



UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

Operating Interorganizational Logistical Workflows on a Shared Software Platform

*Betrieb organisationsübergreifender logistischer
Prozesse auf einer geteilten Softwareplattform*

Master thesis

as part of the degree program

Media informatics

at the University of Lübeck

written by

Nikolas Knickrehm

issued and supervised by

Prof. Dr. Martin Leucker

Lübeck, 05.05.2020

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne die Benutzung anderer als der angegebenen Hilfsmittel selbstständig verfasst habe; die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe des Literaturzitats gekennzeichnet.

(Nikolas Knickrehm)
Lübeck, den 05.05.2020

Acknowledgements

Before starting with the main content of this thesis, I would like to express my sincere gratitude to a number of people.

During the first semesters of my Bachelor program I was not so ambitious, because I did not agree to certain contents of the program and was unsure if I had chosen the right carrier path. This changed drastically when I started working for Professor Martin Leucker at the Institute for Software Engineering and Programming Languages. I have learned a lot during this time and am very humble to the level of trust, that I received right from the start. This had a huge impact on my personal development for which I can not thank him enough.

To my colleagues at the University of Lübeck I would like to say, that I have never worked in a more friendly and open environment in which everyone's opinion was always heard and respected during discussions and where you could always find someone to talk to. Whether about work related or personal issues. Thank you all, guys!

When I started my own journey and left my parents' house, I was only 18 years old and still had a lot to learn. Even though I made a lot of mistakes along the way my family always supported me in every way possible. Thank you for raising me, supporting me, trusting me, for always being by my side and for allowing me to go my own way!

I thank my aunt Frauke and my cousin Lotta for sacrificing their weekend to correct a number of mistakes in this thesis. You made the last few days a lot easier for me.

Lastly, I would like to thank you, Steffi! When we first met, you saw someone in me that I could not at that time. Your love, trust and support guided me through my ups and downs. I am deeply impressed by your ambition and happiness and by your way of showing me, that I too could be that person. You taught me a lot and if it was not for you, god knows where I would be today. I love you from all of my heart.

Abstract This master thesis examines interorganizational logistical workflows connected to freight transportation through inland-waterways inside the European Union (EU). As the lack of interoperability is one of the major challenges for companies in the logistics domain today, a new software platform could be used to connect all stakeholders in order to operate shared workflows. A concept for such Logistics Platform is elaborated utilizing scenarios, which describe possible use cases for the platform and a thorough analysis of different architecture patterns for interconnected software systems. The resulting event-driven architecture is further evolved by creating the design specification for all components, which together fulfill the requirements for the Logistics Platform. Following an incremental software development process, the platform design is then implemented to create a prototype, incorporating the most important aspects of the design concept, which is then evaluated by simulating a logistical workflow, that is based on a real business case.

Kurzfassung In dieser Masterarbeit werden logistische Prozesse im Bereich der Binnenschifffahrt innerhalb der Europäischen Union (EU) untersucht, die von mehreren Unternehmen gemeinsam durchgeführt werden. Da der Mangel an Interoperabilität in der Branche aktuell als eine der größten Herausforderungen gilt, könnte eine neue Softwareplattform, die alle Akteure miteinander vernetzt, eine Lösung sein, um unternehmensübergreifende logistische Prozesse abzuwickeln. Das Konzept für eine solche Logistikplattform wird daher unter Verwendung von Szenarien erarbeitet, welche potentielle Anwendungsfälle eines solchen Systems beschreiben. Unterschiedliche Architekturansätze für verteilte Softwaresysteme werden auf Basis dieser Szenarien untersucht und das Konzept einer ereignisgesteuerten Softwareplattform samt aller benötigten Teilkomponenten ausgearbeitet. In einem inkrementellen Softwareentwicklungsprozess wird ein Prototyp erstellt, der die wichtigsten Aspekte der Logistikplattform beinhaltet. Dieser Prototyp wird anschließend durch die Simulation eines logistischen Prozesses evaluiert, der auf einem realen Geschäftsfall basiert.

Contents

1. Introduction	1
1.1. Terminology	2
1.2. Goals of this Thesis	3
1.3. Related Work	3
1.4. Outline	3
2. Contextual Analysis	5
2.1. Transportation through Inland-Waterways	6
2.2. Stakeholders	6
2.2.1. Port Authorities	6
2.2.2. Shipping Companies	7
2.2.3. Commercial Customers	8
2.3. Summary	8
3. Methodology	11
3.1. Software Architecture Analysis Method	11
3.2. Requirements Engineering	11
3.3. Incremental Software Development	12
3.4. Combined Methodology	12
4. Scenarios for the Logistics Platform	15
4.1. Exemplary Logistical Process	16
4.2. Controlling Access to Information	18
4.3. Integrating a new Actor	20
4.4. Extending an existing Workflow	20
4.5. Updating an existing Component	21
4.6. Summary	22
5. Requirements Analysis	23
5.1. Information Exchange Between Companies	23
5.2. Deploying Components to the Platform	24
5.2.1. Deployment Pipelines	25
5.2.2. Automated Testing	25
5.2.3. Continuous Integration	25
5.2.4. Automated Low-Risk Releases	26

5.3. Monitoring	27
5.3.1. Business Level	28
5.3.2. Application Level	28
5.3.3. Infrastructure Level	29
5.3.4. Client Software Level	30
5.3.5. Deployment Pipeline Level	31
5.4. Summary	31
6. Designing the Logistics Platform	33
6.1. Comparing Architecture Patterns	33
6.1.1. Monolith	34
6.1.2. Service-Oriented Architecture	38
6.1.3. Microservice Architecture	41
6.1.4. Event-Driven Architecture	48
6.1.5. Conclusion	52
6.2. Platform Architecture Overview	53
6.3. Designing the Software Platform	54
6.3.1. Choosing a Container Orchestration Engine	54
6.3.2. Choosing an Event Bus	57
6.3.3. Designing the Event Structure	59
6.3.4. Choosing a Monitoring System	64
6.3.5. Choosing a Continuous Integration System	66
6.4. Summary	67
7. Implementation of the Platform Design	69
7.1. Setting Up Kubernetes	69
7.2. Setting Up Kafka	72
7.3. Setting Up the Monitoring System	73
7.3.1. Monitoring the Infrastructure Layer	73
7.3.2. Monitoring Kafka	74
7.3.3. Monitoring Other Layers	75
7.4. Summary	75
8. Evaluation	77
8.1. Evaluation Goals	77
8.2. Setting Up the Evaluation	78
8.3. Simulating the Logistical Workflow	79
8.4. Monitoring the Logistical Workflow	81
8.5. Altering a Logistical Workflow	82
8.6. Summary	83

9. Concluding Remarks	85
9.1. Limitations	85
9.2. Future Work	86
A. Appendix	87
A.1. Functional Requirements for the Logistics Platform	87
A.2. Enclosed DVD	90

1. Introduction

Freight traffic through inland waterways is based on complex logistical workflows spanned across multiple organizations. Unfortunately those organizations are currently not sharing information technology (IT) infrastructure on a larger scale, so digitalization efforts, that exceed organizational boundaries, are hard to implement. For example the movements of large ships on the Elbe river and inside the Hamburg harbor are coordinated through the highly advanced Port River Information System Elbe (PRISE) since 2014 (Hafen Hamburg Marketing e.V., 2014). This is an example for a local digitalization project which had a positive impact within a certain area and lead to the Hamburg Port Authority to push digitalization efforts for their whole logistical supply-chain aiming to optimize economic and ecologic outcomes (Hafen Hamburg Marketing e.V., 2020). Other harbors in Europe now follow suit and develop similar technologies which they use for their specific use cases.

Such isolated efforts can result in positive effects for some companies, just like the PRISE system in Hamburg, but lack the holistic view on logistical workflows, which often involve multiple companies like port authorities and shipping companies. When every port authority now develops their own system, which has proprietary interfaces, the number of interfaces a shipping company would have to implement and maintain soon becomes unmanageable. This thesis suggests, that certain workflows within the logistical domain should be standardized and operated on a shared software platform, which allows safe workflow-related communication between all stakeholders.

The requirements for such Logistics Platform will be defined in this thesis using a real business case of a typical logistical workflow involving multiple companies. Different architectural approaches for implementing a shared software platform on the larger scale will be introduced and compared, before a prototype is implemented. This prototype will then be evaluated by deploying a simplified version of the real logistical workflow to the platform to demonstrate a positive business outcome, that was not possible before, due to the lack of interoperability between the stakeholders.

1.1. Terminology

This thesis will make use of the terms *business process*, *logistical process* and *workflow* which are often used synonymously throughout the literature. The demarcation of those three words followed in this thesis is based on definitions made by (Fill, 2013; Jablonski, 1995; Karagiannis, Junginger, & Strobl, 1996).

A business processes works on the domain level and includes tasks performed by domain experts, who are not required to have a detailed understanding of the IT implementation of this process.

A logistical processes is a business process within the logistics domain. It is performed by experts working within the logistics field.

A workflow is the technological realization of a business process that refers to technical properties like information exchange between different services. People working on a workflow are required to have technical know-how about the underlying IT-systems.

A logistical workflow is a workflow, which is the technological realization of a logistical process.

Workflow Management or “*Business Process Management (BPM)*” is defined “[as] a discipline involving any combination of modeling, automation, execution, control, measurement and optimization of business activity flows, in support of enterprise goals, spanning systems, employees, customers and partners within and beyond the enterprise boundaries” (Workflow Management Coalition, 2014). Computer programs implementing BPM are often referred to as *Business Process Management Systems*. Such systems can deploy, monitor and control workflows that are described in some standardized format like the Business Process Model and Notation (BPMN) (Object Management Group, 2011).

While it is possible to describe and even execute logistical workflows using BPMN or similar notations, this implies certain architecture decisions, which will be discussed in Section 6.1 on page 33.

1.2. Goals of this Thesis

This thesis aims to prove that interorganizational workflows can be operated on a shared software platform, which allows safe and standardized communication between companies. Therefore, a real business case will be utilized to design, implement and evaluate such Logistics Platform following existing methodologies of Software Engineering, that will be further explained in Chapter 3 on page 11. As using only one real use case comes at the cost of restricted generalisability, abstraction is used where ever possible while recommendations for further means of evaluation are given at the end of this thesis. The result of this thesis is an architectural concept for a software system which can fulfill the purpose of a Logistics Platform capable of operating workflows between multiple stakeholders, that needs to be evaluated further through future research.

1.3. Related Work

Multiple current and past research projects and theses at the Institute for Software Engineering and Programming Languages (ISP) at the University of Lübeck are focusing the field of maritime logistics in Europe. The research projects MISSION¹, RoRoHafen² and SecurePort³ are cooperative projects of the ISP with multiple industry partners from the logistics domain, most notably the Lübeck harbor authority. Approaches for event-based systems, a concept examined later in this thesis, were already taken into consideration during those projects. (Queßeleit, 2020) described logistical processes at the Lübeck harbor from the perspective of different actors and through his thorough research laid the foundation for an event-driven system, that follows international standards of maritime logistics. This thesis will focus on the Logistics Platform from a more technological perspective and provide a proof of concept for this abstract idea. As the time period during which this thesis was written partly overlaps with the work of (Queßeleit, 2020), the logistical processes that he described in his thesis, could not be taken into consideration.

1.4. Outline

This thesis is split into nine chapters: *Chapter 2 on page 5* will examine the logistics domain as the field of research and introduce the different actors involved. *Chap-*

¹<https://www.isp.uni-luebeck.de/research/projects/mission>

²<https://www.isp.uni-luebeck.de/research/projects/ro-ro-hafen-40>

³<https://www.isp.uni-luebeck.de/research/projects/secureport>

1. Introduction

ter 3 on page 11 explains the methodologies of Software Engineering which will be applied in this thesis. *Chapter 4 on page 15* describes scenarios which are based on a real business case for the Logistics Platform and focus on different aspects and challenges related to operating interorganizational workflows on a shared platform. *Chapter 5 on page 23* contains technological requirements which are clustered to represent different aspects of the platform. *Chapter 6 on page 33* introduces and compares various architecture patterns and explains the design process of the components contained in the software system, which will later become the Logistics Platform. *Chapter 7 on page 69* follows all steps of the implementation process for the platform prototype. *Chapter 8 on page 77* evaluates the platform concept and prototype by deploying and simulating the model of a real logistical workflow. *Chapter 9 on page 85* summarizes this thesis and gives some concluding remarks as well as a prospect for future work.

2. Contextual Analysis

Digital transformation is currently one of the major challenges for companies working in the logistics field. A study conducted by (Maguire et al., 2018) found that out of 433 senior industry and functional executives in logistics, supply chain and transportation, 65% notice fundamental shifts in logistical processes, while 62% say that their own business is experiencing profound transformation. One of the participants, who is the president and chief executive officer (CEO) of a major transportation and logistics provider, claims that *“we’re at a point where there’s more change taking place in this instant than what [he has] seen in 25 years on the front lines”*.

Another study performed by (Hermes, 2019) among 200 decision-makers in the German logistics field showed similar results. The biggest challenge in optimizing supply chains, according to 55% of the participants, was the lack of information exchange between stakeholders. 79% of the participants also said that optimizing the information exchange between stakeholders is the most important measure for optimizing the whole supply chain.

The EU also recognizes the problems faced by the logistics field, explicitly including maritime transport. (European Commission, 2015) names the interoperability of systems and standards to actively connect all players as two of the major challenges of digitalization in the logistics area. The European Logistics Platform¹ is an interest group based in Brussels that connects policy makers and industry stakeholders promoting digitalization efforts in the logistical sector.

In this thesis, a specific subsection of the larger logistical field in the EU is addressed. Transportation of freight over inland-waterways is one of four major ways of transportation in the EU. The other three ways of transportation are road, railway and air transportation. While the Logistics Platform developed in this thesis explicitly targets logistical processes and actors in the maritime context, the platform could be extended to include other areas of the logistics field. Section 2.1 on the following page describes the key features of the logistical subfield examined in this thesis, introducing the different actors involved.

¹<http://www.european-logistics-platform.eu/>

2.1. Transportation through Inland-Waterways

The logistics field focused on in this thesis includes logistical processes and work-flows of transportation through inland-waterways in Europe. This way of transportation is used for many centuries now and is still very important today, especially for commercial shipments (Encyclopedia.com, 2020). Compared to the other three means of commercial freight transportation in Europe, transportation through inland-waterways has a relatively small market share with only 6% which ranks it on the third place behind road and rail transportation (European Commission, 2011; Ionescu, 2016). Figure 2.1 on page 9 shows that road transportation is by far the most common way of transportations today with 76% market share. Still, rail and inland-waterways will become more relevant again in the near future, as they are more efficient than road transportation. In their white paper the EU proposes that transportation over rails and through inland-waterways should be preferred over road transportation as this will have a large positive impact on the European CO² emissions (European Commission, 2011).

The following sections will introduce three of the main actors involved in transportation through inland-waterways:

- Port operators
- Shipping companies
- Commercial customers

2.2. Stakeholders

Today, there are a variety of different companies operating in the maritime logistics field. (Queßeleit, 2020) names three groups of companies that are important for transportation through inland-waterways that should be considered for a Logistics Platform.

2.2.1. Port Authorities

Port authorities, or port operators, are operating one or multiple ports. As this includes a variety of different tasks, they might be split up into different sub-organizations. A port consists of multiple gates and docks which are used by shipping companies. Cargo is moved between different vessels like ships, trucks and trains and might be processed by customs. Because ports are complex ecosystems on their own, where different companies have to cooperate on site, many already

implement cyber-physical systems to optimize their internal processes. One example which was already mentioned in Chapter 1 on page 1 involves the Hamburg Harbor Authority which amongst other things operates a system which optimizes the complex coordination process of large ships on the Elbe river and on the properties of the Hamburg harbor. (Queßeleit, 2020) describes the business processes performed at the Skandinavienkai of the Lübeck harbor which utilize IT on a much smaller scale. In order for the Logistics Platform to succeed, it must be attractive towards many different ports which will be digitalized to various extends. This means that the platform should be capable of hosting all possible workflows to variable extends depending on the requirements of individual stakeholders and integrate with their existing IT infrastructure to connect workflows which are operated outside of the platforms scope. Some of the common roles of port authorities include:

- Operating port and terminal infrastructure
- Handling cargo operations like loading and unloading vessels
- Coordinate vehicle movements (ships, trucks and trains) between ports and cargo destinations
- Managing cargo documents between actors (cargo owners, shipping companies and customs)

2.2.2. Shipping Companies

A shipping company is responsible for transporting cargo owned by their clients. They operate through one or multiple ways of transportation like road, rail, air or sea and might cooperate with other shipping companies in order to fulfill their contracts. In commercial business-to-business (B2B) settings, where clients frequently require the services of a shipping company, there is often an ongoing cooperation between the two companies. Usually shipping companies own and operate the vehicles used to transport their clients freight and make regular journeys between ports during which they ship the goods of many clients at once. Like port operators, there are a variety of different shipping companies, which might provide slightly different services and are digitalized to different degrees. Some companies like Mærsk Line² belong to large cooperations which also own and operate the ports involved in the shipping process. Many shipping companies act on a much smaller scale and are independent from specific port authorities from an organizational perspective. The shipping company involved in the exemplary workflow introduced in Section 4.1 on page 16 is an independent organization which has to cooperate with different port authorities in order to fulfill their delivery contracts.

