common misunderstanding is the idea of having a full exclusion of one characteristic completely. In [13], Brewer reconsiders his theorem and outlines that system designers can optimize all three characteristics by applying trade-offs. In addition, he assumes that cautiously dealing with partition faults in consideration of service invariants may shift the system towards availability or consistency. The shift towards availability or consistency can be applied without a complete exclusion of one of them. In relation to past strategies, the downside of this approach is the requirement to deploy more thoughtfully. The best solution will heavily depend on details about the service's operations [13].

2.2 Microservice Technologies

Mostly, all benefits and challenges of distributed systems apply to a microservice system. There are several standard problems companies will face before bringing a microservice system into production. Some of these challenges can be neglected consciously by agreeing to the given trade-off's, e. g., setting up a minimum amount of monitoring.
Many studies have been performed and a wide variety of companies have presented their solutions in order to face these problems. The range of solutions includes development, deployment and maintenance. Within this chapter, I'll present a set of well-proven technologies solving a specific problem set within the microservice environment. Instead of diving too deep into a single technology, I will try to highlight the most beneficial ideas and the drawbacks behind each of them.

2.2.1 Service Discovery

2.2.1.1 About Service Discoveries

As mentioned in chapter 2.1.2.7, microservices make use of location transparency. Since services can be started on different machines and within different networks, they need to know where other services live in order to be able to establish a communication with them.
A system helping services to locate each other is called *Service Discovery* and requires two basic functionalities: Services need to register themselves and a registered service should be detectable within the system [29, p. 236].

2.2.1.2 Domain Name System (DNS)

One solution could be a DNS. A DNS associates a name with the Internet Protocol (IP) address of one or more machines. For instance, a user service could always be addressable via the name *user.mycompany.net*. All DNS entries should be updated during the deployment process. DNS brings the advantage of using a well-understood and well-used standard that can be integrated with almost every technology stack.
On the other hand, the DNS specification itself brings a downside. Each domain entry within the DNS has its own Time To Live (TTL). TTL defines how long a client should wait before a refresh will be necessary. An updated DNS entry, e. g. caused by a service shut down, infers that clients are holding on to the old IP address for at least as long as the TTL states. Furthermore, caching aggravates this problem [29, p. 237].
An alternative to DNS is the Dynamic Service Registry evolved from the downsides of DNS. There are many attractive implementations within the crowded field of Dynamic Service Registries.

2.2.1.3 Zookeeper

One famous Service Discovery is Apache Zookeeper (see [1]) originally developed as a part of the Hadoop project. Zookeeper can be run as a cluster consisting of multiple equal nodes. Zookeeper provides a universal hierarchical namespace for storing information. Whenever information changes, clients will be alerted. Zookeeper is widely-used and well-tested, but in relation to other Service Discoveries, additional features to fulfil Service Discovery have to be implemented on top of Zookeeper itself [29, pp. 238 f.].

2.2.1.4 Consul

Consul (see [2]) supports configuration management (general purpose information) and Service Discovery. It provides more support for these key features than Zookeeper, like additional health checks on services and a DNS server. The embedded DNS server can supply service resource records including an IP and a port for a name. A system already using DNS can integrate consul without further impact to the system [29, p. 240]. Like Zookeeper, Consul is meant to be run as a cluster [21].

2.2.1.5 Eureka

Eureka (see [3]) is an open source Service Discovery used and developed by Netflix. In contrast to Consul, Eureka is designated as Service Discovery only, and therefore it provides no capabilities to manage general purpose information. Eureka provides a basic load balancer with round-robin lookup for services. In addition it offers a Java client and a REST-based interface to enable system interoperability. While the Java client provides additional capabilities like health checks, the disadvantage of using the REST interface is that it shifts more discovery logic towards the client side. This makes Eureka more attractive to Java-based than to polyglot-based environments [29, p. 240]. Eureka supports server replication and caching while preferring availability over consistency [39, p. 14].

2.2.2 Configuration

As software moves through a Continuous Delivery (CD) cycle (e. g. a development, test and delivery stage), it will be deployed in different environments. Each environment demands a different configuration depending on the infrastructure, like a database password, network addresses etc. As described in [29, p. 114], a good approach to face configuration within a distributed system is to manage individual service configurations with an external agent, instead of managing configurations for each service on their own.
The Spring Cloud project within the Spring Framework (see [4]) for example, offers server and client-side support for distributed systems by an externalized key-value-pair storage. The configuration fits well with Spring applications, but can also be used by applications written in

any other language than Java. The configuration server comes with a resource-based HTTP API. The default implementation uses git as back-end storage, since git comes with labeled versions and a wide variety of tools [38].

2.2.3 Load Balancing

Proxy-based load balancers are requested by clients and before forwarding the requested response they call the demanded server depending on runtime metrics. Unfortunately, this centralized approach could become the bottleneck of a microservice system.
Ribbon (see [5]) is an open source client-based load balancer developed by Netflix. With the help of Ribbon, the client has all information necessary to request the determined server directly [39, p. 335]. Ribbon accomplishes its functionality by utilizing an external interface, which must return a list of servers. For instance, this interface could be provided by a Service Discovery tool. In order to enable load balancing, Ribbon applies certain rules like round robin or availability filtering to the fetched server list. System communication behavior can be optimized, since the server list itself can be accessed by considering server regions [28].

2.2.4 Monitoring

Due to its fine-grained functional size, a single microservice is very comprehensible. Unfortunately, complexity shifts towards inter-service communication. It can become very difficult to understand a system state that includes dozens of microservices each living its own life. To understand such a system, *monitoring* becomes inevitable.
Within a monolithic application, the characteristic "*single point of failure*" brings with it the benefit of simpler failure detection, because the system state belongs only to the monolith. In a microservice architecture, multiple services need monitoring, multiple logfiles need to be evaluated and multiple network latencies could cause problems [29, p. 155].
It is important to track a service's host system and its metrics, such as Central Processing Unit (CPU) usage, memory disk space, network Input Output (IO) etc. If a single metric runs out of its bounds, it should be possible to alert the corresponding authority. Next, the access to the system and service logfiles is required as well. And finally, the service itself should be monitored somehow, e. g., via the service response time [29, p. 157].
Since this thesis is rather more focused on design strategy and implementation than on deployment, I will say more about monitoring services.
If scaling is required, the determination of a single error becomes very cumbersome: Multiple services need to collaborate in order to provide capabilities to clients and in addition, most of the services run on multiple hosts.
Regardless of whether the host system or the application is being monitored, the best approach is to collect and centralize all information through logs and application metrics and then aggregate all of them together [29, pp. 157 f.].

In order to manage and query all logs in the entire system, central log management can be accomplished by using *Logstash* (see [6]) in combination with *Kibana* (see [7]). Logstash is able to parse multiple logfile formats and it can send parsed information to downstream systems for further investigation. Kibana is an elastic-search back-end system for viewing logs. Interested persons can make use of a query syntax to extract the appropriate information from all logfiles. Kibana can even generate graphs from the logs received [29, p. 158].
Within a microservice architecture, multiple services will communicate with each other, e. g., by using synchronous calls or asynchronous messaging. Complex communication chains arise, making it necessary to find potential bottlenecks. *Zipkin*, a distributed tracing system, helps find latency problems in microservice architectures; thus developers are able to determine technical or architectural flaws. Furthermore, it manages the collection and lookup of trace data created during communication chains. Zipkin is based on [34], Google's *Dapper* paper. Client applications are instrumented to report timing data to Zipkin. A web UI offers the ability to browse and search communication traces with the corresponding latencies [41].

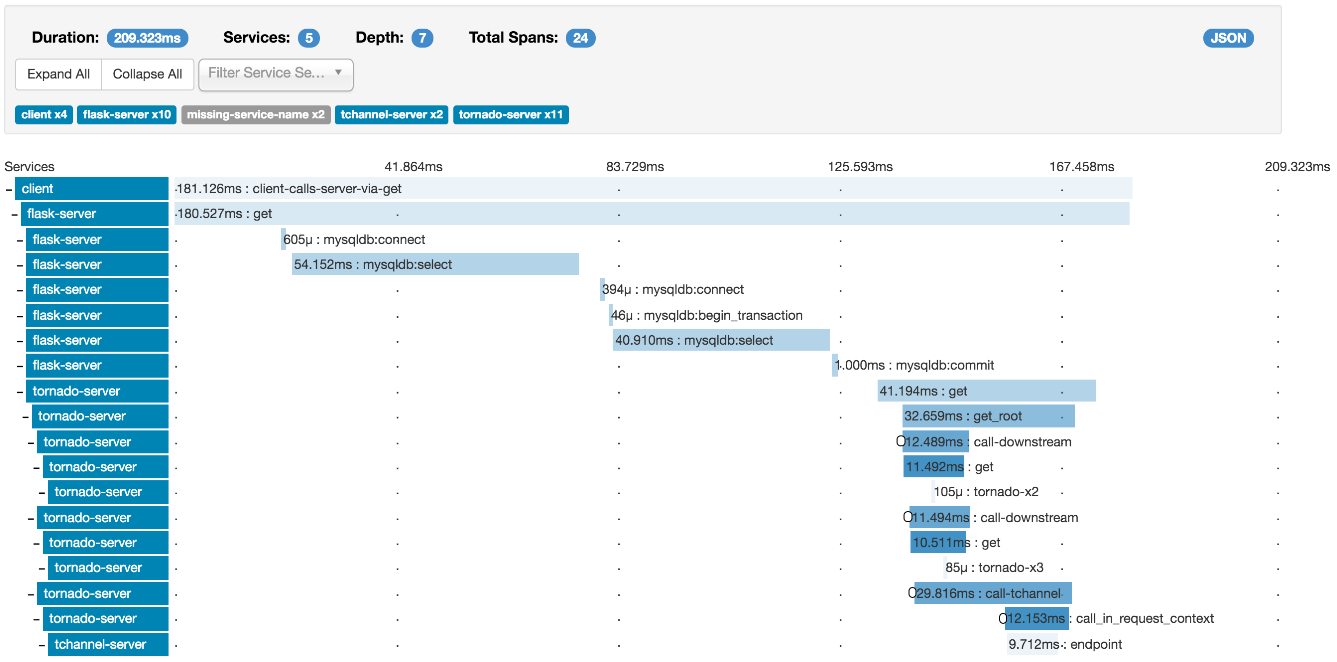


Figure 2.9: An example demonstration of the Zipkin UI taken from [41]. The web view determines a specific trace of a communication, containing multiple services, such as *client*, *flask-server*, *tornado-server* and *tchannel-server*. Within the graphic, we can see the latency of the client's get-call, which is about 181.162 ms. Furthermore, it is visible how long each call-step takes for a specific service. For example, the flask-server applies a MySQL select with a latency of 54.152 ms.

2.2.5 Communication

2.2.5.1 Remote Procedure Calls (RPCs)

The idea behind a Remote Procedure Call (RPC) is to make a local function or method call and have it executed on a remote server. Technologies implementing RPC differ by having an interface definition language or not. Some implementations require both the caller and the receiver to use the same platform (e. g. as Java Remote Method Invocation (RMI) does). The payload format varies as well: some implementations prefer a binary data format (like Java RMI, Thrift, or protocol buffers), while other technologies make use of Extensible Markup Language (XML), e. g. Simple Object Access Protocol (SOAP).
RPCs are very easy to use, since an RPC works just as simply as a synchronous method call while ignoring all the machinery that runs behind it. Unfortunately, there are some potential pitfalls associated with RPCs: they should not be abstracted to the point where the network is completely hidden, since this may encourage the wrong API design and negative performance behavior from the program. Changeability and extendibility of statically defined interfaces can increase workload. For example, if the communication interface used to generate client and server stubs is extended, all consumers that do not need the most recent stubs nevertheless have to import it [29, pp. 46 ff.].

2.2.5.2 Representational State Transfer (REST)

The architecture style *REST* is inspired by the web and is an alternative technology to RPC. Using REST over HTTP is a sensible default choice for service-to-service communication, but REST can be used by any other protocol as well. The most important concept behind REST is the idea of *resources*. Each resource is a textual representation of a domain object like a customer. On request, the server creates different representations of a domain object. The client may ask for a specific resource representation (e. g. JSON or XML), whereby the external representation of a resource is completely decoupled from its internal representation used by the server system. Applying REST with HTTP is advantageous, since HTTP provides a huge technology and tool set and is well-suited to handle large volumes of traffic. On the other hand, HTTP in general might not be a good idea if low latencies are required [29, pp. 49 ff.].

2.2.5.3 Apache Kafka and RabbitMQ

When messaging is required, there are dozens of Enterprise Service Bus (ESB) systems you can choose among. Within this section, the most popular choices are compared: *Apache Kafka* and *RabbitMQ*.
RabbitMQ was one of the first message brokers to achieve a reasonable level of features, client libraries, development tools and quality documentation. The idea behind the development of RabbitMQ was to reach cross-language flexibility, which was achieved by implementing Advanced

Message Queuing Protocol (AMQP), an open wire protocol for messaging with powerful routing features.
RabbitMQ is designed as a general-purpose message broker with several variations of communication style patterns, such as point to point, request/reply and publish-subscribe. The model of RabbitMQ focuses on consistent message delivery and consists of a smart broker as well as dumb consumers.

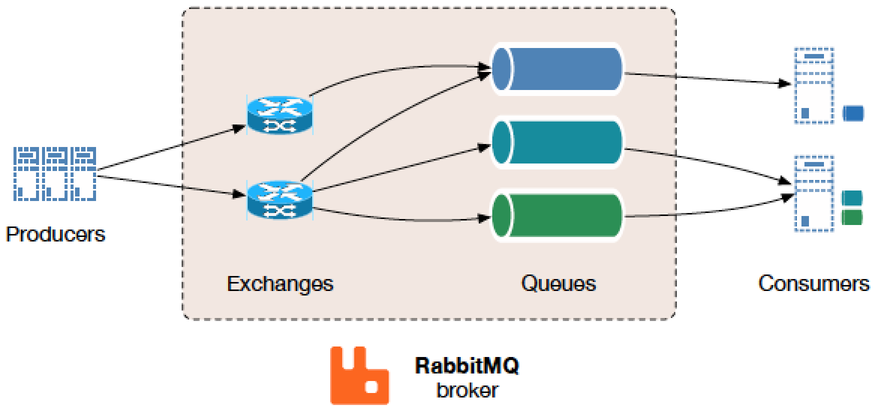


Figure 2.10: The basic architecture of RabbitMQ [27].

RabbitMQ supports synchronous and asynchronous communication as needed. As displayed in Fig. 2.10, a publisher can send messages to so-called *exchanges*. Between these exchanges and the consumers there are *queues* to retrieve messages from. Producers are not burdened with hardcoded routing decisions, since they are decoupled from the queues. In addition, RabbitMQ can be deployed in a number of distributed scenarios whereby it requires all nodes to resolve host names. Without having any external dependencies, RabbitMQ can be set up for multi-node clusters as well as for cluster federation.

Apache Kafka is developed in Scala and was initialized by LinkedIn. Today, Kafka is well adopted within the Apache Software Foundation ecosystem of projects. Its design leads to high volume publish-subscribe messages and streams. The data transaction is fast, durable and scalable. At its heart, Kafka provides a durable message store providing streams of records in categories called *topics*.

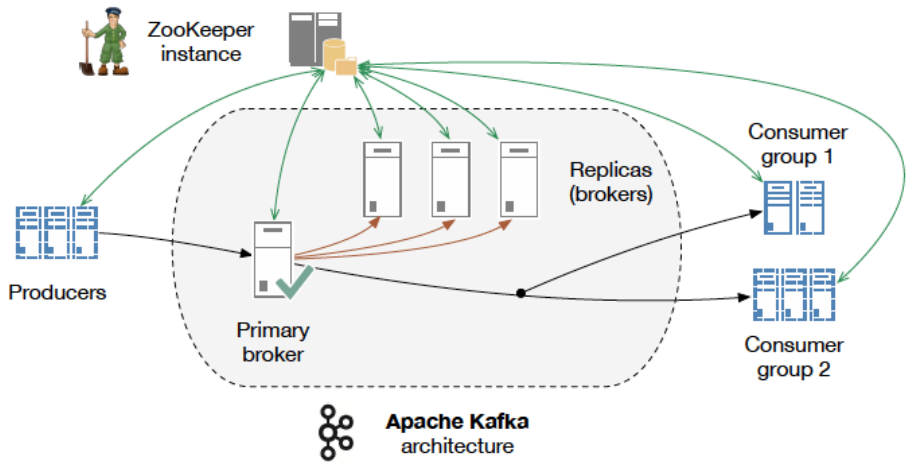


Figure 2.11: Global architecture of Apache Kafka. All agents are registered with a ZooKeeper instance. The replication factor of the system (brokers) is four [27].

Every Message within Kafka consists of a key, value and timestamp. Unlike RabbitMQ, Kafka brokers are dumb while consumers are required to be smart to read the Kafka buffer. Since Kafka does not track which messages were read by each consumer, consumers have to track the location of each message in their log (consumer state) on their own. As a consequence, Kafka can support a large number of consumers and retain large amounts of data with very little overhead. In terms of dependencies, Kafka is not self-sufficient and requires Zookeeper [22].

Kafka and RabbitMQ are two message-oriented middlewares with a fairly different set of capabilities allowing messages to be queued and tasks to be distributed to one or more consumers. Consumers can process messages whenever they are up and running, since messages are written to persistent memory. Both technologies isolate message production and offer asynchronous message consumption [27].
In [27], some more recent differences are outlined in detail. In summary, if the application is Java-based, needs access to stream history (for message replays), and requires very high data throughput, Kafka would be preferred. On the other hand, RabbitMQ is more favourable if well-understood consistency guarantees are required for message delivery and different messaging types. Another use case would be the requirement for combining existing protocols like AMQP 0-9-1, Simple (or Streaming) Text Orientated Messaging Protocol (STOMP), Message Queue Telemetry Transport (MQTT) or AMQP 1.0 [22].

2.2.5.4 Spring Cloud Stream

Spring Cloud Stream, a framework built upon Spring Boot, helps to design message-driven microservice applications.

Spring Cloud Stream abstracts one specific messaging infrastructure: RabbitMQ, Redis or Kafka. Messages are sent and received using a declarative approach. Spring Cloud Stream provides the capability to use multiple messaging providers in one application, for instance, connecting an input stream from Kafka to an output stream of Redis.

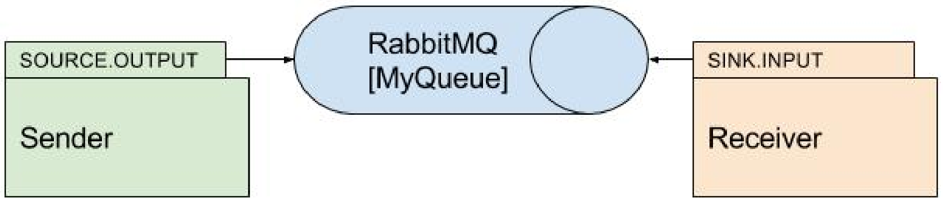


Figure 2.12: The basic concept behind Spring Cloud Stream using RabbitMQ [33, p. 252].