²<https://www.maersk.com/>

2.2.3. Commercial Customers

Any person or organization could potentially use the services provided by a shipping company in order to outsource the transportation of freight, which can completely eliminate direct interaction with port authorities and other companies apart from the shipping company. In order to use these services, clients sign a contract with the shipping company containing detailed information on what is being transported from which origin to which destination. A commercial customer of a shipping company might rely on transportation through inland-waterways in their own supply-chain and close a permanent contract with a shipping company to ensure that their supply-chain is sustainable. The business case focused on in this thesis is one example where a manufacturer relies on a shipping company to move components between facilities at different locations in Europe.

2.3. Summary

While transportation through inland-waterways ranks only third place when considering the whole logistics market throughout Europe, it will potentially become more relevant again, because it is more efficient compared to road transportation. Industry experts and policy makers agree that the lack of interorganizational information exchange is one of the biggest challenges faced by the logistics industry today, which affects all stakeholders. Companies operating in transportation through inland-waterways include port authorities, shipping companies and commercial customers which all have to cooperate in order to fulfill shipping contracts. As companies, even those belonging to one of the three mentioned groups, differ in many ways, a shared Logistics Platform must be flexible enough to handle various kinds of workflows and integrate with pre-existing IT infrastructure.

This chapter still gave a more abstract perspective on the field of research that requires a more detailed look. Therefore, a real business case involving an interorganizational logistical workflow will be introduced in Chapter 4 on page 15. But first the methodology applied in this thesis will be explained in Chapter 3 on page 11.

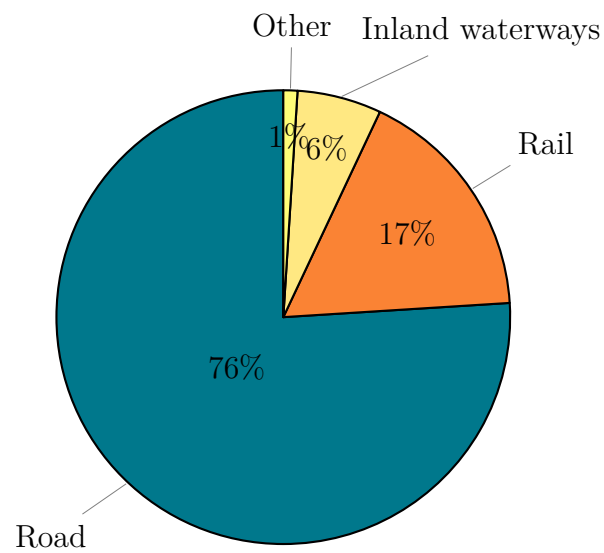


Figure 2.1.: Market shares of the different means of commercial freight transportation in the EU (Ionescu, 2016).

3. Methodology

The aim of this master thesis is to prove that interorganizational logistical business processes in the context of freight traffic through inland waterways can be operated on a shared software platform. Therefore, such a platform will be designed, implemented and evaluated by applying a combined set of different Software Engineering methodologies that are explained in the following sections.

3.1. Software Architecture Analysis Method

(Kazman, Abowd, Bass, & Clements, 1996) introduce a scenario-based analysis for software architectures named the Software Architecture Analysis Method (SAAM). This method can be utilized to evaluate and compare different architectural designs, before starting the implementation process. In order to perform this methodology scenarios are necessary which describe the future use cases of the Logistics Platform. Different architecture designs can be examined using this methodology in order to evaluate if they can, when fully implemented, fulfill those scenarios.

The scenarios necessary to perform the SAAM methodology will include an existing logistical workflow and also operational aspects of the future platform for the platform operator and companies connected to it. Those scenarios will be introduced in Chapter 4 on page 15. In Section 6.1 on page 33 the SAAM methodology will be applied by introducing different architectural patterns and evaluating their suitability according to the scenarios.

3.2. Requirements Engineering

In order to design a safe and reliable software system, that provides all necessary functionalities to become a platform, which hosts interorganizational workflows, a detailed requirements analysis will be performed in Chapter 5 on page 23 following the requirements engineering process introduced by (Kotonya & Sommerville, 1998). Originally the SAAM methodology was developed as an alternative to requirements engineering as the latter focuses merely on the detailed technological aspects, while

scenarios are more flexible and provide a richer look on a software system. In this thesis both methodologies will be combined in order to translate the scenarios into technical requirements which will be used during the implementation process of the software system in Chapter 7 on page 69.

3.3. Incremental Software Development

(Philip, Afolabi, Adeniran, Ishaya, & Oluwatolani, 2010) compare different software development methodologies according to key characteristics of typical software projects. According to this study, the incremental software development process (Larman & Basili, 2003) can be used for complex software projects as it provides the option to split a project into multiple incremental stages, that all result in a functional prototype. This makes the incremental software development process a good fit for the Logistics Platform during the scope of this thesis and also beyond, as there are a lot of different aspects for such platform, that could be implemented in multiple stages enabling evaluation at all of those stages.

The incremental build model is a direct derivation of the waterfall model (Panel & of Naval Research, 1956), which means, that all requirements for the future software system are still acquired at once early in the design process. Then, the development process is split into multiple parts, each addressing a growing number of requirements. The result of every stage should be a functional prototype, that will be used to validate and adapt the requirements for the following development stages. All prototypes should build upon each other.

3.4. Combined Methodology

This thesis will combine all three described methodologies in order to aim for the best possible outcome in the limited time available for this thesis. The SAAM methodology is used early on to describe scenarios for the Logistics Platform, which can be used to evaluate different design approaches in Section 6.1 on page 33. Additionally, a requirements analysis will be performed in Chapter 5 on page 23 in order to define all technical implications of the scenarios and to enable the incremental software development process, which will be followed in Chapter 7 on page 69. Figure 3.1 on the next page illustrates the combined methodology used in this thesis.

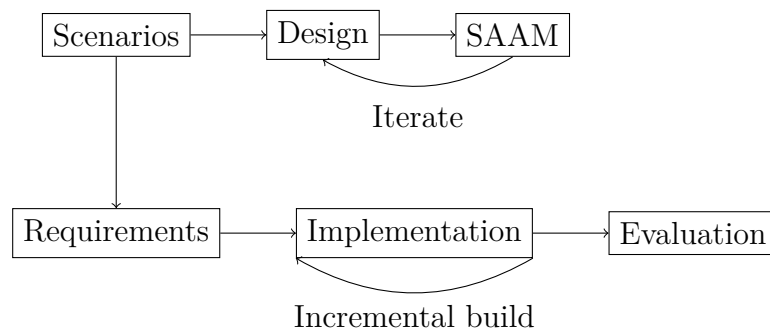


Figure 3.1.: The combined Software Engineering methodology consisting of the Software Architecture Analysis Method (SAAM), requirements engineering and the incremental software development process.

4. Scenarios for the Logistics Platform

In this chapter scenarios will be developed, which will be utilized to evaluate and compare different architectural design approaches in Chapter 6 on page 33 using the SAAM methodology introduced in Section 3.1 on page 11. Furthermore, these scenarios will be used to perform a requirements analysis in Chapter 5 on page 23, which will drive the incremental software development process during Chapter 7 on page 69.

As mentioned before, this thesis will use a real business case containing an logistical workflow involving multiple companies, which will be described in Section 4.1 on the following page. This business case will also act as the primary scenario for the Logistics Platform, which is also used to later evaluate the implemented prototype in Chapter 8 on page 77. Additional scenarios will focus on operational aspects of the future Logistics Platform, which involves the platform operator as well as companies who are participating in shared workflows hosted on the platform. Those scenarios will be introduced in the later sections of this chapter.

(Kazman et al., 1996) suggest to include three different kinds of quality attributes in scenarios used for the SAAM methodology to achieve the most valuable results:

- Quality attributes describing the output of the system as the result of specific input (e.g. correctness, security, reliability and availability).
- Quality attributes describing the activities of development and operational teams (e.g. maintainability, portability, adaptability and scalability).
- Quality attributes describing the activities of particular users (including other systems) of a system (e.g. ease of use, predictability and learnability).

Those quality attributes were taken into consideration for the scenarios described in this thesis.

4.1. Exemplary Logistical Process

The following logistical process is based on a real business case, which was provided by (Leucker, 2019). It addresses the problem of tracking freight during a shipment process, which involves multiple port authorities, a shipping company and a commercial client, who is operating multiple factories across Europe. As those companies are currently not sharing IT infrastructure, information exchange is very limited and results in negative consequences for the commercial client. This business case showcases the impact of a supply-chain bottleneck, which is very common today and the result of a lack of interoperability between stakeholders within the logistics domain.

The mentioned commercial client of the shipping process is a car manufacturer operating multiple factories in Europe, which all fulfill their own business goals within the production of different car models. One of the factories focused on in the use case is located in Estonia and responsible for assembling car components, which are produced at other factories, to the complete cars, which are then shipped to car sellers across the continent.

As the car manufacturer offers many different car models and the components for those models originate from different factories, creating an assembly schedule for the Estonian factory is a challenging task. One of the biggest problems in the process of creating an efficient schedule is the lack of detailed tracking information, when components are shipped through inland-waterways, which is in some cases the only option available. The car manufacturer can only estimate the arrival dates, because there are no updates on the delivery status between the moment that components are picked up at a factory and the moment they arrive at the destination port in Estonia. Those estimated dates often account for multiple days of delay to ensure, that all components will arrive in time. When the estimates were too optimistic, cars are assembled as far as possible and then removed from the assembly halls until the missing components finally arrive. As many employees of the Estonian factory are affected by this uncertainty in their order-based job, the current situation is far from perfect.

The logistical process used throughout this thesis will focus on frequent shipments of car components produced by a factory in Finland to the Estonian factory. This shipping process involves both the Finish and Estonian factory, one shipping company and port authorities in Finland and Estonia. All five actors and their role in the logistical process are outlined in the following sections. Figure 4.1 on page 19 visualizes the logistical process in BPMN notation:

The factory in Estonia is assembling complete cars and relies on components being shipped from other factories, in order to fulfill their business goals. They create assembly schedules based on all tracking information they can acquire which, due to the lack of interoperability, often results in inefficient schedules or production halts. In the logistical shipping process the role of the Estonian factory is rather passive, because they are only waiting to receive their expected delivery and all information connected to it.

The factory in Finland produces some of the components required by the Estonian factory. When said components have been produced and are ready for delivery, the Finish factory will contact the shipping company in order to close a shipping contract. They will prepare the components for pickup, which means that they are packed into containers, which will then be fetched by workers of the shipping company via trucks. After the containers were handed to the shipping company and the Estonian factory was notified, the factory in Finland has completed their tasks within the logistical process.

The shipping company is responsible for the whole shipping process, which includes road transportation and transportation through inland-waterways. They own and operate the trucks and ships used to transport the containers between different locations. They rely on the infrastructure provided by port authorities to dock their ship, which means, that they receive a docking schedule containing information on where and when they are allowed to dock their ship. This process has to be coordinated with their truck divers, who fetch the containers of their clients to bring them to the Finish port. They in return receive information on where and when to arrive at the harbor so that port operators can move the containers from the trucks onto the ship via cranes¹. The shipping company will also receive a docking schedule from the Estonian port authorities, so it has to get there in time. There, workers will move the containers onto trucks of the shipping company, which will then be used to transport the containers to the factory in Estonia. Only then, their tasks inside the shipping process are fulfilled.

The port authorities in Finland instruct the shipping company on where and when to dock their ship at their harbor, after they receive an initial request from the shipping company. After the containers were transported to the drop-off zone at the Finish harbor, their workers will move them onto the ship via cranes. After

¹In this scenario the trucks will drop off the container at the port, from where port operators will load them onto the ship. A common alternative to this is that trucks drive onto a ship and are moved as a whole, which eliminates loading and unloading containers via cranes.

4. Scenarios for the Logistics Platform

the ship was un-docked and has departed from the harbors properties, the Finish port authorities completed their role in the logistical process.

The port authorities in Estonia give instructions to the shipping company on where and when to dock their ship. The docking time frame for the ship as well as instructions for the truck drivers are dictated by the port authorities. When the ship has docked, workers at the Estonian port will unload the containers from the ship via cranes to certain pick-up zones for the truck drivers, who will then move them to the Estonian factory.

The described workflow simplifies the complexity of this logistical process by focusing on the interorganizational aspects of it while leaving out a lot of the internal processes of the stakeholders. After all, the Logistics Platform will focus on those parts of workflows rather than operate company internal workflows as well. (Queßeleit, 2020) provides a more detailed analysis of logistical processes at European harbors and the impact of a shared Logistics Platform, which he performed contemporaneous to this thesis.

Still, the logistical process described in this section relies on the interplay of five different companies, which all have their own IT infrastructure. In order for the Estonian factory to create efficient assembly schedules, tracking information is required, which can only be obtained when all actors contribute to a shared information system. As freight tracking is a very common use case, a platform enabling tracking related information exchange in a standardized fashion offers a practical demonstration on the positive impacts of interoperability.

The following sections describe additional scenarios focusing on different aspects of the Logistics Platform from the perspective of the platform operator as well as those companies which will later use it to participate in shared workflows.

4.2. Controlling Access to Information

A company willing to contribute workflow-related information to the platform should be able to determine who will receive which parts of the information and for what reasons. In case of the delivery process described in Section 4.1 on page 16 not every part of every step is of interest for all named actors. For example port authorities are not required to know of the exact contents of the shipping contract between the customer and the shipping company. The customer in return does not need

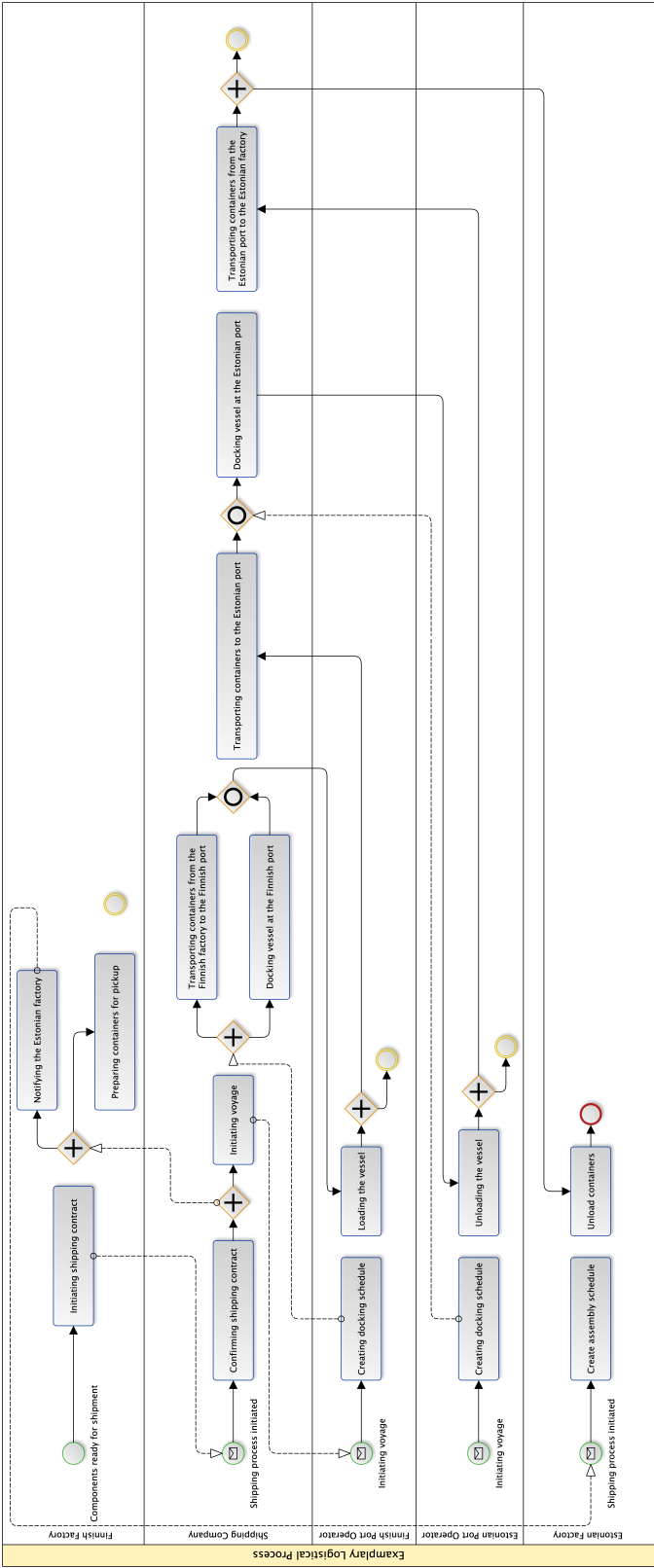


Figure 4.1.: BPMN visualization of the exemplary logistical process of transporting freight from one car manufacturers factory in Finland to another factory located in Estonia.

to know the details of the docking schedules provided from the port authorities to the shipping company. Therefore, control mechanisms should ensure that only the essential parts of workflow related information will be delivered to certain stakeholders. All access to information should be controlled and documented in order to prevent information from being leaked to unconcerned others.

4.3. Integrating a new Actor

The Logistics Platform should be able to integrate new actors without causing negative effects to the platform and other stakeholders involved. Access to the Logistics Platform should generally be limited, so that every actor needs to agree to the terms of service before gaining access to certain parts of the platform. In order to create a high level of trust between all stakeholders operating shared workflows on the platform, access to all workflows should always follow a white-listing principle.

Still, joining the platform should be as simple as possible even for companies that do not have large IT departments. It should also be possible to connect existing information systems to the Logistics Platform, so a company does not need to completely rebuild their IT just to participate. As the Logistics Platform targets a variety of different companies the landscape of information systems, that will potentially be connected to the platform, is diverse, which requires a high level of flexibility and a high number of interfaces to the platform.

One possible scenario where a new actor wants to join the Logistics Platform could be a shipping company, which wants to offer advanced tracking information to their customers. When they are first granted access to the platform they should not be able to directly participate in existing workflows, as the conditions for information exchange must first be negotiated with all other actors involved in a shipping process.

4.4. Extending an existing Workflow

As logistical processes might change over the time, the Logistics Platform must also be ready to adapt to changing workflows and because workflows operated on the platform should involve different companies, changes made by one company must not result in negative effects to any other participating company. In order to prevent such negative effects, all updates to workflow related components should

remain backwards-compatible, for example by only allowing additive changes to data schemes. When an update will break existing functionality in workflows, the update must be performed in an coordinated effort involving all actors participating.

As described in the previous scenario, two likely concerns from a companies perspective are data ownership and distribution of information. When a workflow is extended, for example when a new company is integrated, this could likely result in changes to the information flow. Every company participating in a workflow must know and agree at any time, which parties will receive which parts of the information they provide on the platform. This means that adapted workflows resulting in changes in the information flow must be communicated with all affected stakeholders.

In the described case that a shipping company will join the platform to digitally cooperate with port authorities, their customers and other stakeholders, all of those companies have to agree before information is shared with the shipping company.

4.5. Updating an existing Component

The platform should ensure that updates to the platform will not result in any negative impact to any company connected to the platform. Therefore, safety and security mechanisms should be implemented, to examine the impact of any update in advance before applying it to the Logistics Platform. Such mechanisms should not only validate those parts of the platform, that are developed and maintained by the platform provider, but also those provided by other companies.

As logistical processes can be operated around the clock it is generally difficult to schedule longer maintenance downtimes, where larger parts of the system could be updated at once. Also, as there are many different companies involved, it would be difficult to determine a time frame that suit every company. It should be possible for every company and every team working on the Logistics Platform to autonomously deploy updates to their components without causing a global outage of the system. In the case that a company will perform an update to their own internal IT infrastructure, which is possibly connected to the platform, they should be able to do so, without having to coordinate with every other company involved on the platform. The Logistics Platform must therefore provide mechanisms which compensate eventual downtime of some components without causing a chain of negative effects affecting the whole system.

4.6. Summary

This chapter introduced a logistical process which is based on a real business case of multiple companies cooperating throughout Europe where a lack of interoperability results in a negative impact on one business. Such logistical process is only one example for a workflow, that could potentially be operated on a shared Logistics Platform. Other scenarios focused on operational aspects, which were not directly addressed in the logistical process itself. Those scenarios described how the new Logistics Platform needs to behave when stakeholders and workflows change over the time to enable a high level of safety, security and trust between all stakeholders.

All scenarios described in this chapter will be used to define technological requirements manifesting their explicit and implicit content in Chapter 5 on the facing page. Those requirements will later be used to drive the implementation process followed in Chapter 7 on page 69. Furthermore, as described in Chapter 3 on page 11, the SAAM methodology will be use the scenarios of this chapter to compare and evaluate different architecture designs in Section 6.1 on page 33. The logistical process described in Section 4.1 on page 16 will be implemented and deployed in Chapter 8 on page 77 to evaluate the platform design and the prototypic implementation.

5. Requirements Analysis

In Chapter 4 on page 15 a number of scenarios for the Logistics Platform were introduced and explained. This chapter contains a requirements analysis in order to create a technical specification for the software system which is referred to as the Logistics Platform throughout this thesis. The requirements defined in this chapter manifest the explicit and implicit content of the scenarios of Chapter 4 on page 15 in a clear and technical language.

According to the principles of requirements engineering (Kotonya & Sommerville, 1998) all requirements can be grouped into functional and non-functional requirements (Chung, Nixon, Yu, & Mylopoulos, 2012). Functional requirements describe specific features of a software system that need to be considered or avoided in the system design and implementation. Those requirements therefore focus on a systems behavior rather than its operational costs, which are described by non-functional requirements. As the Logistics Platform itself will be used to host and operate logistical workflows, some functional requirements will focus on operational aspects for these workflows.

The functional requirements for the Logistics Platform are clustered into different aspects, which loosely derivate from the scenarios described in the previous chapter. This is merely an instrument to make this chapter easier to read, so the order of sections and requirements in this chapter do not reflect on their importance to the platform. Instead, the level of importance of every requirement is described by one of the key words *must*, *must not*, *should*, *should not* or *may* as suggested by (Bradner, 1997).

5.1. Information Exchange Between Companies

As the logistical workflows operated on the Logistics Platform require cooperation between different companies, said companies need to exchange information over the platform in order to participate in distributed workflows. This information exchange might contain sensible information, so every company that is sharing information

5. Requirements Analysis

Req. No	Description
R001	The platform <i>must</i> provide mechanisms allowing information exchange between companies participating in shared workflows.
R002	The platform <i>must</i> standardize workflow related information exchange in order to enable interoperability between companies fulfilling similar tasks.
R003	The platform <i>must</i> provide mechanisms that allow a company to limit access to information it provides to the platform.
R004	The platform <i>should</i> ensure that information does not breach the limitation rules defined by a company sharing information.
R005	The platform <i>must</i> log all access to information shared by a company in order to detect breaches and initiate necessary counter measures.
R006	The platform <i>must</i> encrypt all information which is processed on the platform.

Table 5.1.: Requirements for information exchange between companies.

must be able to determine who should receive it. Table A.1 on page 89 lists all functional requirements related to information exchange between companies cooperating on the platform.

5.2. Deploying Components to the Platform

In order to fulfill their tasks in a shared workflow, a company needs to deploy their own software components, which might be connected to other IT infrastructure within their company. As those software components are developed by different teams at multiple companies they need to be deployable autonomously. (Kim, Debois, Willis, & Humble, 2016, pp. 126-193) list and explain four important aspects for creating safe deployment strategies, that can also be automated to a high degree:

- Deployment pipelines
- Automated testing
- Continuous integration
- Automated low-risk releases

Req. No	Description
R007	The platform <i>should</i> provide configuration files to setup production like environments for teams developing software for the platform.
R008	The platform <i>should</i> contain a Git repository holding all configuration files necessary for creating a production like environment.
R009	The platform <i>should</i> only allow deploying software through the standardized deployment pipeline.

Table 5.2.: Functional requirements for deployment pipelines.

Requirements concerning the deployment of components to the Logistics Platform are grouped according to those aspects, which will be examined in the following sub-sections.

5.2.1. Deployment Pipelines

A deployment pipeline allows to standardize the process of updating and deploying software. This pipeline should guide developers and operators throughout the whole process in order to optimize flow and detect bugs as early as possible. To achieve this in the context of the Logistics Platform, multiple things have to be considered. Table 5.2 contains a list of requirements, which are linked to deployment pipelines for the platform.

5.2.2. Automated Testing

Software must always be tested thoroughly before being deployed to the Logistics Platform in order to prevent internal bugs or broken interfaces. Deployment pipelines can integrate automated test suites, that validate the functionality of software, before it is rolled out. Such test suites should perform as many tests as possible at an early stage of the deployment process and should be extended whenever contained tests failed to detect a bug in order to continuously raise safety and security of the deployment pipeline. Table 5.3 on the following page contains the requirements related to automated testing inside the deployment pipelines.

5.2.3. Continuous Integration

The principles of continuous integration (CI) suggest that the different parts of software should not be developed independently for longer time periods, because this usually results in infrequent and complex merges. Such merges might contain

5. Requirements Analysis

Req. No	Description
R010	The platform <i>should</i> run automated test suites against new version of software operated on the platform.
R011	The platform <i>should</i> include all those test suites into the deployment pipeline automatically rejecting deployments that fail mandatory tests.
R012	The platform <i>should</i> ensure that no software can be deployed that breaks the functionality of other components.

Table 5.3.: Functional requirements for automated testing.

Req. No	Description
R013	The platform <i>should</i> allow frequent deployment of software components.
R014	The platform <i>should</i> automatically integrate updates made by different developers and teams on a frequent basis.
R015	The platform <i>should</i> provide tools supporting CI.

Table 5.4.: Functional requirements for continuous integration.

conflicts between changes made by different teams, that could require manual intervention. After every merge containing smaller updates or even completely new features, the software needs to be tested thoroughly again, as the combination of all changes might introduce new bugs.

CI suggests to perform merges of the whole software very frequently, for example on a daily basis. The work of individual developers and small teams could then be merged together on the afternoon, resulting in much smaller and more frequent merges, faster feedback loops and a lot more practice regarding integration.

In the context of the Logistics Platform, which consist of software developed by multiple independent teams, CI could be applied by allowing and encouraging the deployment of those modules on a frequent basis. This would result in more frequent updates to the platform, but also reduce the size and potential impact of those updates. Requirements regarding CI can be found in Table 5.4.

5.2.4. Automated Low-Risk Releases

In order to encourage CI, every team should be able to deploy minor changes to their software without any manual intervention by another party. Of course updates that

Req. No	Description
R016	The platform <i>should</i> allow automated deployments of software which passes all automated tests inside the deployment pipeline.
R017	The platform <i>should</i> allow automated deployments of software when no interfaces were changed in a non backward-compatible way.
R018	The platform <i>should</i> automatically perform tests which evaluate if the interfaces of a software component were changed.

Table 5.5.: Functional requirements for automated low-risk releases.

are likely to affect other modules, like changing interfaces in non-backward compatible ways, should not be possible without manual approval and testing performed by the platform operator and the other stakeholder involved¹. Requirements regarding automated low-risk releases are listed in Table 5.5.

5.3. Monitoring

Monitoring is an essential part of the Logistics Platform. As multiple logistical workflows involving different companies will be operated on the platform, it is important to track the progress of all workflow instances. This enables the platform operator to detect abnormalities or long-term bottlenecks in the processes, that might need further optimization. Also, the companies involved in a workflow instance should be able to track the progress and potential abnormalities in their processes. According to (Kim et al., 2016, p. 208) the metrics of distributed systems can be demarcated into five levels:

- Business level
- Application level
- Infrastructure level
- Client software level
- Deployment pipeline level

The following sub-sections define requirements for monitoring the Logistics Platform according to those five levels.

¹In Chapter 6 on page 33 some ways of making major changes to software components are addressed, that can still be automated to a high degree.

5. Requirements Analysis

Req. No	Description
R019	A component responsible for fulfilling steps of a workflow <i>must</i> provide metrics allowing the platform operator to track the progress of all workflow instances.
R020	The component initiating a workflow <i>must</i> provide a unique identification code for the workflow instance.
R021	A component responsible for fulfilling steps in a workflow <i>must</i> include the unique workflow instance identification code in all communication related to a workflow instance.
R022	The platform operator <i>must</i> be provided with metrics allowing him to track all workflow instances operated on the platform.
R023	The platform <i>must</i> provide tools to the platform operator that visualize the information flow between components that run a shared workflow.
R024	The platform <i>should</i> provide tools that allow all actors to see who has access to the information they share on the platform.
R025	The platform <i>should</i> detect abnormalities inside individual workflow instances.
R026	The platform <i>should</i> automatically inform the platform operator about detected abnormalities.

Table 5.6.: Functional requirements for monitoring the business layer of the Logistics Platform.

5.3.1. Business Level

At business level abstract business goals are tracked using a number of metrics. In this thesis the business layer includes all information related to the logistical workflows and workflow instances operated on the platform. At all times the platform operator should be able to gain an overview on all the workflows currently operated on the platform. Additionally all companies involved in certain workflow instances should be able to track their progress. Table 5.6 includes all requirements concerning monitoring the Logistics Platform on the business level.

5.3.2. Application Level

Metrics on the application level address the internal performance of all components operated on the Logistics Platform. This is useful to identify performance issues during production times, allowing to respond quickly by rolling back a bad performing update or adjust scaling mechanisms for the component. Especially internal failures

Req. No	Description
R027	Every component on the Logistics Platform <i>must</i> write internal failures to a log that can be accessed by the platform provider.
R028	Every component on the Logistics Platform <i>should</i> provide additional metrics (e.g. transaction times) allowing the platform provider to monitor the performance of the application.
R029	The platform <i>should</i> provide a monitoring dashboard for the platform operator containing application layer metrics of all components on the platform.
R030	The platform <i>should</i> provided a monitoring dashboard for every company cooperating on the platform containing application layer metrics of all their components on the platform.
R031	The monitoring dashboard <i>should</i> enable filtering mechanisms to isolate certain metrics and / or components.

Table 5.7.: Functional requirements for monitoring the application layer of the Logistics Platform.

of components should be shared with the platform operator in order to plan an appropriate response. Possibly a failure inside one component might cause a workflow instance from being further processed on the platform, which should be identified as soon as possible. Also failures inside a component in production indicate that the automated testing tools inside the deployment pipeline need to be extended, in order to catch similar failures in future updates. In Table 5.7 requirements for monitoring components on the application layer are listed.

5.3.3. Infrastructure Level

On the infrastructure level metrics like CPU load, RAM usage, I/O operations on disk and network traffic should be tracked in order to prevent failures on the application and business level caused by fraud or otherwise insufficient infrastructure. Consequently monitoring this information over the time helps to identify the steps necessary to sustain higher loads in the future and fulfill availability goals. Carefully monitoring the performance of the infrastructure can help setting up efficient scaling mechanisms, which can result in less operational costs than providing an over-sized infrastructure, that often runs idle. All requirements related to monitoring the infrastructure level of the Logistics Platform can be found in Table 5.8 on the following page.

5. Requirements Analysis

Req. No	Description
R032	The CPU load, RAM usage, I/O disk operations and network traffic of all servers used to operate the Logistics Platform <i>should</i> be monitored in order to setup efficient infrastructure.
R033	Metrics collected at infrastructure level <i>should</i> be accessible by the platform operator on a central monitoring dashboard.
R034	The monitoring dashboard <i>should</i> allow to setup alerts related to individual or a group of metrics.