As shown in Fig. 2.12, the sender defines a logical queue reference called *Source.OUTPUT*. This queue reference serves as a destination targeted by the sender's messages. The receiver defines the logical queue reference called *Sink.Input*, which can be used by receivers to retrieve messages. The physical binding of OUTPUT and INPUT, called *MyQueue*, is managed via configuration. Both queue references, Source.OUTPUT and Sink.INPUT, point to *MyQueue* [33, pp. 252f.].

2.2.6 Persistence

2.2.6.1 Consistency Trade-offs

As already outlined in chapter 2.1.2.14, the CAP theorem forces system architects to make certain trade-off decisions, especially in terms of availability and consistency. The latter can be achieved at different levels:
Strict consistency is the most stringent level. In order to accomplish strict consistency, every read will always return the value most recently written. This behavior is desireable, but the implementation can quickly become very hard to achieve when data is distributed. To utilize the most recent data values on a distributed system, a global clock is essential and it must be capable of timestamping all operations regardless of the data's location.
Causal consistency resembles strict consistency, but is slightly weaker. The idea behind causal consistency is to determine the event source within the distributed system instead of depending on a global clock. All potentially related causal writes must be read in sequence.
In a distributed system, with *weak (eventual) consistency*, all updates are propagated throughout all replicas. This action will take some time, but eventually all replicas will be consistent [23, p. 21].

2.2.6.2 Cassandra

In terms of scaling, a relational normalized database can reach its limits. This behavior is caused by transaction consistency requiring partial database locks. Under heavy load, competitive clients queue up in order to read or write data. A possible solution includes vertical scaling, e. g. upgrading hardware. But system hardware might face its limits as well. Forming a database cluster with multiple machines introduces new problems, like consistency and failover scenarios [23, p. 4].
The Apache project *Cassandra* is an open source project based on Amazon's Dynamo created by Facebook. Cassandra's data model is based on Google's Bigtable. Cassandra is designed as a highly-available, fault-tolerant, row-oriented database [23, p. 17].
The characteristic that enables high availability is Cassandra's distributed and decentralized nature. Every node within a Cassandra cluster is identical - there is no master/slave relationship within the Cassandra ecosystem. The distribution of identical nodes prevents a single point of failure [23, p. 18].
As soon as new nodes are started, each node gets copies of some or all available data. Therefore, no disruptions or reconfigurations of the entire cluster are necessary. The ability to add or remove nodes from the cluster without any trouble is described as *elastic scalability* [23, p. 19].
Failed nodes within a Cassandra cluster can be replaced without any downtime and data can be replicated into multiple data centers to offer improved performance and prevent downtime in case of a catastrophe. These properties make Cassandra *highly available* [23, p. 20].
Dynamo and Cassandra chose to always be writable, with the trade-off that the complexity of reconciliation shifts towards read operations.
Cassandra offers *tunable consistency* for every operation performed by letting the client define a replication factor for a given data-set and individual consistency levels. The replication factor defines the number of nodes addressed by an upcoming update propagation. The consistency level decides how many replicas within the cluster must acknowledge a write operation or respond to a read operation in order be considered successful.
Within Cassandra, the user can define the replication factor describing how much performance should be sacrificed for consistency [23, p. 22].
Besides the consistency trade-off, the usage of Cassandra limits the database model in comparison to relational databases. Cassandra does not support joins and referential integrity across tables but instead encourages well-proposed denormalization of tables, as you will see later on, in chapter 4.2.4 [23, p. 81].

2.3 Integration Technologies

2.3.1 Spring Framework

As mentioned in chapter 2.1.2.2, building microservices can provide technology freedom at first, but when developing multiple microservices, a common problem set must be solved over and over again; for example, embedding and configuring a web server or enabling security standards. Software structuring and coding conventions have to be enforced somehow for each service individually.
In practice, I would like to find a healthy balance between a restricted DRY principle (see chapter 2.1.2.13) and commonly used source code.
The JVM based *Spring Framework* (see [37]) offers a comprehensive programming and configuration model for Java enterprise applications. It comprises dependency injection helping to keep code clean and readable. The business logic is separated from transaction logic by external configuration files. This approach supports easier deployment to multiple environments. In addition, Spring Framework provides support for web applications based on the Model-View-Controller (MVC) pattern and RESTful web services, which are both potential requirements for a microservice implementation [37].

2.3.2 Spring Boot

Spring Boot is based on the Spring Framework and helps to speed up development for stand-alone Spring applications. By the aid of Spring Boot, embedding web servers like *Jetty*, *Tomcat* or *Undertow* can be accomplished directly without the need to deploy Web Application Archive (WAR) files. In addition, production-ready features such as metrics, health checks and externalized configurations are supported. In many cases, Spring Boot offers a configuration automation helping to deal with Spring's dependency injection configuration [35].

2.3.3 Spring Cloud

Spring Cloud helps developers to set up some of the common patterns required in distributed systems (e. g. centralized configuration management, Service Discovery, etc.). Since nearly every microservice gets in touch with distributed system coordination, repeated boiler-plate patterns can be decreased to a minimum by the help of Spring Cloud.
Spring Cloud builds upon Spring Boot and provides many libraries that enhance the behavior of an application when added to the class path [36].

2.4 Domain Driven Design (DDD)

2.4.1 A Concept to Design Microservice Architectures

When migrating from a monolithic application to microservices, one of the most important aspects is segmenting the existing monolithic system. This chapter will outline the ideas behind DDD in terms of a conceptual modelling approach that can be utilized as a guideline to design microservices.

2.4.2 About DDD

Eric Evans' often-quoted book *Domain Driven Design* (DDD) offers a systematic approach to design software in complex domains. Thereby, as a base of his concept, Evans introduces *Ubiquituous Language*, a communication approach for teams aimed to decrease misunderstandings and misconceptions. The main idea behind Ubiquituous Language is enforcing consistency of terms and definitions within the software and its domain, e. g. identifiers, database schemata etc. [39, p. 44].

2.4.3 Building Blocks

In order to design a domain model, Evans identifies *Building Blocks*, fundamental patterns found in any domain.
An object with its own identity (a state) is called *entity*, for example a customer whose address should be mutable.
So-called *Value Objects* do not have their own identity. A measurement reading, for instance, could be a value object, since there is no need to address and change it.
Compounded domain objects are called *aggregates*. An aggregate can be handled as a black box offering specific invariants. An order could be an aggregate composed of multiple order items, each with an own value. An invariant could require that the minimum order value must be at least 10 Euro.
Business logic could be implemented within an entity or an aggregate. But sometimes, external entities or aggregates must be referenced within the business logic process, which blurs domain borders. Within DDD, this task is accomplished by *services*. An example could be the ordering process, including an order aggregate and the customer entity.
The next building block is called *Repository*, which should offer access to all entities of a certain type. In some cases, persistence is implicated within the repository.
The last building block, named *factory*, is meant to construct complex objects such as aggregates containing invariants and associations [39, pp. 45 f.].

2.4.4 Data Integrity and Bounded Context

When source code is based on distinct models, software tends to become buggy, unreliable and difficult to understand. Communicating team members become confused, because often there is a lack of knowledge about what context a model should not be applied in.
An adequate model has to be logically consistent throughout. Thereby, contradictory and/or overlapping definitions should be avoided [18, p. 328]. Each model applies in an explicitly defined context. In terms of team organization, each context must have explicitly defined boundaries, database schemas, and may only be used within specific parts of the application. The model should be kept consistent within these bounds while simultaneously not becoming distracted by outside issues [18, pp. 335 ff.].
Evans defines a Bounded Context in [18, p. 336] as follows:

> A Bounded Context delimits the applicability of a particular model so that team members have a clear and shared understanding of what has to be consistent and how it relates to other contexts.

In terms of microservices, we can use the concept of a Bounded Context (BC) in order to achieve a functional decomposition. Therefore, one BC per single microservice is defined. Within its bounds, model integrity can be kept clear and consistent. In other BCs, other models are applied with different terminologies, concepts and rules.
Each domain model is only valid in limited areas of a system. As an example, E. Wolff states in [39, p. 46] that in an e-commerce application, only the quantity, size and weight of the ordered goods are interesting in the context of a delivery. Whereas prices and tax rates are relevant for the accounting context.
For technical reasons, it can be possible to have multiple microservices within a single BC, for example to enable a better scale of certain business logics. But multiple BCs should never be embedded within a single microservice, because this approach can lead to contradiction with the goal of accomplishing independent feature development [39, p. 52].

2.4.5 Context Map and Relationships

Within multiple-team projects, there might be a lack of information about service context boundaries across teams. Unknowingly, this will lead to changes that blur the edges or complicate interconnections. Therefore, by defining relationships between contexts, confusion can be reduced. This enables a global view of the model contexts on a given project. Evans introduced so-called *Context Maps*.
A Context Map defines the relation of BCs which can be drawn after all BCs have been located. Each single kind of cross-context communication should be considered before defining a relationship type. A context map offers a much better project transparency for stakeholders, which will ease certain design decisions by a lot. Answers to questions like *"what do we need to change in*

order to accomplish a specific functionality?" will be found much more easily [18, pp. 344 ff.].
Within the chapter *Strategic Design*, Evans enumerates seven strategies. Each strategy defines a certain relation of two BCs for unifying and integrating models. Basically, each strategy brings its own trade-off: the seamless integration of functionality against the additional effort of coordination and communication. Independence is sacrificed for smoother communication.

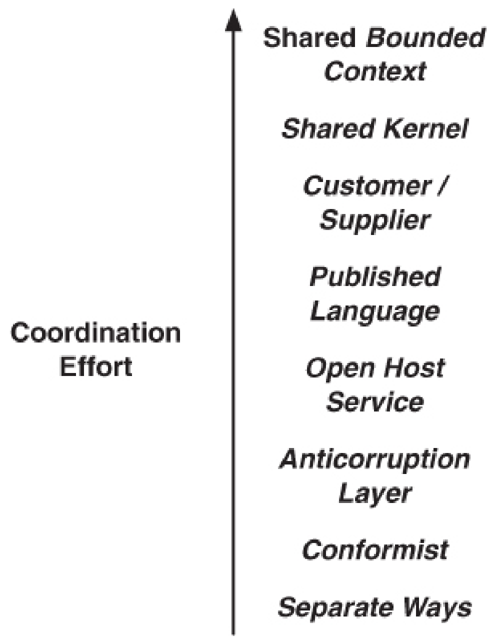


Figure 2.13: The coordination efforts in relation to the collaboration approach of multiple teams working on different BCs [40].

Within a *shared kernel* relationship, domain models share some components, but differ in other sub sections.
When using a *customer/supplier* relationship, the supplier system offers a sub system for its customers, within which the customer is to define the system's structure.
A model can be defined as a common language between two BCs. Evans calls this approach *published language*. Due to its broad usage, a published language is usually hard to change.
A BC offering some specific services is called *open host service*. Offered services can be claimed by every client BC. Each client has to handle the the offered model's integration by himself.
The *anti corruption layer* translates a given domain model into another one. This approach enables full decoupling of two layers. For example, this can be useful when abstracting legacy application models.
Within a *conformist* relationship, the caller exactly uses its callee's model. This can be useful when the delivering system has been designed by experienced domain experts who know plenty of operational scenarios for its offered model [39, p. 48].

2.5 GridVis: a Technical Conclusion

2.5.1 Projects, Devices and Values

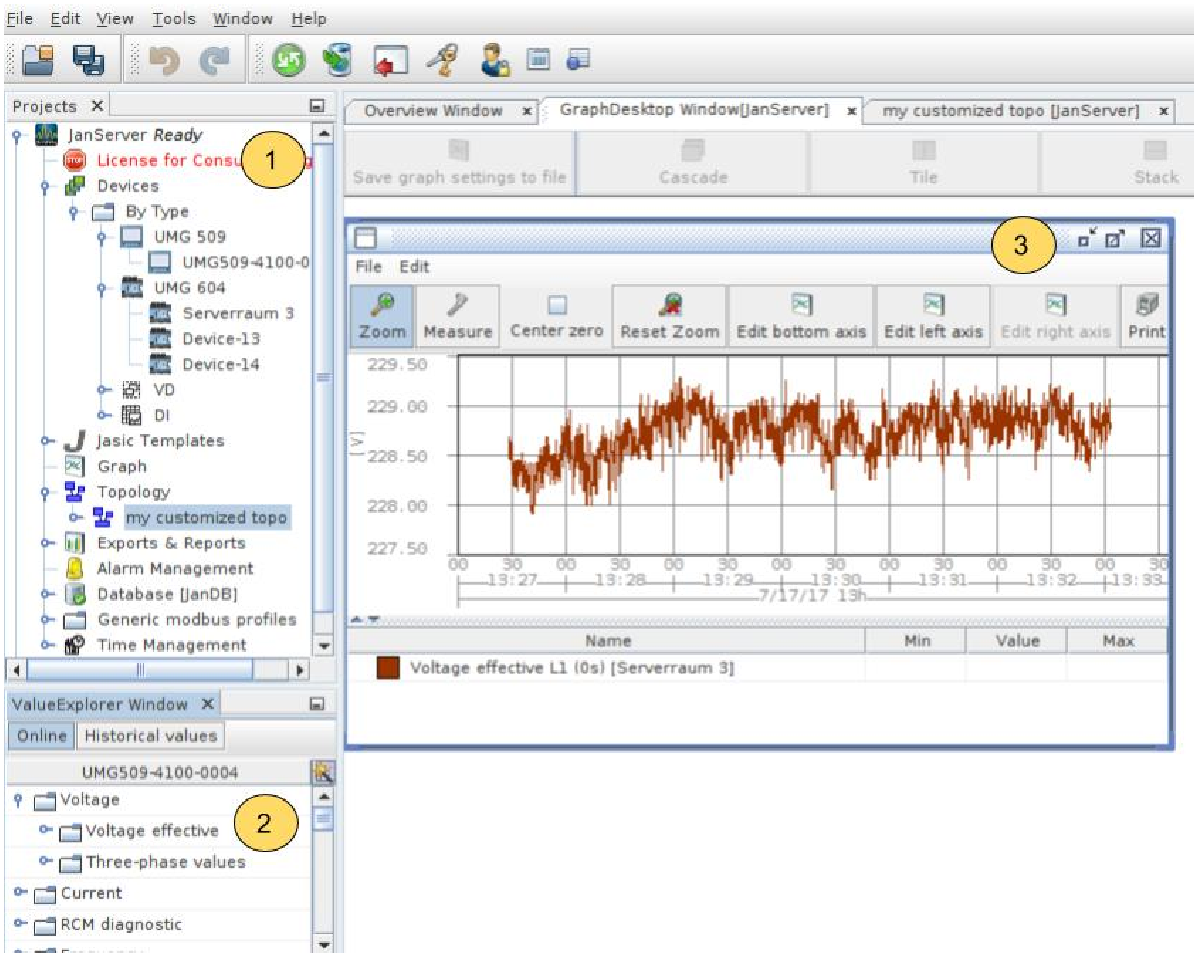


Figure 2.14: GridVis Desktop UI. (1) displays all project-specific options underneath the project node. The project itself is named *JanServer*. Underneath the *Devices* node, all connected devices are listed categorized by type. As you can see, there are some more very specific and complex features, such as *Jasic Templates*, *Graph visualization*, *Topology*, *Exports and Reports*, *Alarm Management*, etc., located underneath the project node. (2) lists measurements the user can select from (e.g. in order to display a graph). The measurement list loading is triggered by a previous device selection. (3) shows the previously mentioned *Graph* component. The graph displays the last recent measurement readings of device *Serverraum 3* (engl. server room no. 3) for the measurement *voltage effective L1*.

GridVis is capable of communicating with more than twenty different device types; a detailed list of featured devices can be found in [8]. Each device can record a technically limited set of measurements such as voltage, current, frequency, power, energy and more. GridVis is then able to browse these measurements and process them individually. All devices that GridVis has access to are bundled within *Projects*. Each project contains additional possibilities of configuration: database management, data reports and exports, alarm management and more (as shown in Fig. 2.14).

2.5.2 NetBeans Platform

2.5.2.1 Rich-Client-Platform

Within a client-server architecture, the term *rich-client* is related to a client application processing data delivered by a server. The client comes with a UI and is extensible somehow, e.g. through loading plug-ins or using a module system.
Since nearly all desktop applications have very similar demands, such as menus, toolbars, status bars, file management, persistence, etc., a *rich-client-platform* offers a framework to speed up development. Usually, configurability and extensibility are promoted by such platforms.
The GridVis system was built upon *NetBeans*, a *rich-client-platform*, using its functionalities: UI components (windows, toolbars etc.), wizard and file system framework, data and central service management, representation framework and internationalization [11, pp. 27 ff.].

2.5.2.2 NetBeans Module System

The approach to separate functionalities in multiple modules offers more flexibility in terms of expandability. Each module consists of Java classes, defined interfaces and dependencies. The *NetBeans Runtime Container* can load each module at runtime dynamically. Each module is an independent unit loaded by its own class loader. Dependencies are defined in a manifest file used by the NetBeans runtime container [11, pp. 33 ff.]. A customer could add additional features by installing a NBM via the NBM Manager provided by the NetBeans platform.

2.5.2.3 Lookup Concept

The lookup concept offers central service management, which enables decoupling of system components, handles instances of objects, and introduces inter-module-communication. Modules can offer instances that can be published as a service provider or type and safely query existing services using the lookup [11, pp. 89 f.]. All published classes are *singletons*. This implies that the NetBeans runtime container only creates one instance and passes the instance's reference around.
Service providers and lookups may differ in their scope. Some service providers are installed in a global context while others are installed within a project scope only. The possibility of handling instances per project scope offers the ability to manage multiple system states per application.

```java
// An example import via global lookup from an external NBM
final ExternalClass imported = Lookup.getLookup().lookup(ExternalClass.class);
```

2.5.3 Modularity

The entire GridVis system is composed of a service and a desktop application built via Apache Maven. In addition, the service application is partially built by NPM and Grunt. Each application contains a set of *module suites*, each encapsulating a given functionality, e. g. database access, device communication, etc.
A module suite is a nested module (NBM or Maven) containing a set of coherent sub modules, usually a set of *API modules* and a set of *implementation modules*. Within the implementation modules, business logic is pooled with exported service provider functionalities while the API modules contain interfaces that can be referenced and loaded via the NetBeans lookup concept. Each NBM is constructable with the help of Apache Maven. POMs are used to structure the GridVis module system at fine grain while NBMs are mainly used to provide plug-in availability and to extend Java's scoping mechanisms. POMs solve multiple problems, like versioning, dependency management and tooling, e.g., the extraction of translations.