Table 5.8.: Functional requirements for monitoring the infrastructure layer of the Logistics Platform.

Req. No	Description
R035	The platform <i>should</i> provide APIs allowing client software to feedback performance metrics.
R036	The platform provider <i>should</i> be able to access the performance metrics provided by all client software.
R037	The performance information of client software <i>should</i> be accessible by the company who owns the software.
R038	The monitoring dashboard <i>should</i> visualize performance metrics provided by client software.
R039	The monitoring dashboard <i>should</i> provide functionalities to filter client software performance metrics.
R040	The monitoring dashboard <i>should</i> allow to setup alerts concerning the performance of client software.

Table 5.9.: Functional requirements for monitoring the client software layer of the Logistics Platform.

5.3.4. Client Software Level

Monitoring the performance on the client side allows to measure transaction times for operations performed on the Logistics Platform. In case of the platform the *client side* includes all application programmable interfaces (APIs), which are consumed by external components, and all user interfaces. An API can be monitored from the client side by providing a client library, which is used to consume the API. This library contains automated feedback mechanisms providing metrics to the platform operator. A user interface, for example provided by a web application, can also contain code, which is executed on a users device to send back metrics concerning the performance of the user interaction. Table 5.9 contains requirements for monitoring client software interacting with the Logistics Platform.

Req. No	Description
R041	The platform <i>should</i> gather telemetry on all deployment pipelines used by components of the Logistics Platform.
R042	The platform provider <i>should</i> be able to access the telemetry of all deployment pipelines on a central dashboard.
R043	The teams developing modules for the platform <i>should</i> be able to access the metrics measuring the performance and usage behavior of their deployment pipelines.
R044	The platform provider <i>should</i> be able to setup alerts concerning the performance and usage metrics of deployment pipelines.
R045	The teams developing modules for the platform <i>should</i> be able to setup alerts concerning the performance and usage metrics of their deployment pipelines.

Table 5.10.: Functional requirements for monitoring the deployment pipeline layer of the Logistics Platform.

5.3.5. Deployment Pipeline Level

When setting up deployment pipelines, that allow teams to deploy updates to their software autonomously, the status of those pipelines should be aggregated on a central dashboard, which can be accessed by the platform operator. This allows the operator to detect abnormalities like broken pipelines or suspicious deployment frequencies so he can intervene at any time. Gathering as much telemetry as possible on the usage and performance of such pipelines helps to optimize them further over the time. The requirements concerning the monitoring of deployment pipelines are listed in Table 5.10.

5.4. Summary

This chapter performed a thorough requirements analysis which is the base for a software system which can become the Logistics Platform in the following chapters. All requirements were grouped into different sections according to distinct aspects of the platform. Based on the scenarios described in Chapter 4 on page 15 the requirements manifest their explicit and implicit contents. The requirements will be used to develop the detailed architecture design in Chapter 6 on page 33. During the incremental software development process followed in Chapter 7 on page 69, the specification elaborated in this section will be implemented in multiple stages.

5. Requirements Analysis

Important aspects of the Logistics Platform include the safe and standardized information exchange between companies, monitoring on all layers and mechanisms allowing teams to autonomously develop and maintain their software components. An aggregated list containing all functional requirements for the Logistics Platform can be found in Section A.1 on page 87.

6. Designing the Logistics Platform

Based on the scenarios in Chapter 4 on page 15 and the requirements analysis in Chapter 5 on page 23, this chapter will evaluate the fitness of different architectural designs for the Logistics Platform and create the detailed design for the software system to be implemented in Chapter 7 on page 69. Therefore this chapter is split up into three larger sections:

- Section 6.1 will introduce and explain different architecture patterns, which can be used to build distributed software systems. Those design patterns will be evaluated individually using the SAAM methodology, which utilizes the scenarios described in Chapter 4 on page 15. At the end of this section one architecture design will be selected, that will lay the architectural foundation for the Logistics Platform.
- After selecting an appropriate system architecture, the following two sections showcase the specific design for the Logistics Platform. Section 6.2 on page 53 gives a holistic view on the software system to be developed and introduce different components used to fulfill specific tasks within the platform.
- In Section 6.3 on page 54 the components of the Logistics Platform are examined in more detail, highlighting the architectural design decisions, that still have to be made. This includes the comparison and selection of existing technologies to be integrated in the system architecture.

6.1. Comparing Architecture Patterns

In order to create a software system suitable for hosting and operating logistical workflows involving multiple organizations, the base architecture design has to be carefully selected. This thesis will utilize the SAAM methodology, which was introduced in Chapter 3 on page 11, to compare and evaluate different architectural design patterns with the help of scenarios, which describe the future use of the Logistics Platform. The scenarios for the platform were already introduced in Chapter 4 on page 15. As there are many different approaches on system architecture, that could be used for a distributed software system, the SAAM methodology allows to examine them in the context of a specific use case at an early stage in the design

process.

The architecture designs discussed in this section are a collection of popular approaches in software development as well as some of their derivations and variants introducing interesting new concepts, which could potentially match the scenarios and requirements of the Logistics Platform. Every distinct architecture pattern is examined in a new sub-section which contains and compares different variations of this pattern before the abstract architecture design is evaluated using the SAAM methodology.

Sub-section 6.1.5 on page 52 concludes this section by following the decision process, which results in the selection of an abstract architecture pattern for the Logistics Platform. As this is merely an abstract decision Section 6.2 on page 53 and Section 6.3 on page 54 describe the detailed specification for the Logistics Platform to finalize the system design.

6.1.1. Monolith

The word *Monolith* is the umbrella term used to describe a variety of different architecture designs. (Villamizar et al., 2015) defines a Monolith as “*an application with a single large code-base [...] that offers tens or hundreds of services using different interfaces such as HTML pages, Web services or/and REST services*”. Such architecture design is nowadays often considered outdated, because it was introduced many decades ago, when software was much simpler compared to the large interconnected systems of today. One of the disadvantages associated with monolithic software concerns the lack of dynamic scaling functionalities. As a monolithic software is just one large unit, it can only be scaled as a whole, which can result in inefficient use of resources, when certain parts of the software are used more frequently while others run idle most of the time. It can also be difficult to manage large monolithic applications, when multiple development and maintenance teams have to coordinate their work. This can result in large merge conflicts, complex manual integration tasks and as a result less team autonomy and infrequent releases of software.

But the generalization that a monolithic architecture is always a bad design decision is wrong, even though more modular architecture approaches are often favored nowadays. A monolithic architecture also brings a number of advantages and can outperform software following other architecture patterns. Having a single code-base for the whole software results in less interfaces between applications, which

are usually more error prone and difficult to maintain. So a monolithic architecture approach can still be a valid and good design decision today, as long as sophisticated design rules are enforced in order to mitigate possible disadvantages. There are a number of solutions for the problems of monolithic applications, which enable development and maintenance teams to work on different parts of a monolithic software at once, without causing much conflicts between individuals or teams. One popular derivation of the traditional monolithic architecture approach, the so called Layered Monolith pattern, is introduced in the following sub-section.

Layered Monolith

One way of structuring monolithic software is by defining multiple vertical layers, which separate the application logic executed between an interaction with the software from the outside and persistent operations on a database system or disk. The control flow of a layered monolith is top-to-bottom, meaning that user or application interaction at the topmost layer is passed throughout all the underlying layers before eventually database operations are performed and information is passed back through the layers back to the user or the interacting application. There is no general rule on the number and exact composition of the layers in a Layered Monolith, as this highly depends on the usecase for the software. A popular four layer composition of a Layered Monolith consists of the *Presentation Layer*, the *Business Layer*, the *Persistence Layer* and the *Database Layer*:

- The *Presentation Layer* contains all user interfaces.
- The *Business Layer* includes all the logic responsible for fulfilling business goals.
- The *Persistence Layer* translates create/read/update/delete operations (CRUD-operations) on objects to uniform database queries.
- The *Database Layer* stores information persistently on disk often using relational database systems.

In most Layered Monolith's modules inside one layer are only allowed to communicate with modules on the same layer or the layer directly below. Such rules can be enforced automatically through automated tests and even at the very moment a software developer is writing code in a modern Integrated Development Environment (IDE): The vertical layers can be represented by packages and access between modules of different packages can be restricted. Of course those restrictions should always be evaluated through autonomous tests as well, as a developer could use a mis-configured IDE, but usually this can be a helpful tool to guide developers in

	Business A	Business B	Business C
Presentation layer			
Business layer			
Persistence layer			
Database layer			

Figure 6.1.: The Layered Monolith pattern separates modules vertically to allow a structured control flow. Additionally, modules might be grouped horizontally when they belong to separate business areas.

their daily work and prevent illegal access between modules early on.

A Layered Monolith can also be split up horizontally in order to group modules belonging to specific business areas. In such an architecture, calls between vertical layers are usually only allowed, when the modules are also in the same horizontal group. Communication between modules on the same vertical layer are still allowed, even when they breach horizontal boundaries. One way to introduce horizontal layers in an IDE utilizes distinct namespaces which are available for modules in all packages, which in return depict the vertical layers of the Monolith, as described earlier. Figure 6.1 shows how a Layered Monolith consisting of multiple horizontal and vertical layers could be structured.

Evaluation

In theory it is possible to design the Logistics Platform as one large monolithic application, likely following a structured approach like the Layered Monolith pattern. The topmost layer of the system would contain a number of APIs, which could be consumed by the IT systems of companies involved in the platform.

The exemplary workflow, which is described in Section 4.1 on page 16, could be hosted and operated on this platform. A monolithic application could integrate a

popular BPM framework like Camunda¹ or Activiti² in order to execute logistical workflows described in BPMN. Companies associated with a workflow can receive their own API endpoints to the Logistics Platform, where they fetch workflow related information and receive tasks, which they later mark as completed using the API as well.

Implementing a monolithic platform also enables the platform operator to control all access and the flow of information from within one application. Monitoring, especially on the business layer, is also very simple to implement, because BPM engines often contain tools for visualizing workflows out of the box. Some of them also contain monitoring features for detecting abnormalities in individual workflow instances.

Unfortunately, a monolithic approach also introduces a lot of disadvantages. Most notably the lack of dynamic scaling options can be a huge problem for the Logistics Platform on the long run. When operating only one workflow involving a small number of stakeholders, like the one described in Section 4.1 on page 16, there is not much need for dynamic scaling. But as the Logistics Platform might grow over the time to host and operate hundreds or even thousands of different workflows it has to sustain the traffic produced by all associated actors. This could result in availability issues for a monolithic platform or cause enormous operational costs in order to sustain the load at any given time. At an early stage, when the Logistics Platform is rather small, a monolithic architecture approach has a lot of advantages and could be very simple to implement, but in order to become a viable solution, which could drive maritime logistic operations throughout Europe, a more scalable architecture is recommendable.

To identify the associated company of a software consuming APIs provided by the Logistics Platform, authentication and authorization mechanisms are necessary. In a monolithic architecture this is rather simple to achieve, as those mechanisms can be integrated into the application itself. Through white listing rules the access to information could be restricted and all access to information could automatically be documented to an append-only log on disk. Authentication and authorization can be more problematic in systems which follow a more decentralized approach.

Extending or altering workflows might require new APIs for the Logistics Platform or existing ones need to be changed. In order to ensure functionality, even when changes to APIs are non-backward compatible, all APIs should include versioning mechanisms and slowly deprecate older versions over the time, so all stakeholders

¹<https://camunda.com/>

²<https://www.activiti.org>

can update their software components. As described earlier, the prejudice that monolithic software is always more difficult to develop and maintain is in reality largely dependent on the way the system is designed in detail. When strict design rules are enforced, creating new or updating existing interfaces is not necessarily more difficult in a monolithic application compared to a software following another architectural design. As a Monolith is still just one big application, updates made by different developers and teams need to be integrated and tested thoroughly before they can be deployed to production safely. When the Logistics Platform reaches a certain size, this process can become more and more complex so updates could take a long time and involve much manual work before they are completed, which makes a Monolith less dynamic compared to other architecture designs. The updating process itself might also cause long maintenance downtime, because the platform always needs to be updated as a whole. As workflows might be operated around the clock it could be difficult to find a time frame which is long enough to update the Logistics Platform without causing problems to companies involved in the platform.

The following sections will focus on more distributed approaches on system architecture, which tackle a lot of the problems commonly associated with monolithic software.

6.1.2. Service-Oriented Architecture

A service-oriented architecture (SOA) structures a system as a set of services that can fall under different ownership domains (OASIS Committee, 2012). Typically, there are three different roles within a SOA which a component might implement (Al-Khanjari, Alkindi, Al-Kindi, & Kraiem, 2015):

- *Service providers* are responsible for designing and implementing services. They specify the interface that other components can consume and publish all information related to its services to the central service registry.
- *Service consumers* rely on functionalities provided by the service of other components in order to perform their own business logic or otherwise integrate into the holistic system. Before a service consumer can access those services, it first needs to learn the location and specification, which is hosted by the central service registry.
- The *service registry* stores service descriptions and locations of all service providers in the SOA. Its address and interfaces have to be known to all service providers and service consumers in the system, so that consumers can find and connect the service provider hosting their required functionalities.

Figure 6.2 on page 41 depicts the relationship between service providers, service consumers and the service registry in a typical SOA. The SOA approach has been a very popular architecture pattern for decades and there are numerous derivations of this pattern, which further drive and extend the modularity approach. For example the service-oriented device architecture pattern (SODA) is still very similar to the original SOA approach, but removes the need for a central service registry by allowing service providers to publish their service descriptions using broadcasting and multicasting methods³. Other SOA derivations, which further differ from the traditional SOA design approach, are introduced, explained and evaluated in their own sub-sections of this chapter, as some deserve a more extensive examination. The following evaluation concerns the traditional SOA approach.

Evaluation

As the main purpose of the Logistics Platform is to enable cooperation between different organizations, it is natural to consider SOA as a potential software architecture pattern. Therefore this idea will be evaluated using the scenarios described in Chapter 4 on page 15.

Companies working on shared workflows need to exchange information in some way that is safe and transparent to everyone involved. Decentralized services could be used to achieve this kind of information exchange and introduce shared responsibility. When a company is capable of providing analog logistical services to other stakeholders, a matching virtual service could act as a digital twin. In a traditional SOA, all services would be listed in a central service registry, which enforces service descriptions to be in some standardized format. Companies providing similar services, for example because they are operating within the same business domain, should implement standardized service interfaces to drive interoperability within the Logistics Platform. This could be validated by the platform operator before a service provider is able to enlist in the central service registry. This way, the exemplary logistical workflow described in Section 4.1 on page 16 could be implemented in a way, that also works in other similar business cases involving different companies.

One downside of the traditional SOA approach is, that monitoring the platform as a whole is rather difficult, because, except for the initial service binding involving the central service registry, all communication happens directly between service consumers and service providers, which are owned by different stakeholders. In order to

³The ISP was part of the OR.NET research project which resulted in the IEEE 11073-SDC standard family, which defines a SODA, that is fit for medical contexts (Kasparick et al., 2018).

6. Designing the Logistics Platform

overcome this problem, all service providers could be forced to include standardized monitoring services to the platform provider, in order to gather monitoring metrics on the business and application layer. This comes at higher development and maintenance costs for the companies involved in the platform and is difficult to validate by the platform operator, when the platform consists of potentially thousands of services.

Another problem with the traditional SOA design is the lack of centralized authentication and authorization. Authentication could be implemented and provided as an additional service by the platform operator and could be used by service providers and service consumers to validate the identity of the components they are interacting with. There are many existing authentication technologies available for such use cases, many of them working with temporal authentication tokens that a component can include in their communication and which can be validated by their communication partners.

Authorization would probably be carried out by the service providers themselves in order to control which service consumers and therefore which companies are allowed to consume their functionalities. This in return also causes more effort for all stakeholders cooperating on the platform, many of them not having large IT departments or a team of software developers at hand.

Integrating a new company to the platform could be achieved easily within a SOA, because it does not require changes to any of the existing components inside the system. As authorization of service consumers lies in the responsibility of a service provider, a component introduced by a new company should only be able to access information available for the public, but ensuring that a new actor has no access to restricted services can not be achieved by the platform operator. When a service consumer is allowed to consume the interfaces of a service provider, this has to be implemented by the service provider.

Updating components on the platform is a bit easier in a SOA compared to a monolithic architecture, as all components are applications which are operated on their own. Temporal downtime of service providers could be compensated by implementing mechanisms on the service consumer side. When a service consumer loses the connection to a service provider it could try to fall-back on other service providers enlisted in the service registry or try to reconnect after some time has passed. This way, a company updating a service provider could deploy the updated version to the platform, enlist it in the service registry and then shutdown the old version of the service, so that all consumers will connect to the updated service. This could delay some workflows, but in many cases has only a small local impact to the platform, so other workflows can be operated without any interference. When service interfaces change in a way that breaks existing service consumers, this can have a larger local

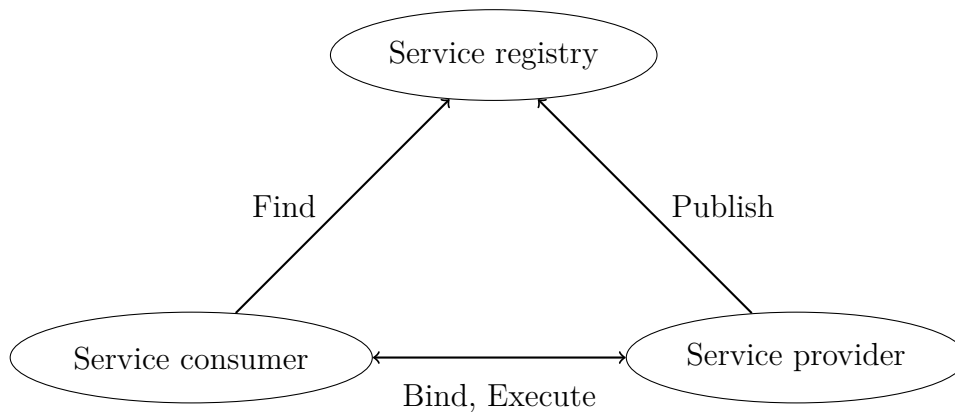


Figure 6.2.: The relationship between service providers, consumers and the registry in a service-oriented architecture (Jana, 2006).

impact, because some workflows might get stuck in an incomplete state. Therefore, standardized service interfaces should be used where ever possible to prevent such outages. A traditional SOA does not natively provide such mechanisms, but there are ways to implement this, some of which are introduced in the following sections.

Ideally every module in a SOA has a minimalistic role in the holistic system, so that updates can only cause very limited and traceable effects, but the SOA principle per se does not enforce components to be atomic in that manner. Every stakeholder could use a monolithic application, which includes all of their service providers and service consumers allowing them to cooperate through the Logistics Platform. Such distributed Monolith comes at the cost of all the negative effects, which were explained in Sub-section 6.1.1 on page 34 and should therefore be avoided. A traditional SOA distributes business logic and responsibility to all the stakeholders, which develop and operate service providers and service consumers, which makes operating the platform itself rather easy at the cost of more effort for all other stakeholders. As joining and using the Logistics Platform should be as simple as possible for any company operating in the maritime logistics sector, this is not an ideal solution. Sub-section 6.1.3 and Sub-section 6.1.4 on page 48 examine SOA derivations, that are more suited as a base architecture design for the Logistics Platform, because they address those problems.

6.1.3. Microservice Architecture

Since the Logistics Platform will be used by many different companies for a variety of workflow-related use cases, the platform needs to be dynamic in order to support

6. Designing the Logistics Platform

all of those use cases and adapt to changed logistical processes in due time. Modules used to drive the workflows on the platform will not only be developed and operated by the platform provider, but also by the companies directly involved in the workflows. Such modules might be connected with existing IT infrastructure operated by the stakeholders in order to enable seamless integration with internal workflows where ever possible.

The microservice architecture pattern is a newer derivation of SOA which could match the described use case of a Logistics Platform. While the term *Microservice* has no clear origin or unified definition⁴, one popular characterization of microservices that most industry professionals agree with was written by (Lewis & Fowler, 2014). He describes that microservices are “*an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies*”. A microservice can include custom APIs, user interfaces and even its own database system to persistently store information in an isolated environment within a complex system consisting of many other microservices. Usually microservices are designed to be stateless, meaning that all persistent information is stored externally, which allows to create, duplicate or destroy a microservice at will. (Richardson, 2017, pp. 8-11) further explain the concept of microservices by utilizing the *scale cube* which was developed by (Abbott & Fisher, 2015). The *scale cube*, as depicted in Figure 6.3 on the facing page, introduces three dimensions of scaling for computer software:

- X: Cloning identical instances of the application and evenly distributing all incoming requests. Throughout the literature this concept is commonly referred to as *horizontal scaling*.
- Y: Decomposing an application into microservices.
- Z: Splitting up responsibility between multiple instances of an application based on request parameters.

While scaling an application along the X and Z axis can improve the availability of an application, growing development and maintenance costs will not be solved by scaling along theses axes. Those problems can instead be addressed by scaling along the Y axis by decomposing a large application into atomic microservices.

⁴(Rodgers, 2005) was the first person to speak of *micro (web-)services* during a conference talk, but the principles behind *microservices* are based on the UNIX philosophy first documented by (McIlroy, Pinson, & Tague, 1987).

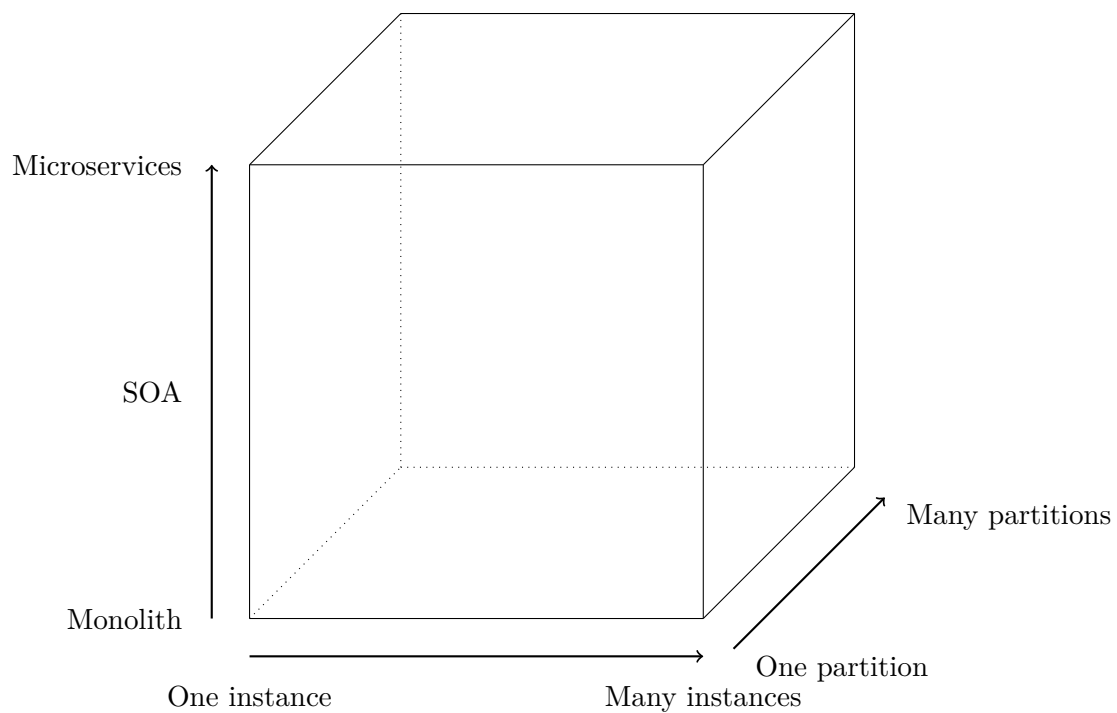


Figure 6.3.: The *scale cube*, as depicted in (Abbott & Fisher, 2015), defines three dimensions to scale an application: Balancing load across multiple identical instances of an application (X); Functionally decomposing an application into a set of microservices (Y); Balancing load across across multiple instance based on request parameters (Z).

6. Designing the Logistics Platform

	one to one	one to many
Synchronous	Request-response	-
Asynchronous	Asynchronous request-response	Publish-subscribe
	One-way notifications	Publish-async responses

Table 6.1.: Communication patterns categorized in two dimensions according to (Richardson, 2017, pp. 67-68).

According to (Richardson, 2017, pp. 14-17) the microservice architecture pattern offers a variety of benefits:

- Continuous delivery and deployment of large complex applications
- Services are small and easy to maintain
- Services can be deployed independently
- Services can be scaled independently
- Autonomous teams
- New technologies can be adapted more easily
- Fault isolation

Those benefits directly address a number of requirements, especially those defined in Section 5.2 on page 24. As the microservice architecture permits a number of different communication patterns, those patterns will be further examined and compared in the following sub-section.

Communication Patterns

(Richardson, 2017, pp. 67-68) characterizes communication patterns for interprocess communication (IPC) in two dimensions as depicted in Table 6.1. The first dimension describes whether the communication occurs between two components only, or if there might be multiple recipients at once. On the other dimension, communication patterns are separated according to synchronicity or asynchronicity. In a synchronous communication, a component awaits a response to a request within a predetermined time period. Until the response is resolved or the time period expires, the component is waiting in a blocked state, whereas an asynchronous communication is non-blocking and does not require an immediate response. The request-response and the publish-subscribe pattern will be further explained, as they are both commonly used in a microservice architecture.

Request-Response Communication

Many communication protocols follow the request-response principle. This principle works as follows: When a component requires information provided by another component, it sends a request asking for it. This request will then be evaluated by the hosting component and a response will be returned, which either contains the requested information or an error message. The Hypertext Transfer Protocol (HTTP) (R. Fielding et al., 1999) is a popular protocol following this communication pattern, which is used to access most information on the internet today. It is also used in many microservice architectures to exchange information between services. Many APIs today combine HTTP and the Representational State Transfer (REST) paradigm (R. T. Fielding & Taylor, 2000) in order to implement APIs. Information is then often formatted in some standardized format like the Extensible Markup Language (XML) (W3C, 2008) or the JavaScript Object Notation (JSON) (Bray, 2017). The advantage of such protocol stack is that it can be interpreted by humans (more specifically IT specialists) as well as computer programs. Request-response can also be implemented using binary message formats like the language neutral gRPC which uses HTTP/2 which can only be interpreted by computer programs, but might be more efficient in some scenarios compared to a REST-API.

In a request-response communication, the information flow is controlled by service consumers, as they have to explicitly request information, before a service provider will respond. This makes the role of a service provider rather passive in the request-response paradigm. As a result, some functionalities like a notification service are not as easy to implement when following this communication pattern. Request-driven communication also requires service consumers to know the address and interface of the services that they need to consume. This can result in a tightly coupled system, where changes or outages of one or more services might cause cascading errors, which are hard to predict in a large interconnected system. At a certain size, a tightly coupled microservice architecture can be hard to maintain, because the impact of an unavailable component or an updated API could have unexpected side effects and could even bring the whole system down.

(Fowler, 2014) and (Nygard, 2017) introduce the circuit breaker pattern to address this problem. They suggest to wrap all calls to remote services in circuit breaker objects. Those are monitoring the failure rate of all requests and at some point *break the circuit*, resulting in the execution of local fallback code during the outage. Circuit breaker objects are usually connected to a central monitoring dashboard and can also contain functionalities, which automatically reopen the circuit, when a

6. Designing the Logistics Platform

remote service is available again. Netflix⁵, a company streaming media to millions of customers every day, implements this pattern in their request-response driven microservice architecture and even introduces artificial outages to their production system in order to ensure high availability and robustness even when certain services are temporarily unavailable (Christensen, 2012).

As depicted in Table 6.1 on page 44, the request-response pattern is used for one-to-one communication that is in many cases also synchronous. Today many applications and programming languages prefer the use of asynchronous calls wherever possible, because it usually performs better, as synchronous communication can temporarily bring an application to a blocked state.

Applying the request-response paradigm to the whole Logistics Platform would require a deep shared understanding of the whole system by every stakeholder and the installation of safety nets like the circuit-breaker pattern introduced by Netflix. While Netflix operates a complex interconnected system of microservices to fulfill their business goals, they are in a position to ensure that all of their teams follow company guidelines. As the Logistics Platform will connect various different companies, which all contribute software to the platform, this would be far more complex to achieve.

Apart from that, companies involved in the platform need to be able to limit access to their services in order to control, which stakeholders receive workflow related information. Implementing authentication and authorization mechanisms is comparably easy to achieve in request-response communication, because the communication is one-to-one and every party could identify the other one using a central authentication technology provided by the platform operator. Asynchronous request-response can also be used for one-way notifications, which is a potential use case for the Logistics Platform: A service consumer sends a request to receive a notification to a service provider. Because of the asynchronicity the communication is non-blocking and does not require an immediate or any response. Therefore, the service provider could use a response to eventually send a notification to the service consumer when it is deemed necessary.

Publish-Subscribe Communication

Instead of sending information only when it is explicitly requested by a service consumer, some protocols allow service providers to actively send messages. Those

⁵<https://www.netflix.com/de/>

services, often referred to as publishers, can group messages into one or more categories in order to narrow down the number of recipients. Service consumers, in such paradigm often named subscribers, can subscribe to all messages linked to specific categories and can passively wait for all incoming messages. Publish-subscribe patterns usually involve third party technology, responsible for routing messages from publishers to subscribers. As every component is only communicating with this messaging technology, the existence of producers is unknown to the subscribers and vice versa. Because this eliminates most explicit interfaces between components, this kind of architecture is considered loosely-coupled.

Of course, loose coupling also comes at a cost: Subscribers can only perform their tasks if they receive the messages they need. If the messaging system is unavailable, the producer stops sending messages, a message is assigned another category or the message format is changed, this has a direct impact to a number of subscribers unknown to the publishers. In a publish-subscribe based communication safety mechanisms need to be implemented in order to avoid such scenarios.

In order to ensure interoperability in a loosely coupled system, following the publish-subscribe communication pattern, all messages should follow standardized schemes which also determine the linked message categories. Components, that produce or consume messages, could be tested before they are deployed in order to evaluate, if they comply to the standardized message format. This could be further extended by introducing contract-based testing to all components operated on the Logistics Platform (Ciupa & Leitner, 2005). Every publisher and subscriber has to close a contract, specifying how they have to behave in order to publish or consume certain types of messages. The fulfillment of such contracts should be ensured through autonomous and manual testing.

Evaluation

The workflow described in Section 4.1 on page 16 can be broken down into microservices, each of them responsible for one task within the workflow. As the workflow involves different stakeholders, every organization provides those microservices required to perform the tasks they are responsible for in the logistical process. Some parts of the workflow might be standardized in the future, so that interoperability between companies can be assured. This could even result in standardized microservices for certain workflow steps, which the platform provider could offer to any interested stakeholder to facilitate the process of joining the platform. Integrating new stakeholders and their services into a typical SOA can be achieved with very little effort, as mentioned in Sub-section 6.1.2 on page 38. In a microservice

architecture, which is merely a more atomic approach to SOA, this is very similar.

As the pattern used to communicate between microservices has a big impact on the system architecture, both the request-response and the publish-subscribe pattern were introduced with a focus on they could be implemented in the context of the Logistics Platform. When the request-response principle is followed, the system becomes tightly coupled, which makes it harder to maintain when it reaches a certain size. Choosing the publish-subscribe pattern results in a tightly-coupled system, which would be more flexible and could be easier to maintain.

In general, the microservice architecture suits the Logistics Platform very well, because it promotes components to fulfill a single purpose, which makes them reusable and far less complex compared to a monolithic application, that incorporates various services at once. This allows for a very dynamic system, that could quickly adapt in order to include new or changed workflows operated in the platform. Of course the problems related to a more dynamic system have to be addressed in the specific system design. The following section will further examine a microservice architecture, following the publish-subscribe pattern.

6.1.4. Event-Driven Architecture

Event-driven describes an architectural design approach for distributed software systems, which can be used as an addition to the microservice architecture. A system following this design suggests, that all communication between microservices should follow the publish-subscribe communication pattern where every message is considered an event, meaning that it describes a significant change in state (Chandy, 2009). In the context of the Logistics Platform an event could contain an update related to a workflow instance like the fulfillment of a workflow step. An event-driven architecture can follow two distinct philosophies of governance for communication flow (Butzin, Golasowski, & Timmermann, 2016):

- An *orchestrated* governance approach, meaning centralized control.
- A *choreographed* governance approach, meaning decentralized control.