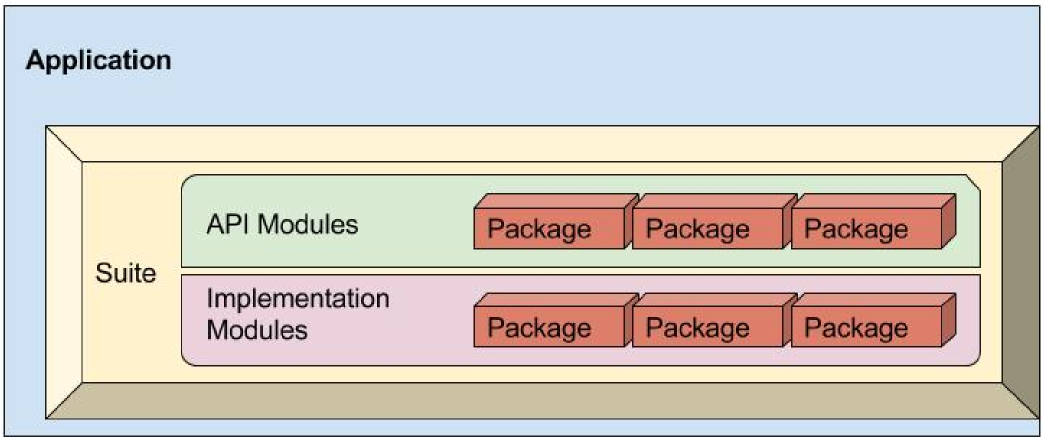


Figure 2.15: An overview of the modularity within GridVis. Each Application references multiple suites. A suite is composed of an API and an implementation module set. Each module includes multiple packages with Java source code.

2.5.4 Persistence

The GridVis system stores a wide set of data tables and data schemata. For that purpose, different databases are supported: Microsoft Structured Query Language (MSSQL), My Structured Query Language (MySQL), and Derby and Janitza Database (JanDB).
All tables storing measurement readings are optimized for specific queries, which can be searched

for timestamps, devices and measurements.
The so-called *Plugin Config* provides optimized storage for textual data, e.g., reports and different configuration parameters. This storage can be accessed by using the NetBeans lookup from an externalized plug-in, that implements the corresponding *Service Provider Interface*.
The last persistence technology used within GridVis is binary storage. Its purpose is to persist binary data, such as images or even NBMs.

Chapter 3

Migration Process

3.1 Migration Strategy

3.1.1 Common Migration Strategy

An application with the scale of GridVis (or larger) cannot be completely reimplemented into a new architectural paradigm in a single iteration. A complete rewrite implies huge technical risks and cost. In relation to business demands, for many companies, a complete rewrite is impossible because customers will not agree to wait for years in order to receive a desired bug fix or feature. In my view, the most suitable way around this is pursuing an iterative approach: The extraction of one single functionality at one time.

I assume that the software project's initial situation is decisive as to whether a migration can be successful or not. Being successful involves the work-load's size behind each migration step. During this work, I have experienced that the difficulty of an extraction is strongly based on the software's structure and code quality. While migrating, the quantity of reused code, the size and functionality of dependencies, as well as the effort to replace individual dependencies are all important. Dependencies vary in type, e. g., some common dependencies are related to the UI, persistence or other business layers. They differ in their size and become manifest in classes, packages and modules. Basically, the fewer dependencies exist within a portion of code, the easier the extraction will be.

In [19], Michael C. Feathers introduces the concept of a so-called *seam*, and in [29, p. 79], Newman summarizes such a seam as follows:

> A seam is a portion of code that can be treated in isolation and worked on without impacting the rest of the codebase.

Consequently, while migrating, finding seams is very important. Bounded Contexts (BCs), as described in chapter 2.4.4, make excellent seams in relation to microservices, since they represent cohesive and loosely coupled boundaries within an organization.
Depending on the programming language applied, software is structured in some set of modules and packages, which are a good first step towards finding appropriate service boundaries [29, p. 80]. Furthermore, unit tests are another springboard to find well-fitting seams: They cover a small, mostly independent functionality and make use of mocking and stubbing all of the required dependencies.

3.1.2 Project Specific Migration Strategy

Due to GridVis system's structure, mentioned in chapter 2.5, the investigation of seams starts with an analysis of all GridVis suites. In summary, suites within GridVis may contain either an almost standalone feature or a more commonly used technical library.
As I understood the concept behind BCs, when designing microservices, source code must be bounded as a functional, cross-technological unit (as visualized in Fig. 2.7, chapter 2.1.2.12), because each of them should contain its own access to persistence and possibly its own UI layer to keep the service boundaries clear.
Before examining certain dependencies, each module suite is evaluated by its functional intention.

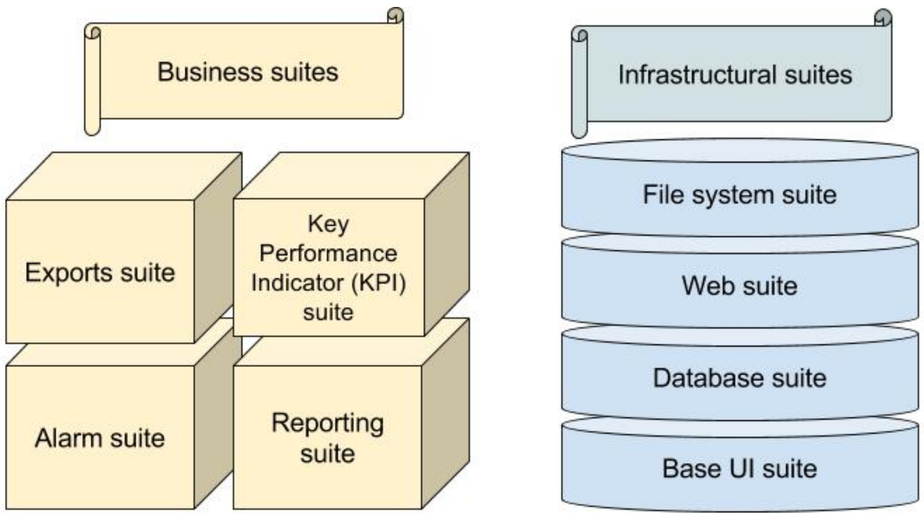


Figure 3.1: A subset of existing GridVis suites separated by their functional intention, depending if they are composed as a functional set of business logic forming a specific feature (yellow blocks) or a technologically composed set forming a framed access to a certain infrastructure (blue cylinders).

As an example, Fig. 3.1 opposes a subset of suites within the GridVis system. Both suite sets are separated by their intention, depending if they are designed as a composition of business functionalities or as an infrastructure-specific framework.

On the left, business module suites are visualized as yellow blocks. Each block contains a business suite encapsulating a specific feature.
On the right, blue cylinders represent commonly used infrastructure-oriented suites. Each of them encapsulates a certain infrastructure level (e.g., UI, persistence, web, etc.) and provides commonly used code.
For example, the *Reporting Suite* mainly contains all logic necessary to create several types of reports, such as cost, energy or quality reports. Naturally, creating a report needs configuration and information should be displayed somewhere. Therefore, the Reporting Suite contains a UI layer, in which individual reports' UI elements are defined within their module bounds. In terms of the DRY principle, common rendering tasks are implemented within the Base UI suite displayed on the right-hand side of Fig. 3.1. Base UI provides access to extended NetBeans platform UI components. Consequently, the Base UI API needs to be referenced as a dependency within the Reporting Suite in order have its rendering classes used. For persistence, the process is very similar: The Reporting Suite needs the Database Suite API as a dependency in order to persist its configuration, and so on.
The extraction of a certain technology layer, such as the *Base UI layer*, should be strictly avoided. Services separated by technological capabilities slow down development in terms of changeability, maintenance and team communication, as described in the chapters 2.1.2.12 and 2.1.2.13.
Example: Let's assume that multiple services share a common Base UI microservice responsible for applying multiple rendering tasks. As a consequence, each change on an individual client-microservice UI component may also imply a change to the commonly shared Base UI microservice. If the latter broke, all dependent clients would break, too. In addition, team organization would also become much more difficult because of blurred team-service-affiliations. All teams owning services depending on this Base UI microservice must communicate and agree upon the changes. Therefore, a migration should only be applied to business-intended suites, e.g., the Reporting-Suite.
After the number of potential modules for a migration has been limited, the remaining business suites can be evaluated for a possible migration. One approach is focusing on a very low risk suite first to efficiently learn best practices before migrating more complex sub systems. Architectural benefits and drawbacks of a specific module extraction should be considered as well, as outlined in chapter 2.1.2.

3.2 Extraction: Alarm System

3.2.1 Motivation for Extracting the Alarm System

To provide an empirical case study, I have decided to extract the functionalities of the *Alarm System* suite for the following good reasons:

1. The Alarm System encapsulates all relevant functionalities, including business logic, domain objects and the interfaces necessary to run all features. This suite itself is very independent and could be easily deleted without a larger impact to the rest of the system.

2. It involves the need for resilience: Customers want to receive alarms day and night, no matter whether there is a bug within another sub system or not.

3. Single partial functionalities within the Alarm System can be run as x and z-axis scaled instances (see scaling types outlined in 2.1.2.9). Within larger projects (a few thousand devices), this approach can optimize system performance at very high data throughput [42].

3.2.2 Analysis of the Alarm System

In order to extract the underlying domain model, every functionality within the Alarm System suite needs to be analysed and named.
Within my first studies of the Alarm System's legacy code, I realized that it is not very efficient to analyse legacy source code like classes, database schemata or similar implementation structures in order to understand the system's behavior. Most of the existing source code, such as classes modelling domain entities, can be misleading, although such investigations may bring some detailed information about certain constraints.
Rethinking legacy code structures and entity relationships is necessary, because the system should be understood as a domain model instead of a concrete implementation. In my experience, a more promising approach is the exploration of certain feature functionalities from the user's point of view, e.g., by interacting with the UI, reading and understanding the output of the software. Naturally, each kind of documentation, for example, a functional specification document, is a great convenience.
Based on this approach, a migration process can be established including the following steps:

1. Detecting, understanding and naming relevant domain objects within the system.

2. Categorizing domain objects within defined bounds (each bound represents a BC forming a microservice later on).

3. Analysing processes within the BCs in detail to identify data requirements within the system (this implies where communication between contexts is required and where it is not).

4. Finding adequate technologies to realize system requirements (such as solving common distributed system problems like Service Discovery, messaging, etc.).

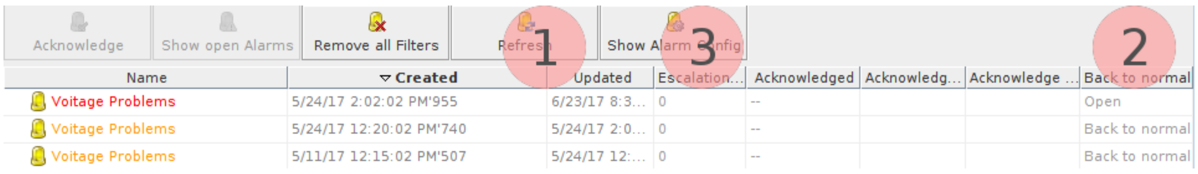
After having run through the above-mentioned steps, the microservice implementation process with feature reimplementation can be started. Through the example of the GridVis Alarm System, this migration process will be outlined in detail in the following chapters.

3.2.3 Detecting, Understanding and Naming Relevant Domain Objects

The Alarm System, also known as Alarm Management, was designed to quickly send messages as soon as defined faults occur, e.g., if measurement readings are running out of bounds. An **alarm** is defined as an event implying a prompt reaction by a responsible energy or operation manager. During an alarm occurrence, different **actions** can take place, like the sending of reports to multiple persons (or systems) via configurable channels [16]. As soon as an alarm stays unacknowledged for *too long*, the alarm's urgency rises and a new action could be triggered. The level of urgency is declared as the **escalation level** while an alarm's recognition is declared as an **acknowledgement**. An acknowledgement is applied by a **user** providing an **acknowledgement comment** at a given moment called an **acknowledgement date**.

Application example: After voltage effective has risen above 230 Voltage (V), a new action is applied and an alarm report is sent to the local energy manager. Unfortunately, the receiving energy manager is ill on this working day. As a consequence, the alarm stays unacknowledged. After one hour has passed with voltage exceedance, the alarm increases in urgency and reaches a new escalation level, which triggers a new action: A report is generated and sent to the head of the department, who acknowledges the alarm himself and engages the problem by sending an engineer to the related device to fix it.

From the user's point of view, the currently existing Alarm System is separated into two main sections: The *alarm overview* and the *alarm configuration* section.
As shown in Fig. 3.2, the alarm overview section displays the **alarm history**, a tabular overview of all actual and historical alarm events. Each row item within the alarm history table provides an **alarm log** containing a so-called **alarm detail** with information about the related alarm. Such a row item includes timestamps defining when the alarm was created or updated (1), the current state of the alarm (2) (new, set back to normal, acknowledged or reopened) and the current escalation level (3).



Name	▽ Created	Updated	Escalation...	Acknowledged	Acknowledg...	Acknowledge ...	Back to normal
Voltage Problems	5/24/17 2:02:02 PM'955	6/23/17 8:3...	0	--			Open
Voltage Problems	5/24/17 12:20:02 PM'740	5/24/17 2:0...	0	--			Back to normal
Voltage Problems	5/11/17 12:15:02 PM'507	5/24/17 12:...	0	--			Back to normal

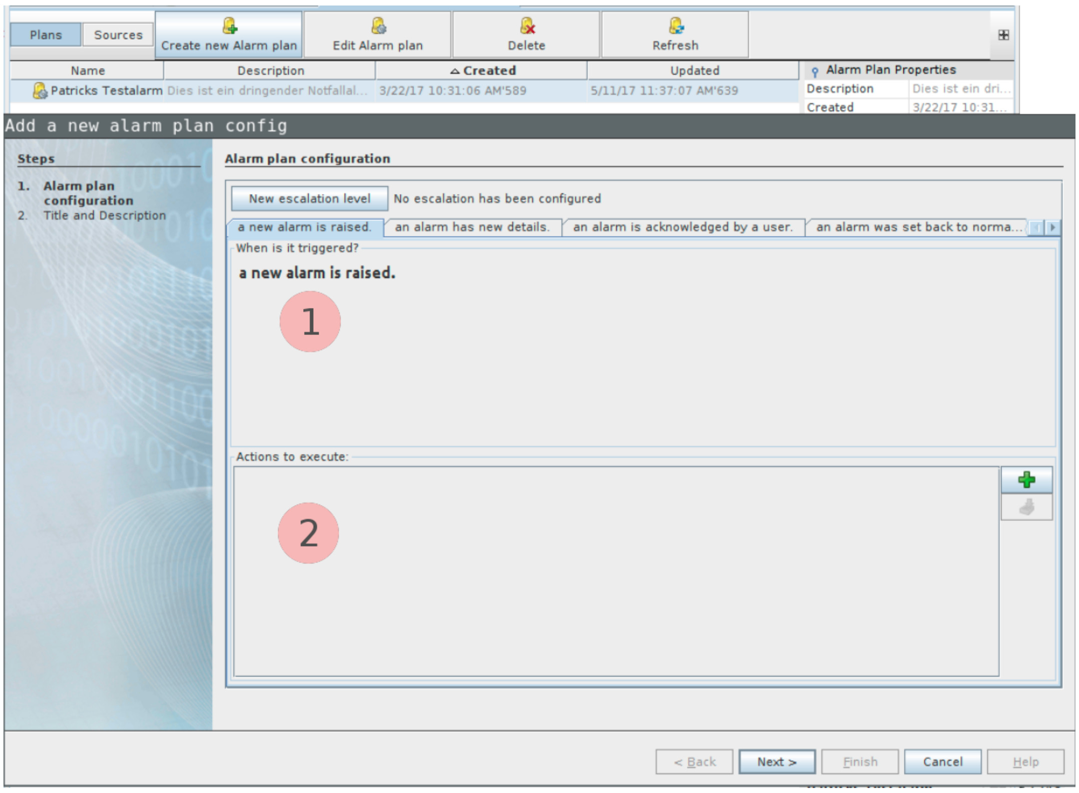
Figure 3.2: Alarm overview UI within the GridVis Desktop application.

The second section, the *alarm configuration*, is divided into two subsections responsible for configuring the Alarm System.
The first subsection is called *alarm plan configuration*, which is intended for defining alarm sources and mapping alarm events to actions. The user begins by creating a new **alarm plan**, see Fig. 3.3. Within this configuration step, the user can choose which **alarm event** (1) will trigger certain actions (2).

The alarm event describes a change to the actual system covering the following states:

- a new alarm *was created* (new defective system behavior detected)
- *has new details* (the system behavior is still defective, but the cause has changed)
- *was acknowledged*
- *was set back to normal by its source* (system behavior was corrected somehow)
- *escalation* (a configured escalation level was reached)

An **action** can include the execution of a custom program or the dispatch of an email containing crucial information.
In order to capture the rise in urgency, the alarm plan configuration initializes the **escalation driver** defining when the escalation level of an existing alarm will rise. For example, a rise could be applied after a certain time has passed. The final configuration step of an alarm plan configuration is setting up a description and a name.



Name	Description	Created	Updated
Patricks Testalarm	Dies ist ein dringender Notfallal...	3/22/17 10:31:06 AM'589	5/11/17 11:37:07 AM'639

Alarm Plan Properties	
Description	Dies ist ein dri...
Created	3/22/17 10:31...

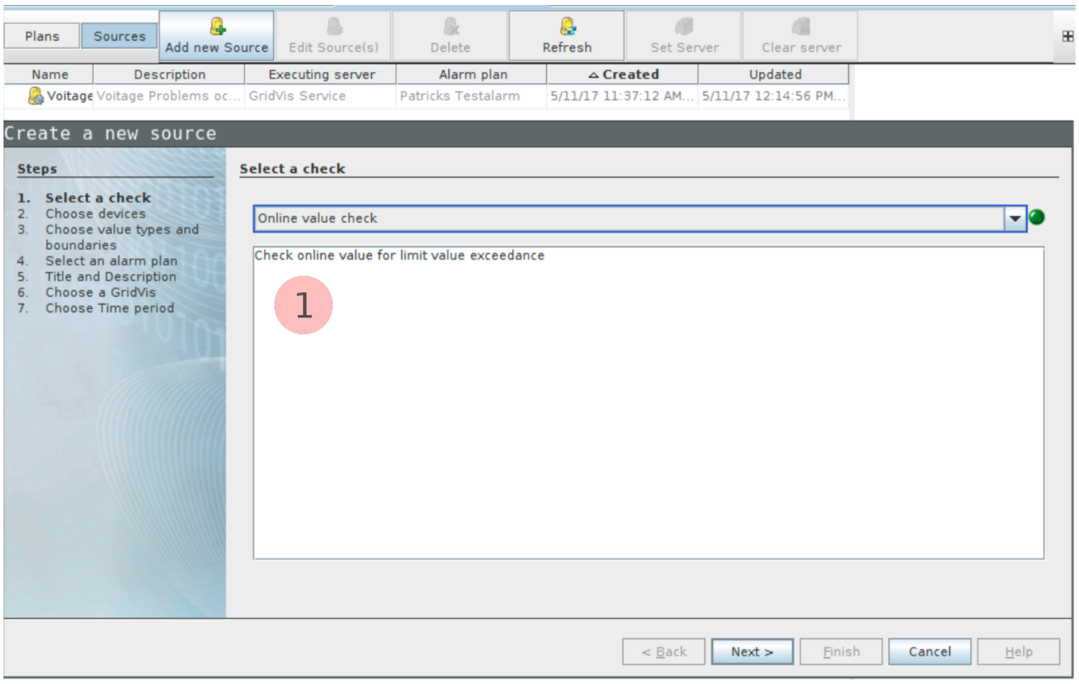
No escalation has been configured

a new alarm is raised.