Both governance approaches will be explained and examined individually before the event-driven approach is evaluated at the end of this section.

Orchestration

In an orchestrated architecture, one centralized module is responsible for controlling the communication flow between all other modules in the system. This approach enables central responsibility for workflows for example by the platform operator. The one central component distributes tasks to components provided by the stakeholders who are integrated in a shared workflow. Those components then have to report all progress related to this task back to the central component, which might trigger other components as a result in order to proceed with the workflow.

Workflow management systems are usually designed in a way, that one central component, often called *workflow engine*, is orchestrating all other components in the system to fulfill all workflows. While the workflow engine keeps track of all workflow instances, the completion of workflow steps is outsourced to other components. The workflow engine can be compared to a state machine tracking the state of all workflows within the system. It is responsible for updating the state of workflow instances and will trigger other components in order to move workflows along the predetermined paths. In order to distribute workflow tasks, the workflow engine provides APIs, which can be consumed by *task workers*. They can consume those APIs to receive new tasks and update task related information, which is an approach that could also be used in a publish-subscribe communication, by assigning each component their own message queue. Most workflow engines today use request-response based communication for their APIs, but there are also technologies which support the publish-subscribe communication pattern. The popular workflow engine Camunda⁶ can be combined with messaging technologies instead of using a REST-API (Camunda Services GmbH, 2020). Other workflow engines like Zeebe⁷ purely focus on publish-subscribe communication.

Unfortunately, orchestrated governance is not the ideal solution in a microservice architecture, as the existence of one central microservice in control contradicts the philosophy behind such architecture, because every microservice should fulfill a single atomic task, for which it is responsible on its own. Having one central microservice in control makes this service a bottleneck for the whole system, as all functionality is depending on it. Interestingly, this anti-pattern occurs very often in practice and even has its own name. (Smith, 2017; Tabbaa, 2019; Tengstrand, 2016) introduced the term *Microolith* to describe this anti-pattern, which is also commonly referred to as *Micro-Monolith*.

⁶<https://camunda.com>

⁷<https://zeebe.io/>

6. Designing the Logistics Platform

Having a central workflow engine also implies, that every workflow must be defined within this engine before it can be operated by task workers. This top-to-bottom approach limits dynamic adapting workflows and slows down the introduction or altering of workflows on the Logistics Platform. A workflow would always need to be altered in the engine first, before task workers can also adapt. The only real advantage of the centralized governance approach is, that the platform operator is in full control of the information flow and can prevent information from being distributed to the wrong stakeholders. Having multiple components in control can result in more effort to govern the platform.

Choreography

A different governance approach in an event-based system is the use of choreography, which means that the governance is highly decentralized. When taken to its extreme, all connections between components should be implicit, which results in a very loosely coupled system. In such architecture, events do not have a specific recipient. Every service can produce and subscribe to those events necessary to fulfill its own workflow tasks. This makes choreographed architectures much harder to control, as any additional, updated or removed component may cause a chain reaction of effects, that might be difficult to predict. Therefore, a choreographed system requires careful design and monitoring to verify the correctness of its behavior. In order to limit the access to information, every company could be assigned their own topic within the messaging technology, which only contains those event connected to workflows directly involving said company. The platform provider could provide microservices, which forward events between topics, in order to allow a managed control flow, which is realized through multiple components, following a decentralized approach.

Choreographic governance in the event-based microservice architecture is ultimately more favorable, because orchestration breaks many of the advantages, which were introduced by microservices (Butzin et al., 2016). This design choice implies that there is no central microservice in control of all information flow. Instead, multiple microservices coordinate the flow of events between stakeholders, resulting in a highly flexible and dynamic system. In order to compensate the challenges connected to decentralized governance, a reasoned system design is vital to the success of the Logistics Platform.

Evaluation

The described business case of a shipping process involving two factories, two port authorities and a shipping company, every step within this logistical workflow could be described by using events. Every company interested in the workflow can subscribe to workflow-related events in order to receive updates on workflow instances, which might act as a trigger for performing their own tasks within the workflow. Dedicated microservices controlled by the platform provider ensure that events will only be mediated to those companies which are allowed to receive the information contained.

Adding new actors to the platform is not a challenge in such event-driven architecture, as every company would receive their own event queue or topic. Achieved by authentication and authorization mechanisms, stakeholders can only publish and subscribe to their own topic, which initially isolates them on the platform. Additional microservices of the platform provider can mediate events between topics in order to distribute information among companies in a shared workflow. The conditions for this forwarding process to happen, could be negotiated with a stakeholder for every type of event that he wants to send or receive.

Updating or adding components to the system can also be achieved very easily. As there are no explicit interfaces between microservices, a new, updated or removed microservice can not directly impact another microservice. Still, indirect effects need to be considered and avoided. For example a message schema changed by a publisher can result in subscribers not being able to interpret the messages in the future, which might be unnoticed for some time. In order to prevent events and the related information from getting lost, three mechanisms should be implemented inside the Logistics Platform:

1. The queue / topic of a company should hold events for a predetermined time period, so no information gets lost, while a component is updated or is otherwise unavailable.
2. Every component should be able to replay events after a specific point in time, in order to safely recover after an update or system failure resulting in a crash.
3. Every component responsible for publishing a certain event type should be tested before it can be deployed, in order to ensure that the event follows a standardized schema, which other components can understand. Subscribers should also be tested to ensure, that they can correctly interpret the events, that they subscribe to.

6.1.5. Conclusion

Overall, the event-driven architecture is a good match for the Logistics Platform as the scenarios described in Chapter 4 on page 15 could be fulfilled by such architecture, when certain design aspects are considered, when the detailed architecture design is created. Those design details will be elaborated by also utilizing the requirements defined in Chapter 5 on page 23, which contain a more technical perspective on the scenarios for the Logistics Platform.

While the publish-subscribe communication by the means of events is ultimately more favorable for the platform, some IT systems used by the stakeholders, willing to cooperate through the platform, might only support request-response based interfaces like a REST-API. This problem could be addressed by developing microservices, which can translate between both communication patterns. For example, a port authority might want to fetch a ships manifest at once, rather than creating it on their own from a series of events, describing containers being loaded onto the ship. This could be achieved by a microservice, which collects and aggregates all those events within a topic and provides a REST-interface for the port authorities to consume. As this is a use case, that can be generalized, such microservice could be provided by the platform provider to every port authority.

Still, all platform internal communication should be implemented using events to truly achieve the decentralized and loosely-coupled architecture. Microservices operating on the edge of the Logistics Platform could be allowed to have other interfaces, which can be consumed from the outside. This not only includes APIs for other information systems to consume, but also user interfaces.

After the base architecture for the platform was elaborated, the next two sections will create the detailed architecture design for the whole Logistics Platform. First, Section 6.2 on the facing page will give a holistic view on the system design to demonstrate how the components contained focus on different aspects of the platform. Then, Section 6.3 on page 54 will follow the design process for those components in detail, explaining all design decisions made along the way.

6.2. Platform Architecture Overview

Before all components are described in more detail in Section 6.3 on the following page, this section will give an overview on the platform design, which means, that certain design decisions like the selection of specific technologies will be fore-shadowed. Still, it is easier to follow the detailed design process behind specific components, when the holistic platform design is introduced in advance. This section can also be used as an index to directly jump into the design process of certain components of interest.

The Logistics Platform will be the host to an unknown number of software components, so a flexible platform solution is needed. Such technology must enable deploying and operating software in a shared environment, that meets the requirements defined in Chapter 5 on page 23. One technology, which can achieve this, is the container orchestration engine (COE) Kubernetes (K8s), which provides sophisticated tools for scheduling containerized software and allows applications to safely interact inside a virtual network. The complete decision process, which ultimately lead to the selection of K8s is followed in Sub-section 6.3.1 on the following page.

An event-driven architecture approach relies on a messaging technology, which follows the publish-subscribe communication pattern. For the Logistics Platform Apache Kafka was selected, because of its distinctive feature of storing all messages persistently on disk in append-only log files instead of processing messages solely in memory. A more extensive explanation on why Kafka was selected, including comparisons with similar technologies, can be found in Sub-section 6.3.2 on page 57. Kafka can be operated within the K8s cluster used to drive all other components of the platform as well. Still, a lot of design decision are necessary to enable the choreographic governance principle for workflows, which was briefly outlined in Sub-section 6.1.4 on page 50. This has to be achieved in a way, that fulfills all requirements defined for the Logistics Platform and is described in detail in Sub-section 6.3.3 on page 59.

One aspect of the platform, which is linked to a lot of requirements in Chapter 5 on page 23, concerns monitoring. As described earlier, monitoring an interconnected software system relies on metrics being collected on multiple layers to provide a complete picture of the systems state. This includes monitoring the K8s cluster itself as well as the Kafka message brokers and all other components operated within. The collection of metrics should be automated to a high degree, so the implementation and integration effort for other stakeholders can be minimized. Monitoring related

aspects of the Logistics Platform are described in Sub-section 6.3.4 on page 64.

Unfortunately, due to the lack of time, aspects like CI and CD could not be fully completed within the architecture design. Still, Sub-section 6.3.5 on page 66 will introduce possibilities to incorporate those aspects into the platform at a later point in time, highlighting the use of GitLab CI and similar tools, which can be connected to the K8s cluster. Unlike the other parts of the system design, CI/CD will not be implemented in Chapter 7 on page 69.

6.3. Designing the Software Platform

This section introduces the different components of the Logistics Platform in more detail, following all design decisions leading to the selection of specific technologies and design aspects. As the different parts of this design build upon each other, the order of sub-section matches the different stages of the incremental software development process used in Chapter 7 on page 69.

6.3.1. Choosing a Container Orchestration Engine

The Logistics Platform has explicit requirements concerning the availability and scalability for the microservices it contains. To achieve this, the platform has to be operated on multiple servers, so when one of them fails, the platform is still available to all stakeholders. This results in the fundamental question of how the microservices can be operated efficiently across multiple servers at once.

In general there are multiple different ways of running software on a computer. The most primitive one is to simply execute a compiled application without any further means of encapsulation. This is the usual way in which software is operated on a personal computer and is still also very common on servers hosting applications consumed by multiple clients. Unfortunately this is a very simple setup that does not scale very well and also raises safety and security problems, as applications can interfere with another.

One popular technique used to overcome these problems is to use virtualization, which many use synonymously with the concept of virtual machines (VMs). VMs simulate the complete architecture of a real computer and can be used to operate software in an isolated environment. Although the use of VMs is considered very stable nowadays, there is still a large resource overhead, when virtualizing a whole

computer, instead of just those parts necessary to encapsulate applications (Villamizar et al., 2015).

Today a popular alternative to operating software within VMs is to use so called containers. This is another form of virtualization, that is more lightweight and efficient compared to VMs (Villamizar et al., 2015). Containerized software is still sharing the same kernel of their host machine, while all other required resources can be container exclusive. The concept of containerized software is surprisingly old, as it is linked many years back to the introduction of the *chroot*-command in Unix version 7 in 1979⁸. But as it was rather difficult to implement and use for decades, it only became very popular to the development community, after the first version of Docker⁹ was released in 2013. Docker allows to create containers from existing software, which can then be shared and operated using a very lightweight command line interface. Even though containers are a Linux-exclusive feature, Docker also allows to operate containers on Windows and macOS computers by running a Linux VM, which hosts all the containers.

Containers are often used in systems following the microservice architecture approach, because of their lightweight nature and because they can contain all of the dependencies required to operate the software, like specifically configured run time environments. Still, the mere use of containers does not solve the problem of operating microservices on multiple servers at once, in order to ensure high availability and scalability. This in return can be achieved by Container Orchestration Engines (COEs), which are used to create complex software systems often consisting of thousands of containers, scattered across multiple servers. The following sections introduce and compare three popular COEs, that allow to operate containerized software across multiple servers¹⁰. The main purpose of a COE is to *schedule* containers, which means, that they can start, stop, distribute and scale containerized software on multiple nodes. COEs also allow to assign storage volumes to containers and setup virtual networks, where containers can safely communicate with each other or even have an external IP address assigned.

⁸(Bernstein, 2014) follows the history of containers demonstrating, that early versions of containers can also be found in FreeBSD (1998) and its Jail feature and also Solaris 10 (2004) and its Zones feature. Containers were standardized in 2008 when Linux Containers (LXC) became a part of the standard Linux distribution.

⁹<https://www.docker.com>

¹⁰The words *node* and *server* will be used synonymously from now on

Kubernetes

Kubernetes (K8s)¹¹ is a tool originally developed by Google, which is now open source and part of the Cloud Native Computing Foundation (CNCF). K8s has a rich set of native features like auto-scaling, service discovery, load balancing, volume management and secret management. It is often operated in managed environments like Amazon Web Services (AWS) or Google Cloud Platform (GCP), but can also be setup and operated on own infrastructure, which from the experience of the author of this thesis is not recommendable as explained later in Section 7.1 on page 69.

Docker Swarm

Developed by the Docker Foundation, Docker Swarm¹² is the official COE for Docker. It is tightly integrated with the API of the Docker core, which makes it very easy to get started for people that are already experienced in Docker. Like K8s, Swarm uses YAML files to configure the scheduling of containers. Auto-scaling and load balancing currently require third-party technologies, but support for those features might be added in the future. Since the Docker Foundation officially started supporting K8s in their Enterprise Edition as well, the future of Docker Swarm is not so clear (Bohn, 2018).

Apache Mesos

Mesos¹³ works a bit differently than K8s and Swarm, as it has an even stronger focus on decentralized control. Unlike the other two COEs, Mesos allows to operate multiple master instances at once, which results in higher degrees of flexibility and availability. It also supports to operate containers using multiple container engines at the same time and can even be used to host K8s or Swarm clusters. Unfortunately Mesos is far more complex to use. While the other two technologies have a rather steep learning curve, Mesos requires a detailed understanding of the underlying technology, before it can be used to perform simple scheduling tasks for containers. In some use cases, where the features of K8s and Swarm are not sufficient, Apache Mesos could be the only real option today.

¹¹<https://kubernetes.io>

¹²<https://docs.docker.com/engine/swarm/>

¹³<https://mesos.apache.org>

Conclusion

All three technologies could be used to operate the Logistics Platform. As it is more difficult to setup and operate and it requires a lot of background knowledge, Mesos will not be used for the Logistics Platform within the scope of this thesis. The point at which the other two COEs might become insufficient is very distant, as K8s and Swarm are both technologies which are capable of powering complex systems consisting of thousands of containers.

While Docker Swarm is easier to setup and operate and it is tightly integrate with the Docker API, it contains less features compared K8s. Especially the lack of dynamic scaling and load balancing out of the box, makes K8s the better option in the long run. Also it remains in question, if the popularity of K8s within the Docker community, will result in Docker Swarm being discontinued in a few years.

6.3.2. Choosing an Event Bus

In an event-driven architecture one component is responsible for transmitting messages between all other components. This section gives an overview on popular messaging technologies following the publish-subscribe pattern and explains why Apache Kafka will be used for the Logistics Platform.

Java Message Service

The Java Message Service (JMS) (Hapner et al., 2015) is an API for Java, that allows to exchange messages between two or more clients, without any explicit coupling between them. JMS allows two distinct messaging models:

- Messages can be sent to a *queue*, which allows multiple senders but is bound to only one client.
- Messages can be sent to a *topic*, which allows multiple senders and also multiple recipients, who are subscribed to the topic.

The second messaging model is an implementation of the publish-subscribe pattern, which could be used inside an event-driven architecture. Unfortunately JMS only supports languages that operate on the Java Virtual Machine (JVM) like Java, Scala or Kotlin.

Advanced Message Queuing Protocol

The Advanced Message Queuing Protocol (AMQP) (International Organization for Standardization, 2014) is an open source message-based standard for inter-process communication (IPC). Unlike JMS, which provides a high-level API for Java applications, AMQP is a wire protocol, that specifies a binary message format, which can be implemented in most modern programming languages. The protocol also offers multiple delivery-guarantees for messages such as promises, that a message will be consumed at-most-once, at-least-once and exactly-once. AMQP can split the load between multiple AMQP brokers by defining a hierarchy of topics, which can be used for routing purposes.