Figure 3.3: Configuration wizard for alarm plans within the GridVis Desktop application.

The second alarm configuration subsection includes selecting a source that may cause alarms, see Fig. 3.4. Such an alarm source is declared as an **emitter** observing a configured system

behavior and triggering alarm events. When setting up an emitter, first of all, a specific **check** can be selected from a variety of available check implementations (1). For instance, there are checks for detecting disconnected devices or occurring transients (some sort of anomaly within the energy grid) and more.
Next, **devices** and **measurements** can be selected as a source of measurement readings. Afterwards, an existing alarm plan must be selected which the emitter belongs to.
The last three configuration steps include naming and describing the emitter, selecting an application running the check (GridVis Desktop or GridVis Service) and choosing an **alarm check time period** (weekday or weekend) defining when the emitter will act.



Name	Description	Executing server	Alarm plan	Created	Updated
Voltage	Voltage Problems oc...	GridVis Service	Patricks Testalarm	5/11/17 11:37:12 AM...	5/11/17 12:14:56 PM...

Create a new source

Steps

1. Select a check
2. Choose devices
3. Choose value types and boundaries
4. Select an alarm plan
5. Title and Description
6. Choose a GridVis
7. Choose Time period

Select a check

Online value check

Check online value for limit value exceedance

Figure 3.4: Configuration wizard for alarm sources within the GridVis Desktop application.

3.2.4 Categorizing Domain Objects within Defined Bounds

After all necessary Alarm System domain objects have been analysed, the investigation of their relations and interactions is required in order to define corresponding BCs. An overview of the analysed domain model is displayed in Fig. 3.5.

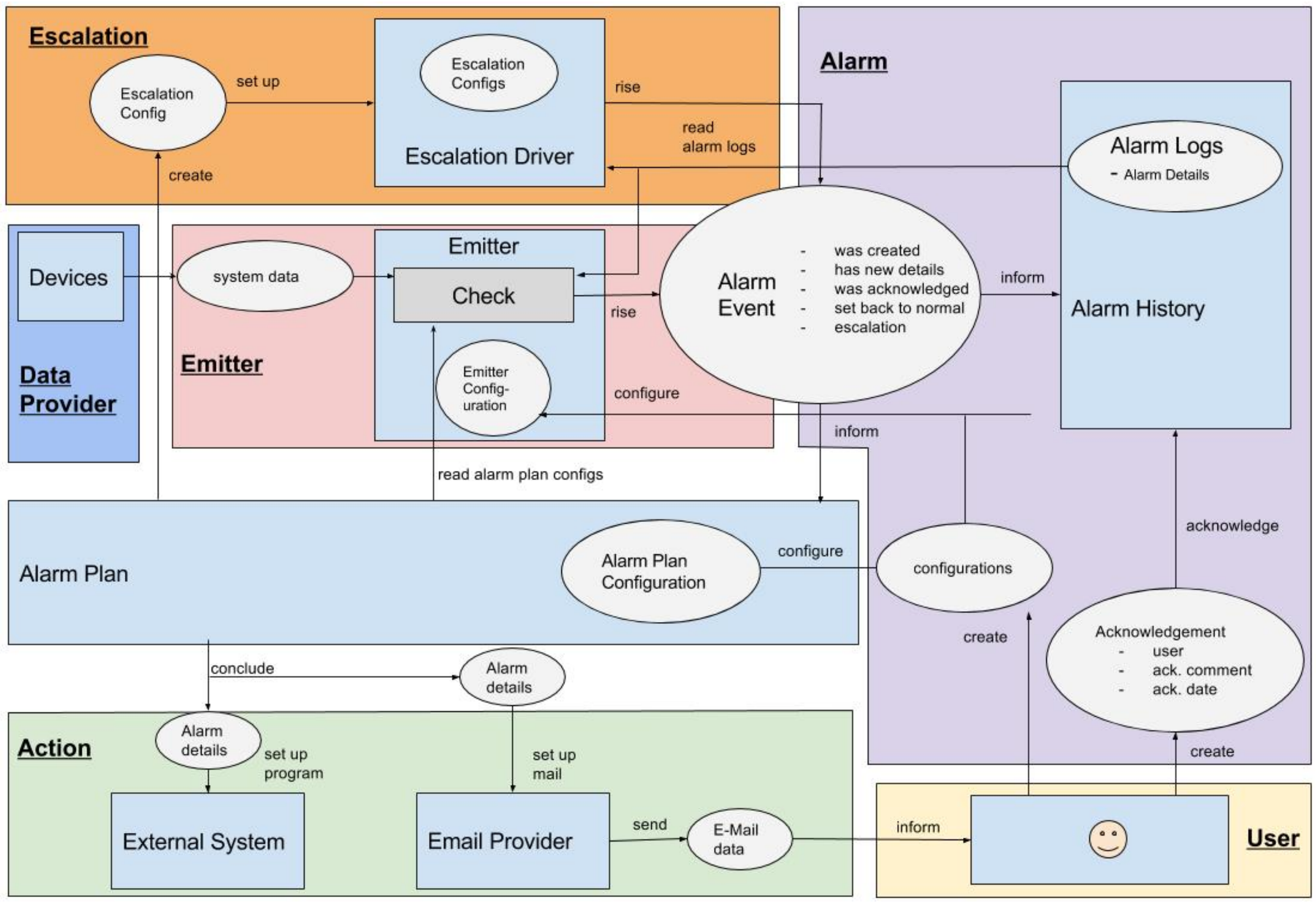


Figure 3.5: An overview of actual relations between domain objects within the GridVis Alarm System. The diagram concept is based on the Fundamental Modeling Concepts (FMC) block diagram. Basically, all angular objects are referred to as agents. An agent serves a defined purpose and communicates via channels. Round objects are referred to as storages. A storage can be either non-persistent if located on a channel or persistent if located within an agent. Channels are symbolized via arrows, whereby the arrow direction defines the data flow [17]. Please visit [9] for more details on FMC.

3.2.4.1 Relations of Domain Objects

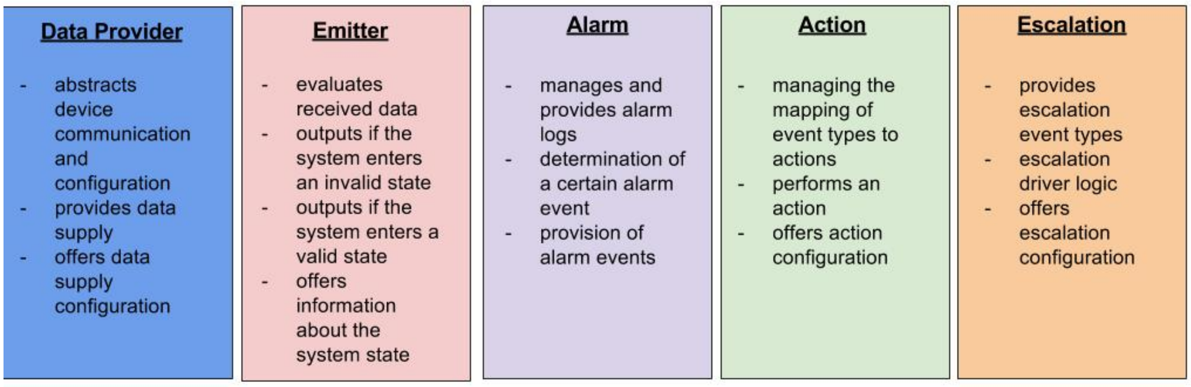
Within the system's centre there is the alarm event, which is raised by checks and the alarm driver. A check is located within an emitter instance and reads system data like measurement readings and existing alarm logs in order to create new alarm events of a specific type.
On the other hand, the escalation driver, configured with the help of the alarm plan configuration reads alarm logs, too. With the help of alarm logs, the escalation driver can determine if a new escalation level should be assigned.
An alarm plan listens for alarm events and triggers an action depending on its configuration. The user configures the alarm plan with a mapping logic. Each event type can be mapped to multiple

actions executed on the external system or via the email provider. The external system can run an independent program and the email provider can dispatch emails containing alarm details. An alarm log is created for each alarm event containing alarm details. Each alarm log is referenced by the alarm history. The user can acknowledge each alarm log individually by sending an acknowledgement containing a comment.

3.2.4.2 Deriving Bounded Contexts (BCs)

Most of the domain objects are embedded within large, named coloured boxes each representing a BC.
One of the most obvious BCs is the user context (yellow box). Of course, user management does not belong directly to the Alarm System. Within GridVis, user management is a common dependency of many components related to business logic and, therefore, it is implemented within its own suite called the *security suite*. Access to the security suite should be encapsulated within its own BC and not be considered as the first migration step of the Alarm System.
Fetching data from devices and storing it within GridVis is a complex process. A communication pool with multiple device types and varying communication protocols needs to be managed. Within the Alarm System, the requirement to configure any of these properties does not exist, for instance, device types do not have any role within our domain object model. Therefore, I have introduced a *data provider context* (dark blue box), taking care of all the processes necessary to gather available energy grid information, such as measurements and measurement readings.
Data fetched within the data provider context can be processed by an emitter. Emitter configuration and specific checks are bounded within the *emitter context* (red box), knowing which measurements are required in order to apply specific checks on them. Therefore, communication between the data provider context and the emitter context is necessary. The emitter runs its checks and publishes its results.
I have separated the emitter context from the *escalation context* (orange box). This segregation might be a discussable decision, because both the escalation driver and the check may raise an alarm event. However, the emitter itself operates as a data pipeline triggered by incoming data, which transforms measurement readings into alarm events. The escalation driver, on the other hand, works as an active component. For example, an escalation driver for scheduled escalations must decide independently for incoming data whether to increase an escalation level or not.
Both the escalation and emitter context require information from the alarm context (alarm details) in order to decide whether an alarm event should be raised or not. When implementing microservices, this dependency will lose its usefulness, since low communication efforts are essential. When defining context bounds, this is a common problem, which implies a restructuring of functional purposes.
The *alarm context* (purple box) only consists of the alarm history domain object. It stores alarm logs created from alarm events received and accepts alarm acknowledgements by users.

The only domain component without any BC is the alarm plan. The problem of classifying its bounds is based on its shared centralized position: The alarm plan contains information about the mapping of alarm events to actions used by the emitter context and simultaneously initiates the escalation driver and certain actions. This makes a clear separation very difficult. Consequently, the separation of its inner components to other BCs is necessary.



Data Provider

- abstracts device communication and configuration
- provides data supply
- offers data supply configuration

Emitter

- evaluates received data
- outputs if the system enters an invalid state
- outputs if the system enters a valid state
- offers information about the system state

Alarm

- manages and provides alarm logs
- determination of a certain alarm event
- provision of alarm events

Action

- managing the mapping of event types to actions
- performs an action
- offers action configuration

Escalation

- provides escalation event types
- escalation driver logic
- offers escalation configuration

Figure 3.6: All extracted bounded contexts of the Alarm System with their individual responsibilities.

Fig. 3.6 summarizes all extracted bounded contexts with their necessary purposes. As previously outlined, the alarm plan domain object has been broken down, and afterwards, its tasks are distributed between the remaining contexts.
The emitter's configuration has been moved to the emitter context. Then the user needs to directly set up a desired check within the emitter context that way, the mapping from check instance to data remains within its logical bounds.
Due to the requirement of accessing alarm logs as easily as possible, the determination of an alarm event has been moved to the alarm context, which needs to consume information from the emitter service.
The mapping from alarm event types to actions has been embedded into the action service, because otherwise, if triggering a certain action on an event was implemented within the alarm context, services would be coupled too closely.

3.2.5 Analysing Processes within the BCs and Resulting Algorithms

Before communication behavior can be set up, determining service-specific data demands will be necessary. It should be clear *when which* service needs *which* data. Therefore, relevant processes within each service should be investigated.

3.2.5.1 Emitter Service

At the heart of alarm processing is the emitter with its specific checks. The check component implies which data is required by the data provider. This could be information about a certain device used to validate of the device's connectivity or its measurement readings.
In order to get a full-stack example of a running Alarm System implementation, I will focus on a single check type only: the *online value check*, see Fig. 3.8. The online value check is a straight-forward validation processing one online value at a time and comparing it to defined boundaries. Therefore, the system's state must be considered in order to decide whether the system behavior has to return to normal or not. An exceeded or unmet boundary implies a system misbehavior, which can be propagated among other services.

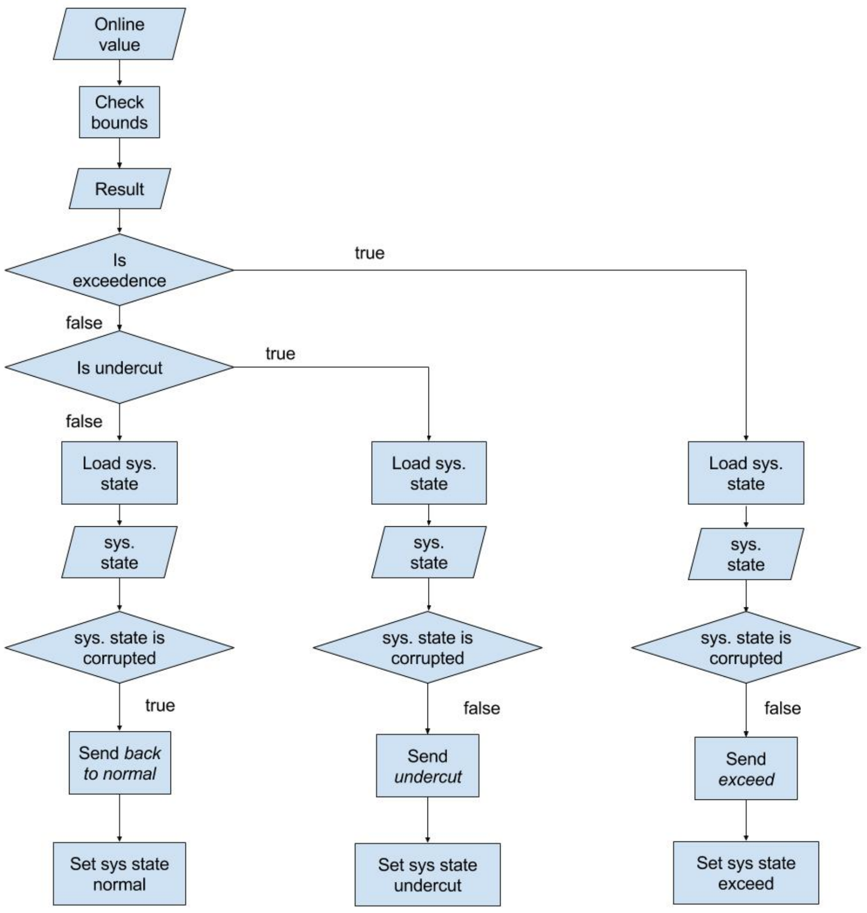


Figure 3.7: Flowchart of the online value check algorithm.

3.2.5.2 Data Provider

In order to fetch requested data from a device, the data provider needs to know which measurement readings are required. As soon as data will not be required any longer, the data provider

should be informed to save resources.
Measurement readings are categorized by a measurement and a device. Technically, device references (device IDs) and measurements are grouped within a *project*. A GridVis project is just identified by its name, but within a microservice implementation, it could also be implemented with sort of Universally Unique IDentifier (UUID). Certain measurements within a project are modelled as so called *IValues*. An IValue can represent a single measurement with no context or a specific context containing a project and/or a device.
Within this thesis, an IValue models the affiliation of a measurement and a device reference within a single project. A textual represented IValue is called IValueID.

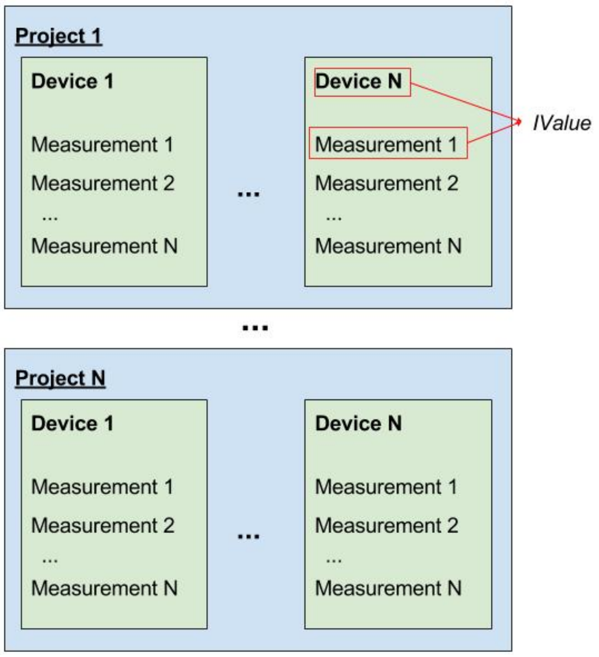


Figure 3.8: The hierarchy of projects, devices and measurements within GridVis.

In order to fetch data from devices, functionalities from GridVis *value suite* and *devices suite* are required, which are not considered within the migration of the Alarm System. Therefore, the data provider context portrays an anti-corruption layer, which decouples legacy GridVis with its value suite from the migrated Alarm System.

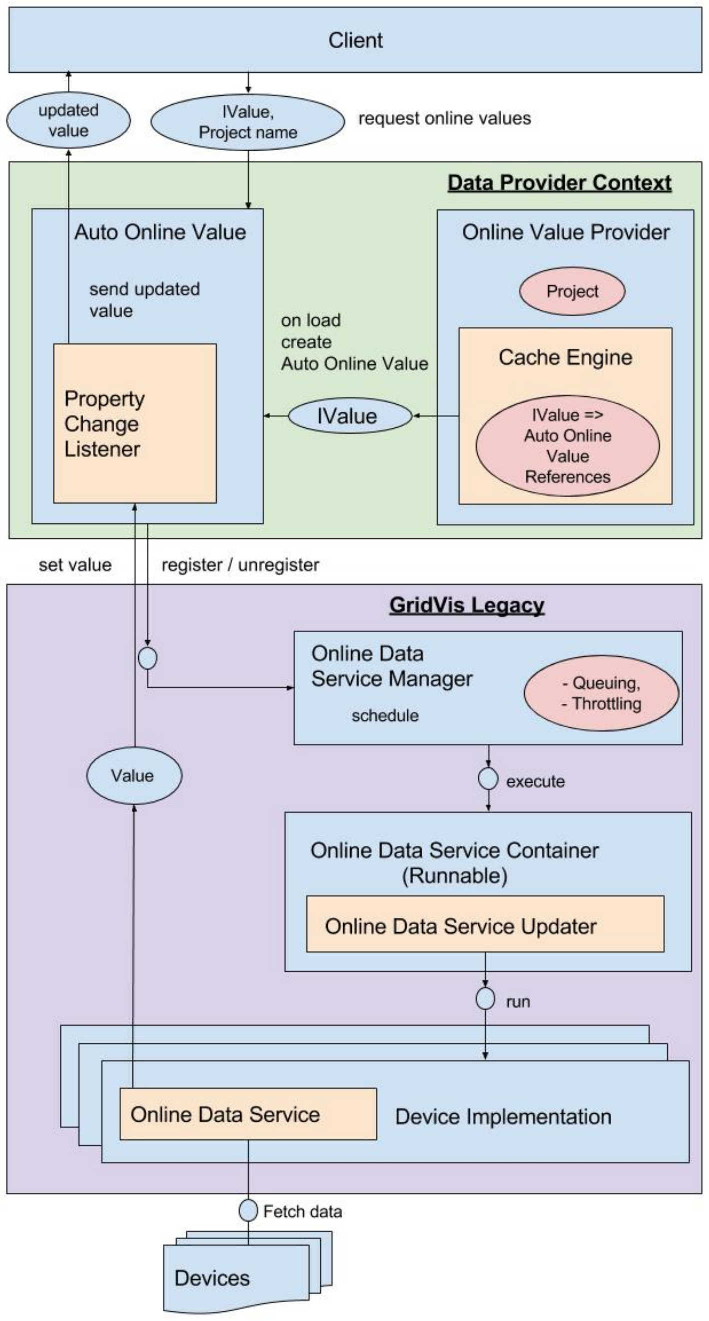


Figure 3.9: Components involved in the online value fetching process.

Fig. 3.9 outlines the components involved during the online data fetching process. Data fetching is initiated by the client. He requests online values with the demanded project name as well as the measurement and device reference embedded within an IValue. Per project, GridVis runs a so-called online value provider. The online value provider implements the NetBeans provider interface and starts when its project is mounted.

In order to join multiple requests that demand the same IValue, the online value provider holds a cache engine that contains a mapping from an IValue to an auto online value.
Auto online values are self-refreshing online value containers. They hold the last updated online value and are refreshed periodically while devices are fetched. Each auto online value loads a device-specific online data service which is embedded within the specific device implementation. The auto online value registers a property change listener on its online data service that updates the last known online value.
Since device communication is an expensive resource, it has to be utilized very efficiently. Communication may not be applied often and an optimal data amount should be fetched when it is. The online data service manager was designed for this purpose. The auto online value is registered with the online data service manager in order to ensure the online data service is executed properly.
The online data service manager is widely used within GridVis. It has the capability to queue up and throttle multiple online data service executions within a designated thread, called online data service containers. When executed, the online data service container runs the online data service, which is finally communicating with the specific device type. After registering at the online data service manager, data will be continuously fetched until a deregistration is applied.
After communication between the online data service and the device is complete, the online data service's state is updated and a change event is triggered. An updated value is set up within the auto online value. Within the figure, I have indicated that the value is sent to the client afterwards, but it could also be accessed more passively, e.g., due to a REST call.

3.2.5.3 Alarm Service

The alarm service consumes emitter updates, containing information about the system state, in order to log new alarms to disc. Every time a new alarm log has been created, the alarm service needs to propagate an event describing the recent alarm log, for example, when a new alarm has been created or when it was set back to normal.
Each alarm log must contain a reference to a project, an IValue, a time-stamp describing its last change time, a time-stamp describing when the alarm was created, some acknowledgement information and the alarm's state.

3.2.5.4 Action Service

The action service takes care of the mapping from an alarm event to a corresponding action. Therefore, it needs to consume updates from the alarm service.
Within GridVis, the action service can execute a program or send an email. In order to implement a full-stack prototype, I have decided to focus on the email action only.
The configuration of the email action service must contain a mapping that includes which project,

device and measurement combination should trigger an email send action to whom on which alarm event type.

3.2.5.5 Escalation Service

In order to start a new escalation process, the escalation service needs to receive alarm events. The alarm event must be evaluated to decide whether to save a new escalation or remove an already existing escalation.
Within GridVis, the current implementation was directly coupled with the alarm detail properties of the alarm plan context. These properties contained information about when an alarm was created and recently updated. Therefore, they could be used to determine a scheduled escalation. Now, the escalation context is not able to access this information directly and its time information must persist within its own BC. Therefore, the algorithm needs adjustment.
Unlike the emitter, action or alarm service, the escalation service is an active component. On start up, the escalation service runs a concurrent task that periodically loads scheduled escalations, which were created before a configured point in time. In Fig. 3.10, I called these escalation entries *hot escalations*. Afterwards, the action service should be informed for each loaded hot escalation.

Example: The user configures an escalation after t_d, where $t_d \in \mathbb{N}$ and $t_d > 0$, representing a configured duration, which defines when an alarm rises in urgency.
Afterwards, the escalation service receives a new alarm event of type *alarm created* with the timestamp of t_c. A new scheduled escalation is saved if it is not already existing.
An escalation is called a *hot escalation* when $t_c + t_d \le t_a$, where t_a is the actual time stamp.

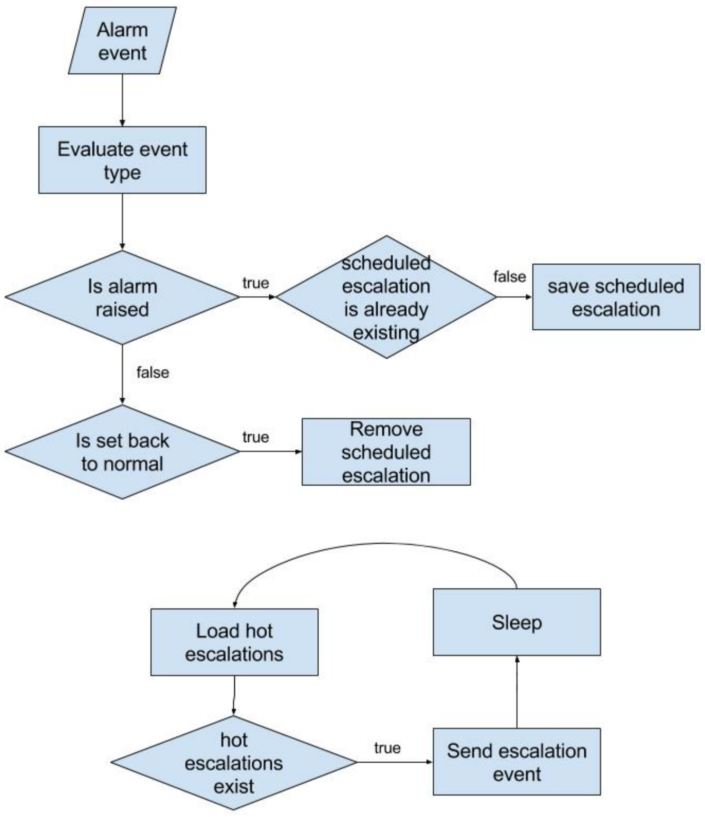


Figure 3.10: The main algorithm of the scheduled escalation service.

3.2.5.6 Communication Behavior

Within the microservice design process, we should consider how collaboration between services should be applied. The comparison between asynchronous and synchronous communication styles was discussed in chapter 2.1.2.1. Both have their benefits and drawbacks, but within this prototype I have decided to focus primarily on asynchronous communication.
The inter-service communication should be efficient, decoupled, fast and as fault-tolerant as possible. Services should react quickly on new information; therefore a blocking service that awaits an answer from another service, which possibly needs multiple hops within our microservice architecture, will potentially not satisfy our performance requirements.
The reactive behavior was considered within the previously outlined chapters, describing processes within the new microservice components.
When it comes to configuring services, especially Create, Read, Update, Delete (CRUD), I tend to deploy a mixture of asynchronous and synchronous approaches. As outlined in more detail in chapter 3.3.6.2, a technology like REST over HTTP fits pretty well for configuration.
A client can send a full universal format like JSON to configure a service. The underlying HTTP protocol keeps the connection between the client and the server alive, this means the protocol

itself stays synchronous. On the other side, services and clients act in an asynchronous manner while processing requests and responses.
For example, when using ECMAScript, the language specification does not support any concurrency mechanisms. Instead, it makes use of asynchronous callbacks that return when the requested REST interface answers. As long as a response is outstanding, the user might activate other callbacks, e.g., on a click event.
On the server side, the REST request is handled within its own thread pool, which executes the rest call when scheduled. In the meantime, other relevant tasks can be handled independently.
In the context of configuration, the REST call should not trigger subsequent hidden nested network calls to other services. The bottom line is that dangerous performance threats behind HTTP-based REST calls are restricted. This approach is based on the fact that the configuration of a service may only affect this one service.

3.2.5.7 Overview

Fig. 4.6 summarizes the possible communication behavior of potential microservice implementations. As we can see, the architecture is mainly event-driven. The communication behavior is quite similar to *pipes and filters*: *Online values* are transformed and filtered by the emitter service into *emitter updates*. The alarm service transforms them into *alarm events*, holds the alarm's state and forwards its alarm events to the action and escalation services.
It is pleasing that side effects and communication complexities can be reduced to a minimum, whether the escalation service has a strong dependency on its persisted state or not.
In addition, it is quite easy to extend the system by adding more consumers without the need to change producers.
All services provide a REST interface where CRUD can be applied on configuration entities, like the action service's IValue: an alarm event mapping to an email recipient. This design decouples the whole microservice architecture from its clients.

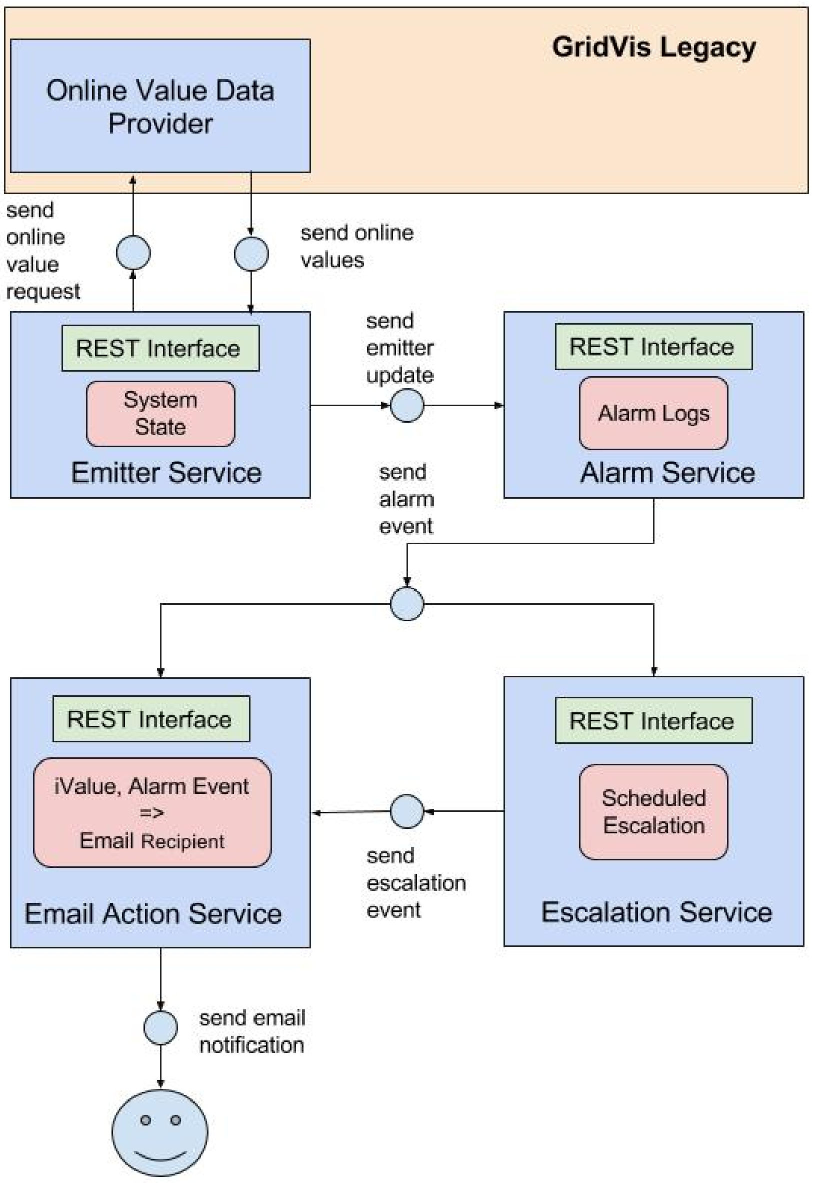


Figure 3.11: An overview of the required communication between alarm service microservices.

3.3 Finding Adequate Technologies

3.3.1 Technology Demands

To ensure that each microservice maintains its own state, some kind of persistence technology is required. State is shared only via communication. Therefore, communication should be efficient and the system should not slow down at a high data throughput. Simultaneously, messages must not get lost, because otherwise the system may fall into an inconsistent state.
When it comes to client-side configuration of services, HTTP-based REST offers a universal API,

which requires some sort of web server that provides the CRUD of configuration objects.
In order to run the system with location transparency, services should fetch information from an accessible Service Discovery that offers service addresses.
The scaling factor is one major advantage of a cloud integration. Therefore, I apply x-, y-, and z-axis scaling for the system. When x-axis scaling is applied and multiple instances of a single service type are running, communication should be load-balanced somehow to enable full efficiency. On top of this, when the amount of data increases, data partitioning must be considered as well.
Since deployment environments vary, technical configuration, such as setting up database connections, ports or messaging parameters should be changeable and flexibly applied on multiple platforms easily.
Another important aspect is adding a decent amount of monitoring capabilities to the system. Since I will not focus on the DevOps sector, a basic tracing mechanism is adequate, within which comprehensive log management is desirable.
Finally, a user interface needs to be integrated somehow, which allows for the configuration of the Alarm System.

3.3.2 Shared Technology Stack

Since all of the planned microservices need the above-mentioned technologies in one way or another, I planned to install a certain technology stack (s. fig 3.12) that can be adjusted on demand.
Although microservices can be implemented in any programming language, I have decided to implement the Alarm System related microservices in Java. This is primarily because all team members in the project-related software department are-well experienced in Java. In addition, the JVM makes Java artefacts easy to deploy and, in the best case, the extraction of already implemented algorithms can be applied without touching them or their tests.
The aim of a common technology stack is to provide faster development of new microservices, whereby teams may decide whether they use the given technology stack or if they go their own way on implementation.
For example, a team could replace a certain technology, like the database integration or choose a completely new programming language. Such a choice may be more work intense than using an already existing technology stack at first, but could pay back after time.

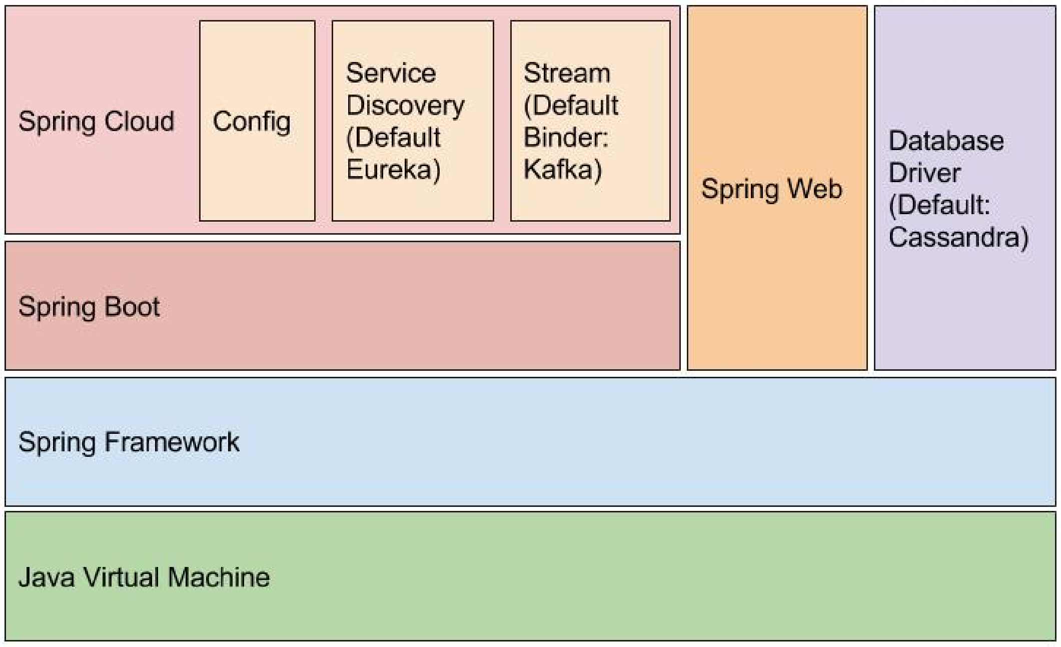

Spring Cloud
Config
Service Discovery (Default Eureka)
Stream (Default Binder: Kafka)
Spring Web
Database Driver (Default: Cassandra)
Spring Boot
Spring Framework
Java Virtual Machine

Figure 3.12: An overview of the common technology stack intended for the new Alarm System's microservice implementations. Each implemented microservice utilizes this technology stack.

The Spring Framework, with Spring Boot and Spring Cloud, opens a huge variety of different technologies that help to implement microservices in an efficient manner.
Spring Framework and Spring Boot take care of dependency injection, code structuring and technology integration. Spring Cloud, on the other hand, enables all essential capabilities to handle distributed system problems, such as configuration, Service Discovery, tracing, load balancing and messaging. Spring Web which builds upon the Spring Framework, enables web server capabilities with an easy set up for RESTful services. Finally, a default database driver that is well integrated with the Spring Framework is part of the technology stack and helps with accessing persistence quickly and easily.

3.3.2.1 Example of Integrating Service Discovery with the Help of Spring Cloud

Within this example, I want to demonstrate one key benefit of Spring Cloud: The easy integration of distributed systems technologies.
In order to enable the registration of the alarm service to a Eureka discovery server, a single annotation within the main class is required.

```java
@SpringBootApplication // Register application as Spring Boot application
@EnableEurekaClient  // Enable registration to Eureka Server
public class AlarmServiceApplication {
    public static void main(String[] args) throws IOException {
        SpringApplication.run(AlarmServiceApplication.class, args);
    }
}
```

Address and communication behavior settings are extracted automatically from the corresponding application file. The service registers on start up and sends heartbeat messages automatically.