Message Queue Telemetry Transport

Another open source messaging protocol is the Message Queue Telemetry Transport (MQTT) (OASIS Standard, 2019), that is very popular in the internet of things (IoT) domain, because of its lightweight implementation. MQTT follows the publish-subscribe communication pattern and allows a hierarchy of topics to which a client can subscribe and publish. Like AMQP, MQTT also offers delivery-guarantees such as delivering messages at-most-once, at-least-once and exactly-once. MQTT brokers can be configured to share the load and compensate failures of individual brokers.

Streaming Text Oriented Messaging Protocol

The Streaming Text Oriented Messaging Protocol (STOMP) (Stomp.github.io, n.d.) is a text based messaging protocol, that follows and extends the specification of HTTP. While STOMP is very easy to implement, the protocol only offers a limited set of features for sending and receiving messages, while all other functionalities must be developed from scratch.

Apache Kafka

Apache Kafka¹⁴, unlike most other message brokers, is writing messages to a persistent append-only log file on disk. Those log files keep messages for a configurable amount of time, before they are eventually deleted. In the context of the Logistics Platform this allows to track the history of events without the need for an additional

¹⁴<https://kafka.apache.org>

database. Kafka is part of the Confluent Platform¹⁵, which provides additional features like KSQL as an extension to the Kafka core. KSQL allows to create SQL-like views on event logs, which can be cached within a microservice. This approach is called *data on the outside*, meaning that data is not stored within microservices, but instead the message broker fulfills the role of a database. The philosophy behind this pattern is *event sourcing*, which claims that the current state of a system can be recreated by following all events leading up to the current point of time (Stopford, 2018). So rather than explicitly storing the current state of a shipment in a database, all events related to this shipment could be recalled in order to determine its current state. Though, KSQL as well as other features of the Confluent Platform are not distributed under an open source license. The Confluent Community License (CCL) shall prevent cloud providers to create concurring offers to the Confluent Platform (Parbel, 2018). In case of the Logistics Platform, features like KSQL could still be used, as Confluent does not sell any directly concurring product¹⁶.

Conclusion

The different messaging technologies introduced all have advantages and disadvantages. JMS does not suit the Logistics Platform, because of the lack of support for different programming languages. Not every actor involved in the platform will likely be experienced in Java or other JVM languages. While STOMP is easy to implement, it lacks many features of the other technologies shown in this section, some of which required by the Logistics Platform. While AMQP, MQTT and Apache Kafka perform very similar in benchmarking tests (Dobbelaere & Esmaili, 2017), the persistent message logs of Kafka allow microservices to recover and replay events after a critical failure, whereas the other two messaging technologies usually delete messages when they were consumed by every registered subscriber in a topic, which makes the recovery for a microservice much harder. Therefore, the Logistics Platform will use Apache Kafka for all internal event-based communication.

6.3.3. Designing the Event Structure

There are two important design aspects to be considered when designing the event-driven system, both related to the structure of events within the Logistics Platform. On the one hand it is necessary to define a uniform format for events, so that they can

¹⁵<https://www.confluent.io>

¹⁶(Confluent Inc., 2019) states that most commercial and free products are still allowed to use KSQL. The license is targeting providers that offer services, that directly compete with those offered by Confluent. A hotel booking engine, which internally uses KSQL, would still be allowed to be operated without any restrictions, as Confluent does not offer such service.

be interpreted and processed correctly by microservices operated on the platform. In addition to that, the structure of topics within Kafka also has to be specified to ensure that events are delivered to all subscribers, which have the permission to receive the information contained.

Event Format

Kafka itself does not enforce any specific format for messages. In order for a consumer to parse information within a Kafka message, a uniform data format for all events should be selected, which all components have to comply with. One popular notation for events in cloud based applications is the CloudEvents specification (Cloudevents.io, 2019a). (Cloudevents.io, 2019b) offers a detailed explanation on how CloudEvents can be used within Kafka, which will be the base for the event specification for the Logistics Platform.

As suggested by this specification, all events will follow the JSON format and contain the following list of attributes:

- **id:** A String containing a unique identifier for the event following the UUID version specification (Leach et al., 2005).
- **source:** A String containing the name of the topic where the event was first published and the producers IPv4 address inside the K8s cluster
- **specversion:** A String containing the version of the CloudEvent specification used. During the scope of this thesis the version “1.0” of the CloudEvents specification is used.
- **type:** A String specifying the type of event using the reverse domain name system (DNS) convention.
- **dataschema:** The URI of the schema description, that the data property of this event will follow.
- **subject:** A String containing a UUID of the workflow instance related to this event. This attribute is only mandatory for events directly related to a workflow.
- **time:** A time stamp of the events occurrence following the uniform time stamp format for the internet as specified in RFC 3339 (Klyne, Clearswift Corporation, Newman, & Microsystems, 2002).

- **data:** The event body. All fields within the body depend on the specific event type and linked data schema which might enforce an additional list of mandatory fields and their respective type.

Listing 6.1 shows an exemplary event, which describes that a unit was loaded onto a ship. The data property of this event follows a proposal made by (Braun, 2019) and incorporates data types currently used in the logistics domain such as the Intermodal Loading Unit (ILU) (Deutsches Institut für Normung e. V., 2011) for identifying units and the United Nations Code for Trade and Transport Locations (UN/LOCODES) (United Nations Economic Commission for Europe, 2019) for identifying locations relevant for trade and transport. This thesis will not provide event schemes related to all steps within the exemplary logistical workflow as described in Section 4.1 on page 16 and instead will just use different event types in the event headers to differentiate between events. Creating standardized event schemes for all workflow related steps would require a lot of domain knowledge and should be achieved in cooperation with different companies working within this domain in order to create a specification suitable for most use cases. Standardized schemes for all the events are highly encouraged as they lay the base for a truly inter-operable platform.

```
{
  "id" : "65b35bd3-c5e2-4f43-b309-2334da7553a7",
  "source" : "facilityA/10.244.10.10",
  "specversion" : "1.0",
  "type" : "port.vessel.unit.loaded",
  "dataschema" : "",
  "subject" : "b3293715-6eed-4c40-a5b5-8b207d944809",
  "time" : "2020-05-19T04:20:00Z",
  "data" : {
    "unitID" : "<ILU>",
    "unitType" : "",
    "unitAttributes" : [],
    "vesselID": "<IMO>",
    "vesselDestination": "<LOCODE>"
  }
}
```

Listings 6.1: Exemplary event describing that a unit was loaded onto a ship. The event follows the CloudEvent specification (Cloudevents.io, 2019a) and is based on a draft created by (Braun, 2019).

Topic Structure

As described earlier, Kafka allows to create multiple topics to separate events, where multiple producers and subscribers can be connected to one or more topics. There are also authentication and authorization mechanisms available in Kafka to create permission rules on a topic level. This allows to limit the scope, in which events are distributed among companies. Topics are also separated into multiple partitions, where events associated with one partition are always consumed by the same subscribers. Partitions are used, when multiple Kafka brokers share the load of the whole system. Replication rules can be enforced to ensure, that topics and partitions are replicated on multiple brokers. The concept of partitions is extremely useful, when multiple instances of one microservice are setup to share the load of an incoming event stream, as all events, which are linked to one subject, could be assigned by using this subject as the partition name. Following this approach, all events associated with a specific workflow instance will always be processed by the same instance of a microservice, even if many replications of the same microservices are operated at the same time.

One design concept briefly outlined before, was to assign each company their own topic, which can only be accessed by components of the company and the platform operator. This is not the only possible options on how to structure the topics within the Logistics Platform, but ensures, that information flow can be contained very easily. The idea behind this concept is, that it allows the platform operator to control the flow of information between companies. Additional to the topics assigned to the cooperating companies, one topic will act as the inner core of the platform, where all workflow related events will be forwarded to. Therefore, multiple microservices will listen to the companies topics and forward every standardized workflow event to the inner platform topic. There, other microservices will inspect those events and forward them to the topic of every stakeholder, which is allowed to receive the information it contains. In order to detect wrong forwarding rules and allow a stakeholder to follow up the distribution of information, that he shares, every time an event is mediated between topics, a notification event is sent back to the topic of the original event producer. This allows every stakeholder to monitor their contribution to the platform from their own perspective. Monitoring dashboards can be setup to visualize this information flow. The flow of one event through the topic structure of the Logistics Platform is depicted in Figure 6.4 on the facing page. The specification for the event, which contains the delivery report of an event to another topic, is described in Listing 6.2. It most notably contains the mandatory fields *topicName* and *companyName* in the data section of the event, allowing to reconstruct the flow of events to certain topics.

```
{
```

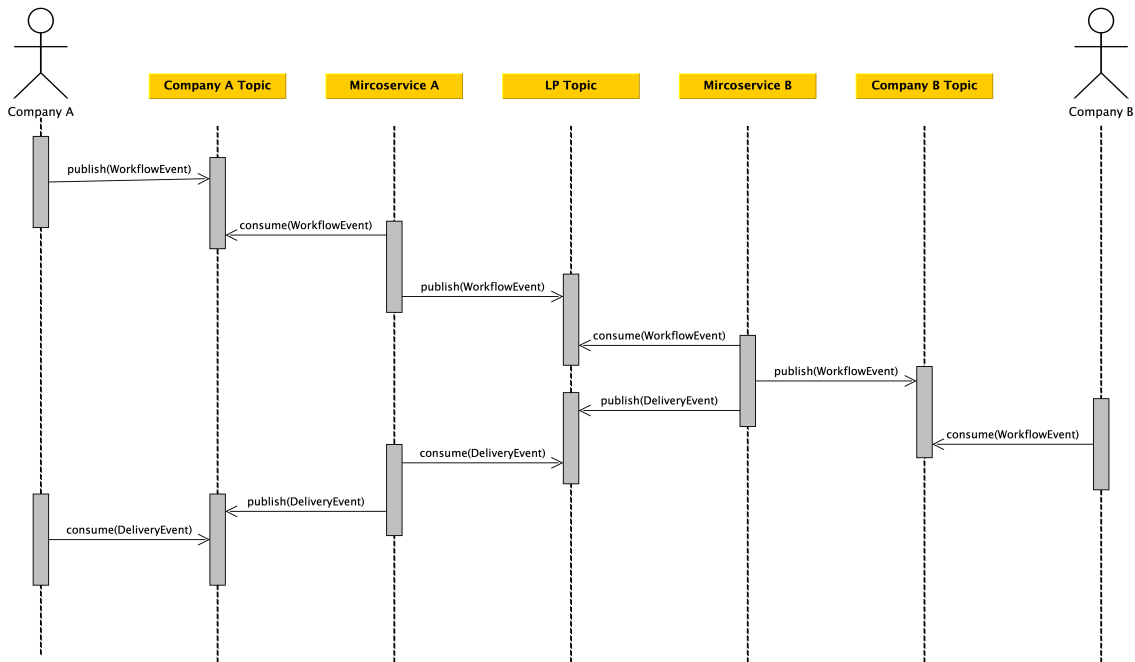


Figure 6.4.: The flow of an exemplary event through the Logistics Platform. Every company has a dedicated Kafka topic, which they can publish and subscribe to. Certain standardized events related to workflows will be forwarded to an internal platform topic by microservices provided by the platform operator. Other microservices will redirect events to the topics of concerned stakeholders and produce a delivery report event, which is send to the topic of the company, that originally produced the event.

```

...
"type" : "platform.reporting.event.delivered",
...
"data" : {
  "topicName" : "<The name of the topic where the event
    was published>",
  "companyName" : "<The name of the company owning the
    topic>",
}
}

```

Listings 6.2: An extension to the general event specification for the Logistics Platform, that reports the delivery of an event to another companies Kafka topic back to the original event publisher.

6.3.4. Choosing a Monitoring System

One important aspect of the Logistics Platform concerns monitoring. Chapter 5 on page 23 describes that monitoring a distributed software system like the Logistics Platform concerns performance metrics, which need to be collected on multiple layers:

- The business layer
- The application layer
- The infrastructure layer
- The client software layer
- The deployment pipeline layer

Monitoring the first four of those layers is addressed in the following sub-sections, as they all work a bit differently. The deployment layer will not be further examined due to the limited time available. As the whole aspect of CI/CD is only outlined briefly in Sub-section 6.3.5 on page 66, monitoring related to this aspect was not completed in time.

Monitoring the Infrastructure Layer

The infrastructure used to operate the Logistics Platform includes K8s and the nodes used to operate it. In K8s a service interface can be enabled, which grants access to performance metrics of nodes and pods¹⁷ from within the K8s cluster. The open source monitoring tool Prometheus¹⁸ can be deployed in the cluster to consume this service interface and collect metrics. As Prometheus has only limited capabilities concerning the creation of dashboards and alerting rules, other monitoring tools are better suited to match the monitoring requirements defined for the Logistics Platform. Fortunately, Prometheus provides interfaces, which can be consumed by tools like Grafana¹⁹ or Graphite²⁰. When combined, Prometheus and Grafana can fulfill all monitoring requirements for the infrastructure layer listed in Table 5.8 on page 30, which will provide powerful monitoring dashboards to the platform operator. Through custom alerting rules, the platform operator can be notified automatically to quickly respond to all problems related to insufficient or damaged infrastructure.

¹⁷A pod is a group of containers, which can share certain resources like storage. Every container must be connected to a pod.

¹⁸<https://prometheus.io/>

¹⁹<https://grafana.com/>

²⁰<https://graphiteapp.org>

Monitoring the Business Layer, Application and Client Level

Collecting metrics on the business layer is connected to a set of requirements, which are listed in Table 5.6 on page 28. Such monitoring capabilities are necessary to track the progress of workflows and to detect abnormalities in workflow instances, which could have a negative impact on the business goals of a company connected to the platform. Those metrics are highly dependent on the use case of the workflow and the components working on it, so every company might define these metrics on their own. For standardized workflows a set of mandatory monitoring events for the business layer might be defined at a later point in order to consolidate monitoring for the platform operator. As all components communicate on the platform through events, monitoring metrics should also be published as events to the stakeholders topic. This has to huge advantage, that all workflow related events as well as all monitoring events are recorded in order inside the topics. A monitoring tool like Graphite²¹ is capable of consuming Kafka events to create custom monitoring dashboards.

The setup involving Graphite and special monitoring events can also be used to collect metrics on the application and client software layer, which are also custom for every component within the platform. This will address the requirements described in Table 5.7 on page 29 and Table 5.9 on page 30. Every company could be provided their own Graphite instance which can only consume their assigned topic. Graphite can also be connected to Grafana, which allows the platform operator to integrate the metrics collected at all four layers within one monitoring tool and even in one single dashboard.

Using the same mechanism, which is used to mediate workflow related events between topics, the standardized monitoring events can also be moved to the platform internal topic. This would result in a single topic containing the whole stream of event on the platform, which could be consumed by the Graphite instance assigned to the platform operator, which could be connected to the Grafana instance monitoring the infrastructure layer.

The event schema introduced in Sub-section 6.3.3 on page 60 and demonstrated in Listing 6.1 on page 61 can be used as the base specification for the monitoring events as well. Listing 6.3 on the following page contains an event schema based on this specification, which can be used for all monitoring events on the business, application and client software layer:

- The event type names the layer on which a metric was collected

²¹<https://graphiteapp.org/>

6. Designing the Logistics Platform

- The body contains fields specifying where the metric was collected
- Every company can use own metrics according to their usecase. In order to monitor to create a view showing the course of a metric over the time, all connected values should be assigned in events using the same metric name.

```
{
  ...
  "type" : "platform.reporting.level.business",
  // "type" : "platform.reporting.level.application",
  // "type" : "platform.reporting.level.client-software",
  ...
  "data" : {
    "topicName" : "<Name of the Kafka topic where the
      event was published>",
    "companyName" : "<Name of the company owning the
      Kafka topic>",
    "metric" : "<The name of the metric>",
    "value" : "<The value of the metric>"
  }
}
```

Listings 6.3: An extension to the general event specification for the Logistics Platform contained in Listing 6.1 on page 61. This event schema can be utilized to report metrics on the business, application and client software layer.

6.3.5. Choosing a Continuous Integration System

Continuous Integration (CI) and Continuous Delivery (CD) are both terms that are strongly associated with DevOps, which is a modern approach on managing software projects, that covers a variety of different development and operational aspects. (Kim et al., 2016) gives an extensive explanation on the history and contents of the term DevOps from the perspective of industry leaders in the IT sector. CI summarizes tools that allow teams to work autonomously on a shared software project, where the work of all individuals and teams is frequently merged together. This shall prevent teams from working on different version control branches for longer time periods to prevent merge conflicts and the introduction of new bugs due to infrequent integration.