3.3.3 Service Discovery

Spring Cloud in fact supports two different main Service Discoveries: Netflix's *Eureka* and HashiCorp's *Consul*. Both are well-proven and work purposefully.
Within the following implementation, I have chosen Eureka as my default discovery service, since it is very modest in terms of resources. For developing purposes, I can run the eureka server as a single instance instead of setting up a complete consul cluster with at least five nodes and its referring configuration.
Consul has many features, including DNS and Hypertext Transfer Protocol Secure (HTTPS) support, but in the current situation my system does not require these features. The main differences are outlined in more detail in [30].
In summary, I will not discard Consul as a Service Discovery, since it is very powerful and well-proven on Amazon Web Services (AWS) cloud environments. But for this prototype, the project size and the ease of setting up a single Eureka instance was convincing.
Please note that aside from setting up the discovery service itself, a migration from eureka to consul is quite reasonable.

3.3.4 Configuration

In order to manage the system's configuration, I have decided to utilize *Spring Cloud Config*, which provides client-side support for external configurations. This technology enables one to change environment-specific variables like addresses dynamically when running through a CI pipeline.
The configuration server stores information for each service individually: application name, which is required for determining service types (e. g. emitter service, alarm service, etc.); active profiles (e. g. development, test, deployment cloud 1, deployment cloud 2, etc.); messaging configurations; database connections, message brokers; and eureka addresses.

3.3.5 Monitoring: Logging and Tracing

Monitoring is a wide field and becomes very important under production.
I have tried to integrate some tracing within the presented technology stack in order to evaluate necessary integration efforts at application level.
Spring Cloud Sleuth is based on the ideas behind Dapper and Zipkin. It captures data simply in logs or by sending it to a remote collector service [14].
To integrate Sleuth into a system, the only thing necessary is to put it into the classpath, e.g., via Maven dependency management. When logging, a trace ID and span ID are added to the logs. These can be used to recreate traces over multiple hops within the distributed system without the need of a global clock [14].
This set up could easily be extended, e.g. by adding Elasticsearch, Logstash and Kibana (ELK), a combination of tools helping to search, analyse and visualize log data.
Since this work does not focus on deployment and DevOps, a host system oriented monitoring approach is not considered.

3.3.6 Communication

3.3.6.1 Messaging

The use of a messaging semantic, such as publish/subscribe, can be easily utilized. But setting up the messaging environment is not trivial: Multiple instances of a single service run in parallel, which implies some sort of partitioning. In addition, message brokers must be accessed, whether their location changes or not. Furthermore, after a certain message broker is installed and committed, its exchange of it may become more work-intensive.
Spring Cloud Stream is a framework for building message-driven microservices. It provides an external configurable abstraction layer from a specific message broker via a binder abstraction. Within the external configuration channels, partitions are set up for each microservice independently. This provides decoupled, changeable and flexible services.
Spring Cloud Stream supports actually two main message brokers: Apache's *Kafka* and *RabbitMQ*. Nevertheless, a developer is still capable of writing his own message binder for specific purposes.
Since the Alarm System must deal with high loads of periodic measurement readings, I have decided to use Kafka, because it promises a higher data throughput than RabbitMQ, which enables better long-term scaling [22].

3.3.6.2 Client-side Restful Configuration

I have decided that each microservice should be configured via an HTTP-based REST interface offering CRUD. REST can be easily used by any external client, whether it is a desktop application client or a web client. Therefore, every service should offer the possibility to create, read, update

and delete configuration objects. The necessary configuration objects are presented in more detail in chapter 4.1.2.
Spring Boot is able to embed its own web server that is capable of handling HTTP-based REST commands. When Spring Web is included within the classpath, it offers annotation-based configuration, which enables everything necessary to handle REST requests within a REST controller.
Basically, a REST method only needs an *@RequestMapping* annotation to be accessible via HTTP.

```
@RequestMapping(method = RequestMethod.GET,
    path="/action/email/read/{emitterId}")
```

The annotation *@RequestMapping* is crucial, since it configures the servlet to accept *HTTP GET* methods at a given Uniform Resource Identifier (URI) path, called */action/email/read/{emitterId}*, where *{emitterId}* must be a valid UUID.

3.3.7 Persistence

In order to keep the bounded context of a service clear, each microservice should possess its own persistence management for its entities [39, pp. 187 f.]. This approach enables the possibility of choosing a specific database technology for each service individually. To run a resilient fault tolerant system, redundant instances of a single service type run simultaneously. But what happens if a single database instance is unavailable somehow?
Since a database system might be a common bottleneck in terms of availability, I have decided to run a distributed database. Basically, there are two options: The first option is running a cluster of relational databases, such as multiple MySQL instances with a master-slave approach. The second option consists of the use of a more complex, more unfamiliar distributed non-relational database, such as *Apache Cassandra*.
The use of relational data bases within this project is promising, mostly because they are well-understood and a Relational Data Base Management System (RDBMS) offers great consistency guarantees. Nonetheless, I have decided to give Apache Cassandra a try, because it scales way better than a MySQL cluster or equivalent in a cloud native-web environment (linear with nodes) [23, p. 12].
Cassandra offers basic support for data partitioning (z-axis scaling) and its tunable consistency may be sufficient. The downside of this decision is the large adjustment of data schemata. A data base replacement of microservices using Cassandra towards a relational data-base at a later stage will be relatively work-intensive.

3.3.8 Load Balancing

Within the presented technology stack, load-balancing is considered in two technology components: *Kafka's partitioning* and *Eureka's internal load balancer*.

3.3.8.1 Kafka's Partitioning

Kafka messages can be broadcast or divided between service instances using so-called *consumer groups* [20]. Kafka consumer groups can be set up using Spring's external configuration, which defines group names for each service. By defining no group name, a message is broadcast to all instances listening on a given topic.
X-axis scaling can be enabled by defining a group name for a service type. Therefore, a set of multiple service instances of a given service type consume a single message instead of all instances consuming that message. This way, z-axis scaling can be enabled, too. For example, this can be done by setting up a group name that contains the service type as a prefix and a responsible data partition as a suffix. The following example would define a group of microservices of type *emitter service* that is responsible for the data partition of a project called *project-one*.

```
//{prefix}###{suffix} => {service type}###{data partition}
emitter-service###project-one
```

3.3.8.2 Eureka's Load Balancer

Eureka has a built-in load balancer that I mainly intended to use for processing REST calls, since messaging is already load-balanced by Kafka.
Clients fetch service addresses from Eureka, which are load-balanced (e.g., via round robin), before applying a CRUD operation on a service configuration.
Hint: REST access would work best with an API gateway like Netflix's *Zuul*, a reverser proxy that keeps API structure clear and helps with routing.

3.3.9 User Interface

The user needs to configure the system somehow, and the display of system's information must be supplied. The software architecture of a UI within a distributed system is a very large and non-trivial problem space.
There are many possible solutions, each bringing its own advantages and disadvantages to the table. For example, there is one approach in which each is linked to its own website and all sites are linked up to one another. This approach restricts the structuring of the UI and affects the look and feel of the system as a whole. Another approach would be the aggregation of multiple website content elements, such as forms and *widgets*. This implies a complex aggregation layer that handles a scaling amount of possibly unavailable services. It must extract the services UI components and combine all of them into a purposeful web UI dynamically.
Within the presented system, I have set up an independent single page application web client, that aggregates the REST interfaces of all referenced microservices and provides a centralized user interface. Unfortunately, a centralized UI service that accesses each service's REST API

individually couples the UI and the referring microservices very closely, since an API change within a REST interface of a single service affects the UI service as well. Basically, I do not recommend this approach for production, since it will slow down software delivery in the long term.

Nevertheless, in order to commit to a certain user interface architecture, each stakeholder should plan around their customers' requirements first before software design and implementation can be applied purposefully, because its architecture will be the primary driver behind the UI's look and feel within a microservice landscape.

Chapter 4

Presentation and Evaluation

4.1 Microservice-based Architecture of the Alarm System

4.1.1 Overview

Within this chapter, I will introduce an example of a microservice system whose features are extracted from a comprehensive existing monolithic application. Things that were necessary in order to begin with a reasonable implementation are outlined in chapter 3.

The implemented system consists of five core services, which contain business logic, and three auxiliary services with an infrastructural purpose. The system is capable of all rudimentary Alarm System features supported by GridVis. This includes: online value access, emitter configuration, alarm logging, scheduled alarm escalations and email notifications on a certain event type.

The system possesses the many advantages of a distributed system while reducing typical distributed system problems by utilizing *Spring Cloud* libraries.

Its architecture and underlying technology stack was designed to develop new microservices faster and more efficiently. With the help of message-driven communication, each service was designed as a reactive, compact and easily understandable unit that can be replaced easily without a larger impact to the rest of the system. Additionally, the integration of a service into some sort of CI cycle, as well as its further deployment, was designed to be possible without a change to the source code of services' inner functionalities.

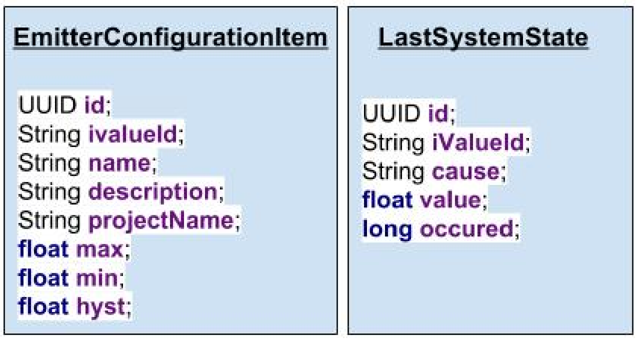
4.1.2 Data Model and Configuration

4.1.2.1 Concept

As has already been mentioned in the previous chapters, each service must only manage and influence its own data in order to ensure the implementation of easy understandable, well-maintainable service. This implies a strict separation of data by its corresponding functionality and requires communication between services. In order to determine data responsibilities, the system was designed with the help of BCs, as described in the section about DDD (see chapter 3.2.2).
Algorithms within services are generally described in chapter 3.2.5, while this chapter focuses on the resulting data model.

4.1.2.2 Online Value Emitter Service

The user can configure the online value emitter service in order to define when the system state is corrupted. Within the implementation, this configuration is modelled as *EmitterConfigurationItem*.
A lower and an upper bound (*min, max*) as well as a hysteresis (*hyst*) define a valid system state. The observed measurement is identified through a project name and an IValue id. Each emitter can be described by setting a name and a description.



```
EmitterConfigurationItem

UUID id;
String ivalueId;
String name;
String description;
String projectName;
float max;
float min;
float hyst;
```

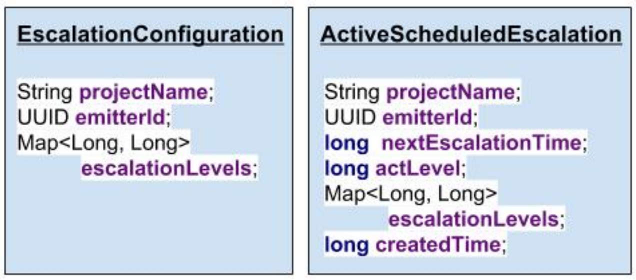
```
LastSystemState

UUID id;
String iValueId;
String cause;
float value;
long occured;
```

Figure 4.1: Entities used within the online value emitter service implementation.

The *LastSystemState*, identified by a UUID, models the last known state of the observed system's behavior. It is loaded iteratively by the online value emitter service in order to determine if an *OnlineValueEmitterUpdate* event, described in chapter 4.1.2.8, needs to be published. The *LastSystemState* includes information about the *cause*, describing the actual system state, an online value measurement reading referenced as *value*, and a timestamp, describing when the system state was detected.

4.1.2.3 Scheduled Escalation Service

The configuration of escalations is modeled as *EscalationConfiguration*. To set up scheduled escalations, the user needs to define a number of individual escalation levels. Each level references a time value representing the duration it takes to enter the relevant level. This configuration is modeled as a map collection (*escalationLevels*). Important to this process is an assignment to an existing emitter (*emitterId*) in order to determine whether an alarm event is relevant or not. The *ActiveScheduledEscalation* must contain additional information, such as the actual escalation level (*actLevel*) and the time it was created. The creation time is very important for making time ranged queries for hot escalations.
Since the scheduled escalation service operates actively, it needs to query the data base iteratively after it starts. In order to avoid querying to much data and enabling z-axis scaling, data partitioning is required. In fact, data partitioning is realized by using partitioning per project. Therefore, the escalation configuration must contain a project name.



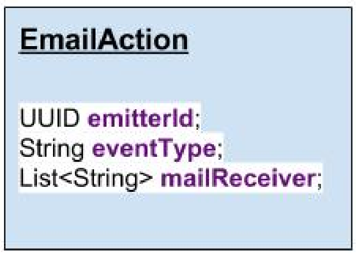
```
EscalationConfiguration

String projectName;
UUID emitterId;
Map<Long, Long>
        escalationLevels;
```

```
ActiveScheduledEscalation

String projectName;
UUID emitterId;
long  nextEscalationTime;
long actLevel;
Map<Long, Long>
        escalationLevels;
long createdTime;
```

Figure 4.2: Entities used within the scheduled escalation service implementation.

4.1.2.4 Email Action Service

Email-sending actions and the corresponding configuration are supported by the email action service. The user can define which alarm event, represented as a string, triggers the dispatch of an email to a list of recipients.
Like the scheduled escalation service, the action service needs the emitter ID within its configuration to determine whether an alarm event is relevant or not.

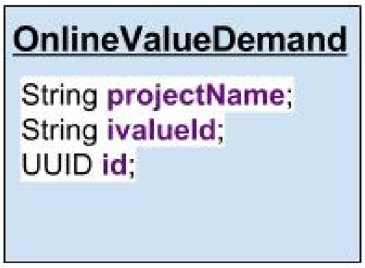


```
EmailAction

UUID emitterId;
String eventType;
List<String> mailReceiver;
```

Figure 4.3: Entities used within the email action service implementation.

4.1.2.5 Online Value Data Provider Service

On the other hand, the data provider with the underlying GridVis legacy system is configured passively via a notification message sent from emitter services. This is a crucial design decision, which is based on the intention of the service: the data provider itself works most efficiently in a passive way, acting only on demand. This is because if the data provider service had been an active service, that is configured actively by a user and pushing online values continuously, then the system would have ran into an inefficient state.
For instance, this could result in the processing of expensive queries to devices with potentially no consumers active. Therefore, the *OnlineValueDemand* models a registered demand on a specific measurement, from a device within a certain project.

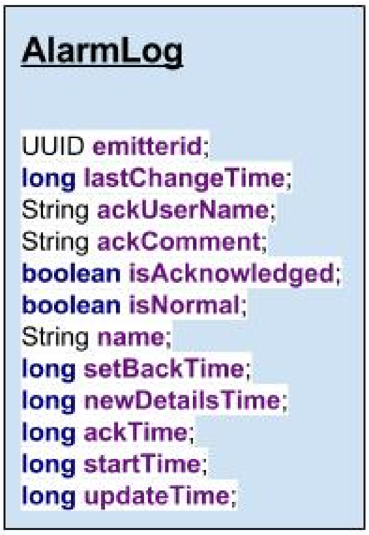


```
OnlineValueDemand
String projectName;
String ivalueId;
UUID id;
```

Figure 4.4: Entities used within the online data provider service.

4.1.2.6 Alarm Service

Actually, the alarm service itself does not need any configuration, since it works as a pure data transformer, that consumes messages from emitter services and outputs the corresponding alarm events.
Alarm logs are a relatively large entity containing much information about alarms. Acknowledgement information is included and contains whether the alarm was acknowledged or not, the user's name, who acknowledged the alarm and the corresponding acknowledgement comment. Furthermore, time information is saved, such as the time stamp when the alarm was changed most recently (no matter the change type), the time stamp when the alarm was set back to normal, the time stamp when the alarm was acknowledged, the time stamp when the alarm was initiated and the time stamp when the alarm cause has changed (*updateTime*). Each alarm is described by a dynamically generated alarm name. Additional flags define the alarm's state: whether it was set back to normal or not and whether it is actually normal or not.
Since multiple alarm logs can be generated by one emitter, each *AlarmLog* needs a reference to the emitter it belongs to (*emitterId*).



```
AlarmLog

UUID emitterid;
long lastChangeTime;
String ackUserName;
String ackComment;
boolean isAcknowledged;
boolean isNormal;
String name;
long setBackTime;
long newDetailsTime;
long ackTime;
long startTime;
long updateTime;
```

Figure 4.5: Entities used within the alarm service.

4.1.2.7 Messaging Integration

Inter-service communication is implemented via Apache Kafka's message streams. The integration of Kafka is completely abstracted by *Spring Cloud Streams*. The only exception is the

communication of the data provider service and the emitter service. These two services make both use of the *kafka-clients* API, which is the native independent API of Apache Kafka.
In order to connect two nodes via *Spring Cloud Stream*, message-sending services need to bind an output channel called *Source* and an injected *MessageChannel* object. Message-receiving services need to bind an input channel called *Sink*. Both Source and Sink are modelled as a Java interface that abstracts a specific binder (Kafka or RabbitMQ).
The following code snippet is extracted from the online value emitter service. It contains the sender class of *OnlineValueEmitterUpdateEvents*.

```
// Binding of a Source with a default channel name called output
@EnableBinding(Source.class)
public class OnlineValueEmitterUpdateEventSender {
    @Autowired // Channel passed in via dependency injection
    private MessageChannel out;
    public void publishEmitterUpdateEvent(final OnlineValueEmitterUpdateEvent
        update){
        out.send(MessageBuilder.withPayload(update).build());
    }
}
```

The next code snippet is extracted from the alarm service. It contains the receiver class for *OnlineValueEmitterUpdateEvents*.

```
// Binding of a Sink with a default channel name called input
@EnableBinding(Sink.class)
public class OnlineValueEmitterUpdateEventReceiver {
    @Autowired
    private OnlineValueEmitterUpdateEventHandler handler;
    // Annotation indicates that a method is capable of handling a message or
        message payload.
    @ServiceActivator(inputChannel=Sink.INPUT)
    public void receive(final OnlineValueEmitterUpdateEvent update) {
        handler.handleUpdate(update);
    }
}
```

The destination of incoming or outgoing messages can be set up via Spring's environment (e. g. the application file), where channel names (e. g. *input* / *output*) are mapped to a certain topic name.

```
// application.properties file: configuration of the online value emitter service
// the channel named output is mapped to the topic called emitterupdate
spring.cloud.stream.bindings.output.destination = emitterupdate

// application.properties file: configuration of alarm service
// the channel named input is mapped to the topic called emitterupdate
spring.cloud.stream.bindings.input.destination = emitterupdate
```

4.1.2.8 Message Formats

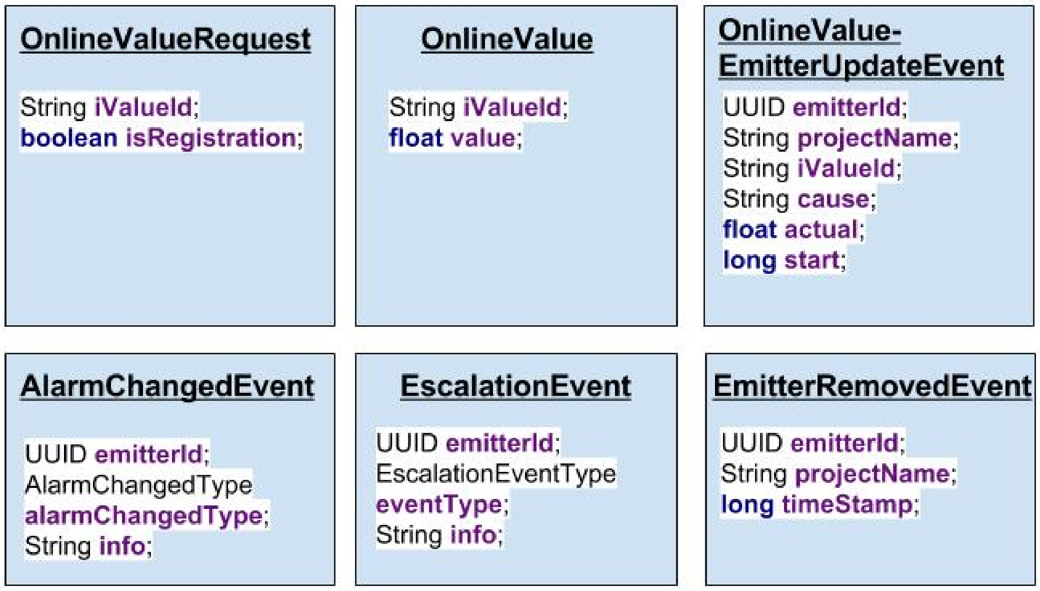


Figure 4.6: All communicated message objects between implemented microservices of the Alarm System.

The system's communication is initiated by the emitter service. In order to receive *OnlineValues*, an *OnlineValueRequest* needs to be sent first. The *OnlineValueRequest* contains information that defines which measurement is requested from which device (*IValueID*) and whether it is a registration or a deregistration (*isRegistration*).
The emitter service consumes *OnlineValues* and creates *OnlineValueEmitterUpdateEvents*. An online value object consists of the relevant *IValueID* and the last recent measurement reading called *value*.
After online values are consumed, *OnlineValueEmitterUpdateEvents* are produced and published.

This message format contains information about the actual system state: The corresponding emitter ID, the project name and the IValue ID of the measurement reading, the cause of the emitter update, the most recent online value and the timestamp when the emitter update was detected.
The only consumer of *OnlineValueEmitterUpdateEvents* is the alarm service, which processes each *OnlineValueEmitterUpdateEvent* and outputs the most recent *AlarmChangedEvent* of the corresponding emitter ID. The *AlarmChangedEvent* message contains the referenced emitter id, a certain alarm changed type (an enumeration of: CREATED, ACKNOWLEDGED, NEW_DETAILS, SET_BACK_TO_NORMAL) and a textual description of the system's misbehavior, called *info* (e. g. underflow, exceeding, etc.)
The scheduled escalation service consumes the *AlarmChangedEvent* in order to start or stop its scheduled escalation. Furthermore, it outputs *EscalationEvents* after a configured duration has been exceeded. The *EscalationEvent* contains a reference to an emitter ID as well as information describing the escalation. The enumeration, *eventType*, contains a pattern, describing the escalation level,e.g., *SCHEDULED_ESCALATION_1* represents a scheduled escalation of escalation level one. The *info* property is a textual representation of the actual escalation.
The email action service listens for *AlarmChangedEvents* and *EscalationEvents* in order to dispatch emails.
One IValue and project name mapping may belong to multiple emitters, since a variety of emitter types could exist within the system. Each emitter represents a single functional unit with a transitive meaning in the complete system. For example, the escalation event has an affiliation to an existing emitter. Without an emitter, any escalation would become meaningless.

4.1.3 Communication Scenarios

4.1.3.1 Example: Communication Behavior of a Configured Online Value Emitter Service and Email Action Service

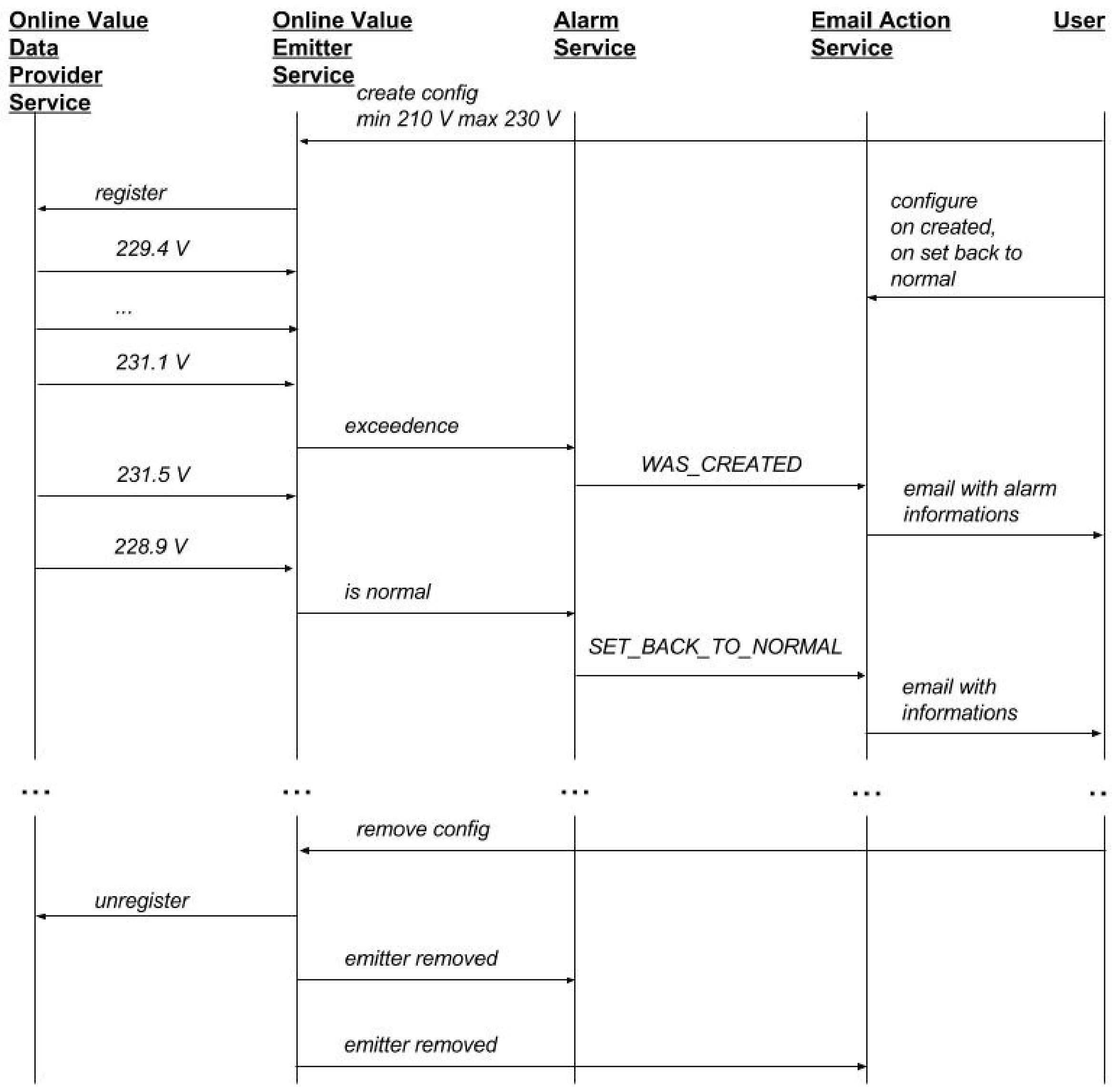


Figure 4.7: Example sequence diagram of a possible communication scenario of a configured online value emitter service and email action service within the system.

The example in Fig. 4.7 outlines the classical system behavior when an online value emitter is configured in combination with an email action service. I have skipped the escalation service, since communication would be applied analogously as described in this example.
First, the user defines an emitter configuration item with the corresponding bounds, measurement

and project device association, e.g., voltage effective and device ID 1 combined as an IValue ID, JanitzaServerProject, minimum 210 V, maximum 230 V, with a hysteresis of 0.
Now, the online value emitter service registers itself at the online data value provider using the project name and the IValue ID. After fetching device data, the online value data provider service starts sending online values while continuously fetching devices. It answers with a continuous flow of online values.
Simultaneously, the user configures the email action service to dispatch an email at the time a new alarm was created and when the system state comes back to normal.
The online value emitter service processes each single online value and detects a violation as it receives an online value containing *231.1 V*. It calculates an exceedance of the upper bound (230 V) and propagates a new alarm event (*WAS_CREATED*) with the corresponding information.
The previously configured email action service reacts to the alarm event and dispatches a new email to the configured recipients.
Later, the online values drop between 210 V and 230 V. As a result, the online value emitter service detects that the system's state is valid again and propagates this information to the alarm service, which outputs a new alarm event (*SET_BACK_TO_NORMAL*).
Again, the email action service reacts to this event by sending an email to the user.
After time has passed, the user decides to remove his configured emitter. Since the emitter reference is shared between the microservices, the online value emitter service broadcasts that the emitter was deleted from the system.

4.1.4 System Properties

4.1.4.1 Scalability

The system was designed to be scalable. This includes functional decomposition (y-axis scaling), horizontal duplication (x-axis-scaling) and data partitioning (z-axis scaling).
The implemented system was built by detecting functional components and their bounds. Each single functionality of the Alarm System was separated. I have even decided to refrain from designing a service, that is capable of handling multiple tasks, for instance, a single emitter service that is capable of processing multiple types of measurement readings (online data, historical data and more). Instead, each microservice is partitioned by a single purpose. This decision brings benefits in terms of scaling to a finer grade and offers much better code quality. A service with a single responsibility is much easier to understand and maintain than a complex monolithic service that handles many tasks simultaneously. The downside of this approach is the higher price of resources, because more infrastructure is required,e.g., more required virtualization (containers), web servers, etc.
Each service can be run as a redundant unit in parallel. This is mainly possible due to the partitioned messaging broker and the load balancer. At the current state, the round-robin load-balancing logic is quite simple and can be optimized on demand.

The possibility of combining x- and y-axis scaling (horizontal duplication and functional decomposition) within the system enables the potential for increasing data throughput in a very cost efficient manner. With the help of adequate monitoring capabilities, functional bottlenecks can be found and the corresponding services can be started in parallel as redundant instances.
The system should be capable of handling very large data sets. When data grows larger, query time for data base reads may increase proportionally. Another problem occurs when services need to read very large data sets. This problem occurs especially with services that need to load many configuration items at start up, e.g., when the scheduled escalation service reads every configured scheduled escalation on start up.
To combat this, data was separated into certain data groups, also known as data partitioning (z-axis scaling). For example, alarm logs could be separated by month or year and they were logged in order to speed up time-ranged queries. Furthermore, configurations could be separated by referencing customer projects in order to reduce the loaded data amount of a service on start up.
The Apache Cassandra data-base includes integrated features for data partitioning and scales with growing data demands. It is possible to add additional nodes to Cassandra in order to increase the speed of reads and writes. Cassandra's scaling capabilities imply it has the potential to handle a growing amount of data.

4.1.4.2 Fault Tolerance

Due to full scaling (x-, y- and z-axis scaling) capabilities, as described in chapter 4.1.4.1, the system provides a very high fault tolerance.
All services can be run as redundant instances sharing their work load. When a service becomes unavailable, another service of the same type can undertake its tasks.
Offline services that miss one or more messages and then come back online can access the message logs of Apache Kafka in order to update their state. This capability is restricted to a fixed number of messages that can be replayed, which is related to Kafka's configuration.
The system was designed to prevent a single point of failure. Redundant Service Discoveries, message brokers and data-base nodes ensure infrastructural fault tolerance. By using a Service Discovery, even the configuration server can run with multiple instances.
In order to ensure such fault tolerance, the infrastructure of the microservice system needs a thoughtful set up. The appropriate configuration of infrastructure components, like Kafka, Cassandra, etc. for this requires some further investigations.

4.1.4.3 Maintainability

Since each service was implemented by considering the single responsibility principle, the extent of a single service is very modest, as shown in the table below. Please consider that the implemented services are prototypes that do not yet meet all requirements.

Name	Data Provider	Emitter	Alarm	Email Action	Scheduled Escalation	AVG
Lines of Code	1358	2599	1861	712	1134	1532.8

All presented services are quite easy to understand and can be replaced within a few weeks with a total rewrite. Bugs in business logic and algorithms can be found quickly and easily. And with the help of tracing, bugs can even be found in complex algorithms with multiple services involved. When it comes to testing, each service functionality can be validated via unit tests and black box integration tests. During development, the implementation of tests granted stability and helped find hidden bugs.
In addition, the stability of interfaces can be ensured via *Spring Cloud Contracts*, a testing suite for interface mocking that is currently not integrated.
However, some errors may occur, but only after a lot of communication and data-base transactions have been performed. This is often caused by the consistency trade-offs of Cassandra and Kafka. Consistency faults offer a high risk and should be well-examined in some sort of long-term test, since their cause is difficult to reason about. Furthermore, it is potentially very painful to bring an inconsistent distributed system back to consistency in a production environment, because manual operations such as data-base migrations could be necessary.
In conclusion, the implemented system increases the development and maintenance speed within microservices. This include fixing business logic bugs and bringing new features online. But everything around this implementation, including all infrastructural components, need much more thoughtful attention before such a migration could be considered complete. Monitoring tools should be installed well in order to save resources in technical emergencies.

4.1.4.4 Installation Effort

As mentioned in chapter 4.1.4.3, the installation of the implemented system is not trivial. First of all, each microservices has to be built with the appropriate configuration. The configuration includes addresses for Service Discovery, data-bases, message brokers and the project name (for data partitioning).
Subsequently, the infrastructural components need to be set up properly. This step takes a lot of time and depends on the environment of the system it is installed on. It would be imaginable to install everything as a single instance on a machine, but this is practically deceptive, since the implemented microservice artefacts are quite large with all of their dependencies (each service has a size of 100 Megabyte and more) and infrastructural overhead with network-based communication is very expensive. The system would lose all the benefits of a distributed system while committing to all the drawbacks.

4.2 Challenges

4.2.1 Detecting Seams in Legacy Code

4.2.1.1 Finding Seams within the Alarm System

Unfortunately, it was very hard to reuse certain functionalities from the Alarm System. The Alarm System suite itself is highly decoupled, but included components are mostly not. Many classes have strong dependencies to infrastructure integration libraries that are not or are only partially suitable for a microservice architecture. For example, complex UI layers and database abstractions that help to integrate technologies like NetBeans UI components or RDBMS databases. Within GridVis, infrastructure integration libraries are complex systems that help developers to ensure code quality and code re-usage within the monolith. Extracting an infrastructure integration library to a single microservice is nearly impossible, because the number of dependencies of such a library is very large and each library has to be split up and adjusted individually. Finally, the system would only become more difficult with respect to maintainability and extensibility.

4.2.1.2 Example: Extracting the Online Value Check

Finding seams within existing source code is a huge advantage, because development time can be shortened and costs for the migration are reduced. Within this chapter, I want to give an example of a possible seam and its extraction.
Within GridVis, every emitter check implements the *IAlarmEmitter* interface, which ensures the implementation of the *execute* method. The execute method contains about 100 lines of code modelling the base algorithm outlined in Fig. 3.8. It considers the emitter's configuration and triggers the persistence of a new alarm event if the system state is not valid any more.
Therefore, the argument of type *IAlarmEmitterExecute* is passed into the execute method and models the *facet* pattern. The IAlarmEmitterExecute parameter includes everything necessary to access the following instances: the emitter's own **configuration** (e. g. its bounds for validation), **existing alarms** with their alarm details, the ability to **create new alarm instances**, **access to the last known online value** and the **persistence of generated alarms**.
However, the seam's extraction cannot be processed without adjustment. The access and the persistence of alarms must not be applied within the emitter context. Consequently, the execute method should not persist within a new alarm any more, but should instead send a message with an *online value emitter update event* (message format described in chapter 4.1.2.8). Moreover, everything in touch with the GridVis internal ConfigurationItems must be removed, since they are coupled too closely with the UI.
Instead of saving and reading alarms, the online value action service needs to store the most recent state of the system, referenced to the processed IValue (see 3.2.5.1).
In summary, side effects, input and output of the online value check core method *execute* had to be

adjusted. It was necessary to remove all dependencies on infrastructure while adding alternative technologies, like message broker or new database libraries.

4.2.2 Holding the System State Consistent

The communication model, outlined in chapter 4.1.3.1, works quite well when the first provided online values do not drive an alarm event. On the other side, a fundamental problem occurs when the first online values trigger an alarm directly at start.

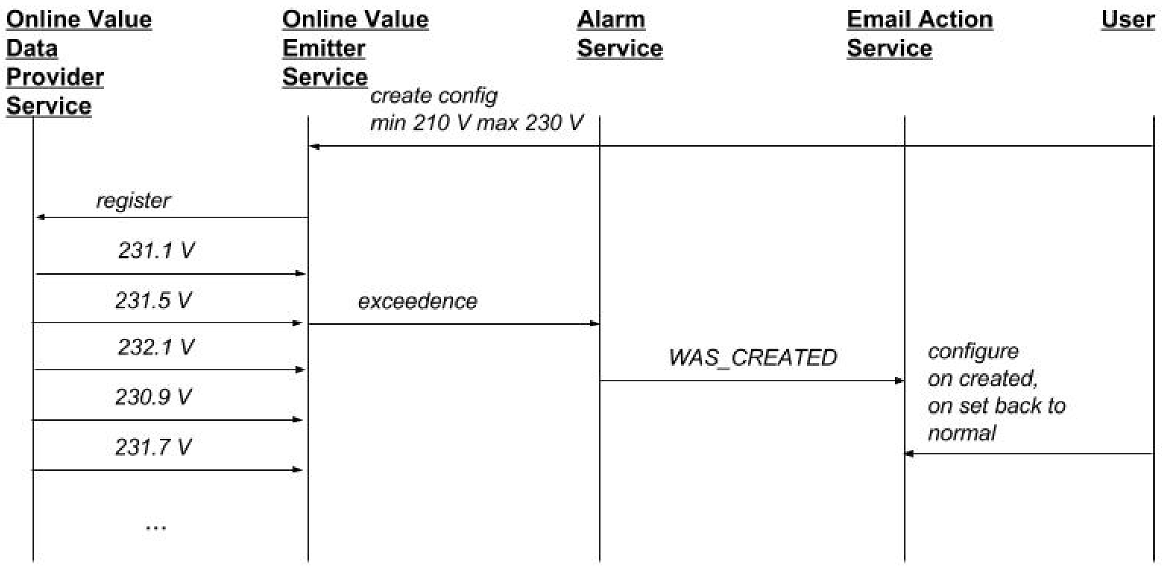

Online Value Data Provider Service
Online Value Emitter Service
Alarm Service
Email Action Service
User
create config min 210 V max 230 V
register
231.1 V
231.5 V
exceedence
232.1 V
WAS_CREATED
configure on created, on set back to normal
230.9 V
231.7 V
...

Figure 4.8: Example of unexpected system behavior, caused by distributed system state.

Within the monolithic Alarm System, the emitter saving new alarms depends on the alarm plan, which maps the alarm events to actions. It was not possible in such a system to create alarms without a responsible configuration. But within the distributed implementation of the Alarm System, the emitter service remembers the violated system state and does not trigger a new alarm event if the configuration of the email action service is applied after the *WAS_CREATED* alarm event was sent. This is an architectural problem in the current implementation and results in a process where the email action service will not dispatch an email action, because it had no configuration at the time the *WAS_CREATED* alarm event was received. As an example, Fig. 4.8 outlines the problem.
A straight-forward solution could consider the distributed system's state as the set of occurred events. This implies a possibility to synchronize the system's distributed components by managing an accessible event log, such as the event log Apache Kafka provides.

On start up, or on certain actions (like setting up a configuration), a microservice should read a defined number of messages and apply all internal actions related to each individual message. The number of messages could be calculated with the help of the message broker.

4.2.3 Data Partitioning

4.2.3.1 General Problem

Since distributed systems may scale to enormous dimensions, data modelling must be applied thoughtfully. Partitioning becomes reasonable for reading large data sets or for writing data at high frequencies.
When partitioning data on Cassandra nodes, it is important to arrange data on all nodes equally in order to avoid slow reads and writes. Cassandra distributes its data through a user-defined partition key. The partition key is hashed and each node within the Cassandra cluster is related to a range within this hash. The partition key can consist of an aggregated set of data values. Finding adequate partition keys can be accomplished by data analysis in combination with a good understanding of processes, services and customer behaviors.

4.2.3.2 Example: Finding Adequate Partition Keys

The following table outlines an example of different granulated partition keys. The composition of each key depends on the entity's access frequency, its data volume and its relations.

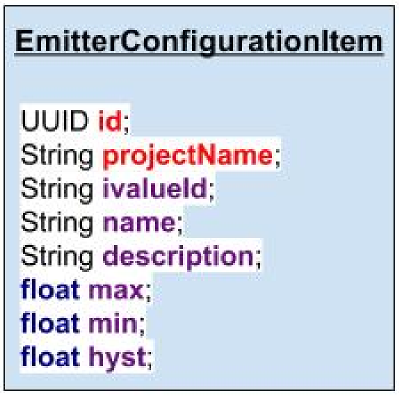
Entity	LastSystemState	ActiveScheduledEscalation	AlarmLogs
Partition key	*EmitterId*	*ProjectName:EmitterId*	*Month:Year:EmitterId*

The *LastSystemState* belongs to a single running emitter. As a consequence, the amount of data is limited to one entity per emitter. There is no need to create a combined partition key, because the mapping from *LastSystemState* to an emitter can be applied with a single emitter ID. This results in an equal distribution of all *LastEmitterState* entities around the Cassandra cluster.
ActiveScheduledEscalations have a n:1 relationship to emitters. If the partition key had contained a time unit representing the hour of the alarm event's occurrence, an unequal data distribution within the Cassandra cluster would have occurred. This behaviour is caused by the business logic favouring a higher data density at customer-specific alarm-intense times. Partitioning by an emitter ID is sufficient, since thousands of *ActiveScheduledEscalations* per emitter instance will practically not exist.
Because reading of a specific *ActiveScheduledEscalation* is not initiated by another service's event message, the service needs to actively and independently read all *ActiveScheduledEscalations*. As a result, the number of *ActiveScheduledEscalations* should be limited to prevent the service from reading **all** *ActiveScheduledEscalations* at once. That is why a partition key is necessary

to reduce the amount of data that the services load on start. I have introduced a partition key containing a specific project name. The project name represents the data scope for which the service is responsible. Unfortunately, this still implies an unevenly distributed data set among the Cassandra nodes, which can be optimized in further development steps.
AlarmLogs is the most frequent persisted entity within the microservice-based Alarm System. Each AlarmLog has an n:1 relationship to an Emitter. In comparison to *ActiveScheduledEscalations*, the *AlarmLog* could possibly exist many times more often per emitter instance. Over time, the persistence of thousands of AlarmLog instances per emitter is imaginable. Therefore, a partitioning by a wider time range (year or month:year) is reasonable.

4.2.4 Data Base Denormalization

The utilization of the Cassandra data-base provides benefits for managing large data sets in a fault tolerant manner. But on the other hand, one drawback involves the redundant persistence of data by denormalized data schemata. The design of Cassandra's data schemata is highly dependent on the required queries. Each query must at least contain the corresponding partition key.
Fig. 4.9 outlines the properties of an *EmitterConfigurationItem* entity. Each *EmitterConfigurationItem* contains a partition key, composed of an *ID* and a corresponding *projectName*.
In order to query a single *EmitterConfigurationItem* instance, the client has to use both: the ID as well as the project name for its query. Unfortunately, by using this data schema, queries are restricted. For example, a query asking for all *EmitterConfigurationItems* belonging to a given project name is not possible.



```
EmitterConfigurationItem

UUID id;
String projectName;
String ivalueId;
String name;
String description;
float max;
float min;
float hyst;
```

Figure 4.9: EmitterConfigurationItem entity with red highlighted partition key components.

In this case, it is necessary to outline which properties are really required in a query result. Next, all of the outlined properties are put into a new schema, e.g., *EmitterConfigurationItem-ByProjectName*, as displayed in Fig. 4.10.
Within an *EmitterConfigurationItemByProjectName* entity, the only partition key is the *projectName* property. In this example, and in most cases, not all properties of the original entity (*EmitterConfigurationItem*) are required, implying that derived entities (*EmitterConfigurationItemByProjectName*) are usually smaller than their origin.
In addition, it is important to keep the origin and all its derived schemata consistent. This means that the deletion of an *EmitterConfigurationItem* implies the deletion of an *Emitter-ConfigurationItemByProjectName*. The problem of keeping such consistencies can be reduced through the use of designated tests specifying such requirements in detail.

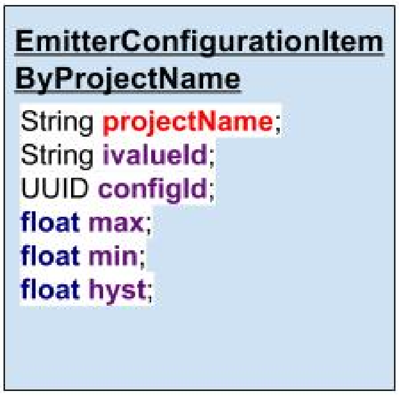


Figure 4.10: Additional (denormalized) EmitterConfigurationItemByProjectName entity for querying by project name.

4.3 Analysis of the Architecture

4.3.1 Advantages Compared to the monolithic Alarm System

Due to its design, the scaling approach for the monolithic GridVis system is hardly possible. GridVis Desktop was never conceived to scale to a higher data throughput and the service application can only be scaled as a whole (x-axis scaling with a load balancer). Within GridVis, tackling bottlenecks by duplicating **internal** functionalities is very difficult to control at runtime (e.g., with advanced thread management) and reaches its limits when hardware is used to capacity.
On the other hand, the presented microservice-based Alarm System can scale in relation to customer's demands dynamically. Bottlenecks within the microservice-based Alarm System and its infrastructure can be opened by running additional services or infrastructure instances in parallel by utilizing data partitioning.
The microservice-based Alarm System is also more fault tolerant, since it is able to run independently from the rest of the GridVis system. For instance, if the GridVis Service application crashes, the creation of new alarms will stop, but the customer still has access to his alarm logs.
Another benefit is an optimized development process, because business logic can be implemented faster and more independently. The development is supported by separation of business functionalities, decoupled communication and a default technology stack helping to integrate new tools and infrastructure faster. The new architecture engages developers to write clean and purposeful source code while sticking to defined interfaces and service boundaries.
After a CI environment has been properly set up for services, newly-developed features can be deployed faster than features within the much larger monolithic GridVis system.

In addition, the architecture scales with increasing size of workforce. As described in chapter 2.1.2.12, responsibilities can be partitioned on higher-order BCs. For example, one team is responsible for the Alarm System BC and another team is responsible for developing and managing the user BC.
The architecture enforces the definition of thoughtful interfaces and the encapsulation of logic within service boundaries. In addition, service encapsulation enables the option to outsource the development of microservice to external companies.
Since every service is nearly cloud-ready via networking communication, the user interface can be designed very universally (e.g., with a web view, a desktop client, a mobile application, etc.). This approach helps to install multiple types of UI clients on a broad variety of devices using the same interfaces.
The utilization of the presented technology stack should increase development speed, but the utilisation is not strictly enforced. It is an option that grants a better technological flexibility than the monolithic GridVis does, because the GridVis system relies heavily on Java, existing infrastructural libraries and the NetBeans platform.

4.3.2 Disadvantages Compared to the monolithic Alarm System

In contrast to the monolithic GridVis Alarm System, the microservice-based Alarm System may lack in consistency. This is a major difficulty becomes more troublesome when network distances increase.
The monolithic GridVis system was designed as a platform-independent *Rich Client* application, built and packed as single executables. This implementation enables a much easier delivery. In contrast, the microservice-based Alarm System brings five relatively large (c. a. 100 Megabyte) artefacts requiring complex infrastructure. It might be possible to ship a preconfigured container solution with a powerful installer containing the microservice-based Alarm System. This approach needs further investigation.
As already outlined, in order to deploy and run the microservice-based Alarm System, it is necessary to set up a lot of infrastructure first. This includes the installation of adequate CI pipelines with a purposeful test environment, virtualization management and monitoring systems. Moreover, maintenance of the cluster resources will become important during production stage.
If Cassandra is used as a data-base, data schemata have to be designed thoughtfully, since extending the data model is more work-intensive than extending a RDBMS schema due to denormalization of Cassandra's data schemata.

4.4 Future Prospects

4.4.1 Cost Efficiency

One driving factor of cloud technologies, especially microservice architectures, is the related cost efficiency: PaaS and SaaS enable economical predictability and a more cost-efficient usage of resources.
In [42], Zloch shows that the computation time, memory consumption and CPU usage of a microservice-based system is significantly higher than the usage of these resources within a monolithic approach. Nevertheless, the situation changes at a higher data throughput, when microservice-based systems are able to handle the mentioned resources more efficiently than a monolithic system.
It would be financially rewarding for the GridVis project to know which data throughput is required on a certain infrastructure in order to make a cloud-based microservice solution profitable.

4.4.2 Ensuring Runtime Stabilization

At the current stage of development, all unit and integration tests of the implemented microservices have been successfully passed. This approach ensures a fundamental stability within services, but as outlined in chapter 2.1.2.11, system wide integration tests are required. The usage of system-wide integration tests includes load tests, where bottlenecks can be identified, and failure tests, where services are randomly shut down.

4.4.3 Alternative Technologies

I put a lot of effort into balancing different technologies to find a purposeful starting point for new services. Naturally, no technology offers a silver bullet. When time passes and priority for architectural principles change, better technology selections will be possible. Therefore, further investigation is advisable in order to optimize development and maintenance.

4.4.4 Finding a Suitable UI Architecture

In chapter 3.3.9, I have outlined the difficulties of finding a suitable UI architecture. At the current state, the implemented UI is not appropriate for any customer.
With existing UI requirements from stakeholders, it would be possible to investigate such an architecture. Since a UI is essential, the finding of an appropriate solution has a high priority.

Chapter 5

Conclusion

Within this thesis, I have presented the possibility of an iterative migrating approach for the monolithic software application *GridVis*, although reusing existing source code was mostly not practicable.

Functional components within the software were extracted with the help of Domain Driven Design (DDD). In order to demonstrate proof of concepts, I have extracted an existing subsystem of the GridVis application and migrated it to a microservice-based implementation.

When searching for adequate microservice-specific technologies, the discovery of fitting technology classes was quite simple, but their individual distinction became work-intensive, because benefits and drawbacks exist at a very finely grained level.

Nevertheless, I set up a certain technology stack solving the common distributed system problems and satisfying all mentioned requirements. This technology stack can be utilized as a starting point for developing new project-specific microservices.

The presented prototype provides all the benefits of a typical microservice system, since it is scalable, enables fault-tolerant operation and supports development productivity.

Finally, I have outlined some potential problem spaces, future prospects, and compared the implemented architecture approach with the monolithic GridVis system.

In my personal opinion, the migration to a microservice-based architecture is reasonable, especially if its architectural principles help to achieve the company's strategic goals, although the beginning of such a migration may be cumbersome and require extensive investigation.

Bibliography

[1] `http://zookeeper.apache.org`.
Accessed: 2017-09-28.

[2] `http://www.consul.io`.
Accessed: 2017-09-28.

[3] `https://github.com/Netflix/eureka`.
Accessed: 2017-09-28.

[4] `https://spring.io/`.
Accessed: 2017-09-28.

[5] `https://github.com/Netflix/ribbon`.
Accessed: 2017-09-28.

[6] `https://www.elastic.co/products/logstash`.
Accessed: 2017-09-28.

[7] `https://www.elastic.co/de/products/kibana`.
Accessed: 2017-09-28.

[8] `https://www.janitza.de/energie-und-spannungsqualitaets-messtechnik.html`.
Accessed: 2017-07-17.

[9] `http://www.fmc-modeling.org/notation_reference`.
Accessed: 2017-09-28.

[10] C. B. Alan MacCormack, John Rusnack, *Exploring the Duality Between Product and Organization Architectures*, Harvard Business School, 2007.

[11] H. Boeck, *NetBeans Platform 7*, Galileo Press, 2 ed., 2011.

[12] J. Borchers, *Migration auf cloudbasierte plattformen aus sicht des klassischen reengineering*.
Uni Siegen: Fachgruppenberichte, 2016.

[13] E. BREWER, *Cap twelve years later: How the rules have changed*, IEEE Computer Society, (2012).

[14] S. CLOUD, *Spring cloud sleuth*.
`https://cloud.spring.io/spring-cloud-sleuth/`.
Accessed: 2017-08-22.

[15] M. CONWAY, *How do committees invent*, Datamation, (1968).

[16] J. EDITORIAL DEPARTMENT, *Alarm-management beschreibung*.
`https://wiki.janitza.de/display/GH/Alarm-Management+-+Beschreibung`.
Accessed: 2017-07-19.

[17] P. EM. DR.-ING. SIEGFRIED WENDT, *Quick introduction*.
`http://www.fmc-modeling.org/quick-intro`.
Accessed: 2017-07-27.

[18] E. EVANS, *Domain-Driven Design Tackling Complexity in the Heart of Software*, Addison-Wesley, 1 ed., 2004.

[19] M. C. FEATHERS, *Working Effectively With Legacy Code*, Prentice Hall Technical Reference, 2005.

[20] A. FOUNDATION, *Documentation*.
`https://kafka.apache.org/documentation/`.
Accessed: 2017-10-18.

[21] HASHICORP, *Basic architecture of consul*.
`https://www.consul.io/intro/index.html`.
Accessed: 2017-07-06.

[22] P. HUMPHREY, *Zipkin*.
`https://content.pivotal.io/blog/understanding-when-to-use-rabbitmq-or-apache-kafka`.
Accessed: 2017-07-12.

[23] E. H. JEFF CARPENTER, *Cassandra The definitive Guide Distributed Data at Web Scale*, O'Reilly, 6 ed., 2016.

[24] J. B. KATHARINA PROBST, *Engineering trade-offs and the netflix api re-architecture*, Netflix Technology Blog, (2016).
accessed 29.06.17.

[25] V. R. B. NACHIAPPAN NAGAPPAN, BRENDAN MURPHY, *THE INFLUENCE OF ORGANIZATIONAL STRUCTURE ON SOFTWARE QUALITY: AN EMPIRICAL CASE STUDY*, Microsoft, 2016.

[26] D. NAMIOT AND M. SNEPS-SNEPPE, *On micro-services architecture*, Open Information Technologies, (2014).

[27] N. NANNONI, *Message-oriented middleware for scalable data analytics architectures*, Master's thesis, SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY, Stockholm, 2015.

[28] NETFLIX, *Working with load balancers*.
`https://github.com/Netflix/ribbon/wiki/Working-with-load-balancers`.
Accessed: 2017-07-06.

[29] S. NEWMAN, *Building Microservices DESIGNING FINE-GRAINED SYSTEMS*, O'Reilly, 2 ed., 2015.

[30] NVISIA, *Comparison of spring cloud with eureka vs. consul.io*.
`http://www.nvisia.com/insights/comparison-of-spring-cloud-with-eureka-vs.-consul`.
Accessed: 2017-08-21.

[31] J. M. G. O. MUSTAFA, *Optimizing economics of microservices by planning for granularity level*.
Uni Oldenburg, 2017.
Experience Report.

[32] E. S. RAYMOND, *The Art of Unix Programming*, Addison-Wesley Professional, 2016.

[33] R. RV, *Spring Microservices build scalable microservices with spring docker and mesos*, packt publishing ltd., 1 ed., 2016.

[34] B. H. SIGELMAN, L. A. BARROSO, M. BURROWS, P. STEPHENSON, M. PLAKAL, D. BEAVER, S. JASPAN, AND C. SHANBHAG, *Dapper, a large-scale distributed systems tracing infrastructure*, tech. rep., Google, Inc., 2010.

[35] P. SOFTWARE, *Spring boot documentation*.
`http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#boot-documentation`.
Accessed: 2017-08-21.

[36] ——, *Spring cloud*.
`https://projects.spring.io/spring-cloud/`.
Accessed: 2017-08-21.

[37] ——, *Spring framework introduction*.
`https://projects.spring.io/spring-framework/`.

Accessed: 2017-08-21.

[38] SPRING, *Spring cloud config*.
`https://cloud.spring.io/spring-cloud-config/`.
Accessed: 2017-07-10.

[39] E. WOLFF, *Microservices Grundlagen flexibler Softwarearchitekturen*, dpunkt.verlag, 1 ed., 2016.

[40] ——, *What are microservices*, Addison-Wesley Professional, (2016).

[41] ZIPKIN, *Zipkin*.
`http://zipkin.io/`.
Accessed: 2017-07-10.

[42] M. ZLOCH, *Eine microservice-architektur für die semantische analyse im kontext der computerlinguistik*, Master's thesis, Christian-Albrechts-Universität, Kiel, 2016.