CD goes even one step further and allows teams to deploy software to production autonomously. This can be achieved through the setup and enforcement of deployment pipelines, that are connected to the source repository. Every time a team pushes changes to the release branch of the repository, the deployment pipeline will

automatically integrate and compile the software and execute a chain of automated tests before eventually deploying the software to production servers.

In practice, CI/CD can be implemented through many different open source technologies, the most popular being Jenkins X²² and GitLab CI²³. Functionality-wise both tools would be fit to use for the Logistics Platform. As all source code of the components developed for the Logistics Platform is currently living within the GitLab of the ISP, it might be easier to initially setup CI/CD using the GitLab CI. Both tools can be connected to Grafana in order to integrate performance metrics of the deployment pipelines into the central monitoring dashboard of the platform provider and are capable to deploy software directly to a K8s cluster. This would meet the requirements defined in Table 5.10 on page 31.

A deployment pipeline for microservices operated on the Logistics Platform should contain the following steps:

- Running unit tests against the new version of the application.
- Building a Docker container from the new version of the application.
- Running automated test suites against the new Docker container.
- Deploying the new version of the container to K8s.

As mentioned before, the aspect of CI/CD could not be implemented in time in Chapter 7 on page 69, which will be further addressed in Section 9.2 on page 86.

6.4. Summary

This chapter followed the complete design process of the Logistics Platform. Section 6.1 on page 33 introduced different architecture patterns with a focus on distributed software systems. All of the architecture patterns were examined and evaluated using the SAAM methodology and the scenarios described in Chapter 4 on page 15. By the end of this section the base architecture for the Logistics Platform was selected as an event-driven microservice approach, which allows for a loosely coupled and scalable software system.

²²<https://jenkins-x.io/>

²³<https://docs.gitlab.com/ee/ci/>

6. Designing the Logistics Platform

Building upon this decision, Section 6.2 on page 53 gave a holistic view on the Logistics Platform architecture naming and briefly outlining the purpose of the different components included. All of the components fulfilling different tasks within the platform were explained in more detail in Section 6.3 on page 54. Remaining design decisions were made in this section, in order to create an architecture design, which fulfills all requirements defined in Chapter 5 on page 23, when the platform is implemented.

Chapter 7 on the next page now follows the implementation process of the Logistics Platform in an incremental process, where all stages are based on the structure of sub-sections in Section 6.3 on page 54. The final implemented prototype will be evaluated in Chapter 8 on page 77 by demonstrating, how the exemplary logistical workflow described in Section 1.1 on page 2 can be deployed and operated on the platform.

7. Implementation of the Platform Design

Based on the architecture design completed in Chapter 6 on page 33 this chapter follows the implementation of the Logistics Platform. As explained in Chapter 3 on page 11 this thesis will follow the incremental software development process, meaning that after every development stage a functional prototype is available, which fulfills a growing number of requirements for the overall system. Due to time restrictions not all aspects of the design will be implemented, which is the trade off for enabling the evaluation of the final prototype, using the exemplary logistical workflow in Chapter 8 on page 77.

The different stages of the incremental process match the sections of this chapter, which in return correspond with the order of sub-sections in Section 6.3 on page 54. After every stage a prototype is available, that includes a growing number of aspects of the Logistics Platform.

- In Section 7.1 the K8s cluster, which will host all other components, will be setup
- In Section 7.2 on page 72 Kafka will be deployed to to the cluster to enable communication between components
- In Section 7.3 on page 73 monitoring on the infrastructure layer and parts of the application layer will be enabled

7.1. Setting Up Kubernetes

The base of the platform is K8s, which will be used to host and operate all other components of the platform. Those components include all the microservices used to operate workflows on the platform. Initially a private K8s cluster was setup on IT infrastructure of the ISP. Unfortunately this could not be completed, because a number of problems occurred along the way. This was mostly due to features, which seem to be part of the K8s core, but in reality rely on third-party software or need custom implementation. First of all, in order to ensure a high level of availability, it was necessary to create a K8s cluster consisting of multiple nodes to compensate

7. Implementation of the Platform Design

eventual outage of one or two nodes. This was achieved by following the official documentation of K8s using the *kubeadm*-tool shipped with K8s (Kubernetes.io, 2020b). As microservices within the cluster need to communicate independent to the node they are living on, K8s makes use of a virtual network, which relies on third-party technology. This was one of the more easier problems to solve, as there are numerous open source technologies available, that can be used with K8s (Kubernetes.io, 2020a). This at least allowed microservices to communicate using cluster internal host names and IPv4 addresses. Exposing services to external IP addresses and load balancing for services is not possible using just the basic open source networking technologies available.

Another problem concerns storage for pods. Many of the components for the Logistics Platform, most notably Kafka, rely on persistent storage in order to function as intended. Even though K8s allows to request storage for pods in their configuration files, those *PersistentVolumeClaims* still need to be resolved by other components within the cluster. So called *PersistentVolumes* classes are not contained within the K8s core and need to be implemented, in order to provide persistent storage to pods. Currently the only workaround to avoid implementing persistent volumes from scratch is to use of NFS volumes, that are operated in the same sub-network the K8s nodes are connected to. Those NFS volumes can be used to provide persistent storage to pods in a strongly limited manner.

These problems were also confirmed in personal conversations with industry experts working for large companies, that tried to setup and operate their own K8s clusters and later switched to managed environments of cloud providers like Amazon or Google (Dang, 2019; Schröder, 2020). As this thesis only uses K8s as a tool to demonstrate the future use of the platform, completing the setup of a private K8s cluster was not advisable, due to more important aspects of this thesis. Therefore the platform was setup in a managed K8s cluster within the Google Cloud Platform, which offers the full set of features for K8s like networking, load balancing and persistent storage. All K8s configuration objects developed in this chapter or Chapter 8 on page 77 could also be used in a cluster within another public cloud like the Amazon Web Services (AWS). The K8s cluster used for the prototype consists of three nodes each assigned one virtual CPU core and 1.21 GB of RAM as depicted in Figure 7.1 on the next page.

Google provides a Software Development Kit (SDK) which allows to connect a K8s cluster within the Google Cloud Platform (GCP) to a remote computers in order to issue commands locally, that will be mirrored on one of the nodes hosting the K8s cluster. *kubectl*-commands to schedule containers can therefore be issued without

Name	Status	Angeforderte CPU	Zuweisbare CPU	Angeforderter Arbeitsspeicher	Zuweisbarer Arbeitsspeicher	Angeforderter Speicher	Zuweisbarer Speicher
gke-your-first-cluster-1-pool-1-b9d57800-6vmf	Ready	418 mCPU	940 mCPU	246.42 MB	1.21 GB	0 B	0 B
gke-your-first-cluster-1-pool-1-b9d57800-nzqx	Ready	380 mCPU	940 mCPU	125.83 MB	1.21 GB	0 B	0 B
gke-your-first-cluster-1-pool-1-b9d57800-rch5	Ready	100 mCPU	940 mCPU	0 B	1.21 GB	0 B	0 B

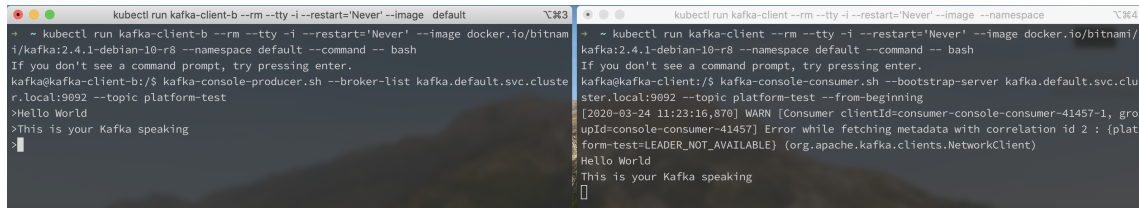
Figure 7.1.: The managed Kubernetes environment for the Logistics Platform which is hosted on the Google Cloud Platform. It consists of three servers each assigned one vCPU core and 1.21 GB RAM. Storage and external IP addresses can be acquired at additional costs.

logging in to the nodes via SSH. In order to deploy applications to the cluster, the containerized software must be hosted in a Docker image registry, which is accessible by the nodes of the K8s cluster. This does not exclude privately image registries like the one contained in the ISP GitLab server, where the source code of all components, which will be developed in the scope of this thesis, is hosted. In order to access these Docker images, the K8s cluster needs authentication credentials for this GitLab server, which grant read access to the repository connected to a specific Docker image. Listing 7.1 shows how the credentials can be saved within the cluster to allow the deployment of all microservices developed in this thesis.

```
kubectl create secret docker-registry regcred \
  --docker-server=<registry> \
  --docker-username=<username> \
  --docker-password=<password> \
  --docker-email=<email>
```

Listings 7.1: Authenticating Kubernetes to a private Docker image registry.

7. Implementation of the Platform Design



```
kubectll run kafka-client-b --rm --tty -i --restart='Never' --image docker.io/bitnam
i/kafka:2.4.1-debian-10-r8 --namespace default --command -- bash
If you don't see a command prompt, try pressing enter.
kafka@kafka-client-b:/$ kafka-console-producer.sh --broker-list kafka.default.svc.cluste
r.local:9092 --topic platform-test
>Hello World
>This is your Kafka speaking
```

```
kubectll run kafka-client --rm --tty -i --restart='Never' --image docker.io/bitnam
i/kafka:2.4.1-debian-10-r8 --namespace default --command -- bash
If you don't see a command prompt, try pressing enter.
kafka@kafka-client:/$ kafka-console-consumer.sh --bootstrap-server kafka.default.svc.clu
ster.local:9092 --topic platform-test --from-beginning
[2020-03-24 11:23:16,870] WARN [Consumer clientId=consumer-console-consumer-41457-1, gro
upId=console-consumer-41457] Error while fetching metadata with correlation id 2 : {plat
form-test=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)
Hello World
>This is your Kafka speaking
```

Figure 7.2.: The Kafka setup was tested using two containerized applications, which provide a basic CLI for a Kafka producer (on the left side) and consumer (on the right side).

7.2. Setting Up Kafka

As the event mediator of choice, Kafka will be setup to operate within the K8s cluster. One challenge of such setup is that Kafka relies on Apache Zookeeper. The combination of these tools implements its own scaling mechanisms, which can not be linked to the scaling capabilities provided by K8s. As one instance of the Kafka broker can handle up to 1000 events per second (Dobbelaere & Esmaili, 2017), this will not become a huge problem very fast, but might result in less efficient use of available infrastructure. In this thesis, three instances of the Kafka broker will be deployed, without the automatic scaling mechanisms of K8s enabled. This allows Kafka and Zookeeper to use their own means of scaling, so some brokers might run idle most of the time. While the resource overhead is not a huge problem in this kind of setup, a Logistics Platform handling millions of events per second might require Kafka to be operated outside of K8s. Bitnami provides K8s objects¹, that were adapted to setup Kafka for the Logistics Platform containing three Kafka brokers and one instance of Zookeeper. All K8s objects used in this thesis can be found on the DVD which is enclosed to this thesis. The Kafka setup was initially tested by setting up a generic publisher and subscriber applications as shown in Figure 7.2. Kafka can be configured to limit the access to topics by using authentication mechanisms like Kerberos (Adams, 2011) and authorization rules defined in Access Control Lists (ACL) (Confluent Inc., 2020). This will not be performed during the scope of this thesis, but is required when the Logistics Platform will host the logistical workflows of real companies. All communication between a publisher or subscriber with Kafka should be end-to-end encrypted using SSL to prevent man-in-the-middle attacks.

¹<https://github.com/bitnami/charts/tree/master/bitnami/kafka>

7.3. Setting Up the Monitoring System

As described in Chapter 5 on page 23, monitoring the Logistics Platform is necessary on different layers. The base for the platform is the infrastructure on which all parts of the software system are hosted. In case of the Logistics Platform K8s is used to operate all other components that are part of the system. The infrastructure layer therefore contains metrics collected at node level as well as basic performance metrics of K8s. Sub-section 7.3.1 will explain, how the underlying infrastructure of the Logistics Platform is monitored using Prometheus and Grafana. Monitoring the application layer will, in the scope of this thesis, only be implemented for Kafka in Sub-section 7.3.2 on the following page. Unfortunately monitoring the application, business and client-software layer were not completed in time. This means, that for all microservices used to simulate the workflow in Chapter 8 on page 77 only performance metrics on the infrastructure layer will be collected.

7.3.1. Monitoring the Infrastructure Layer

K8s is operated on multiple nodes in order to split the load produced by the platform to multiple machines and provide coping mechanisms in case a node is unavailable. In K8s a service API can be enabled, which makes performance metrics collected on the node level available to certain components within the K8s cluster. To use this API, *Tiller*² needs to be deployed to the cluster. Tiller is the server component of the *Helm*³ package manager, which can be used to deploy applications to a K8s cluster without having to manually provide the required configuration files. Tiller also provides an API containing performance metrics of the K8s cluster, that it is deployed on. This allows a monitoring tool like Prometheus to consume the *Tiller*-API and to further process this information. As described in Sub-section 6.3.4 on page 64, Grafana will be connected to Prometheus as it provides more powerful features for creating dashboards and alert rules. Grafana can also be configured to consume additional data sources like potentially Graphite to include monitoring Kafka events as suggested in Sub-section 6.3.4 on page 65. Overall this makes Grafana more flexible to meet current and future monitoring requirements.

(Vermeulen, 2019) gives a detailed explanation on how Tiller, Prometheus and Grafana can be used to monitor a K8s cluster and instructs how those components can be deployed and configured to create a robust monitoring setup. In order to keep the K8s cluster structured and later restrict access to certain parts of the

²<https://v2.helm.sh/docs/glossary/#tiller>

³<https://v2.helm.sh/>

7. Implementation of the Platform Design

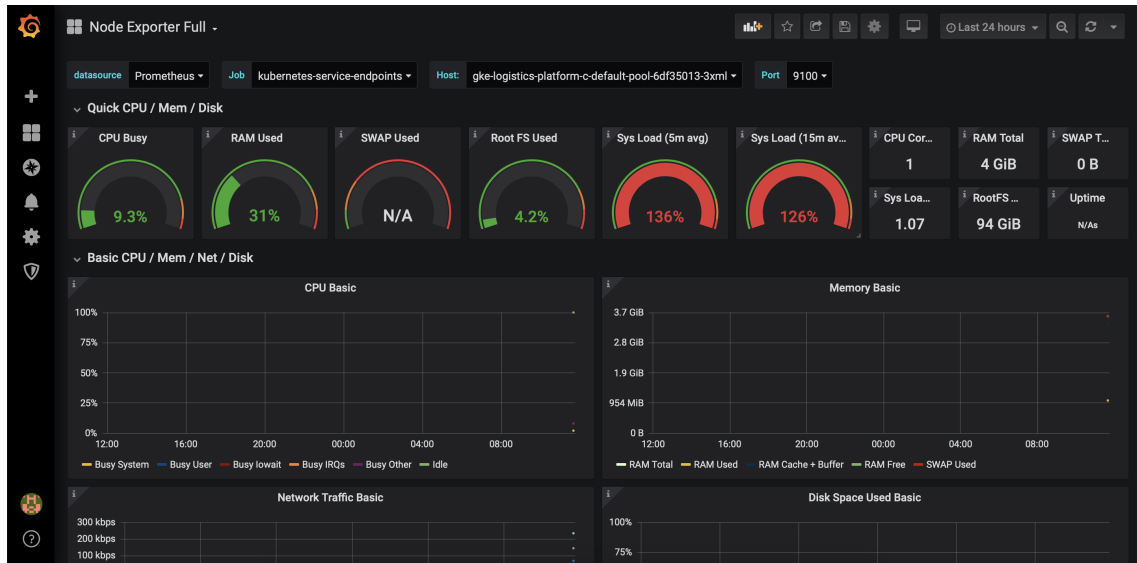


Figure 7.3.: The Grafana dashboard containing metrics collected on the infrastructure layer. This includes CPU load, RAM usage, disk I/O and network performance of the hosts forming the K8s cluster.

platform, all monitoring related components are deployed in the namespace *monitoring*. Figure 7.3 depicts the Grafana monitoring dashboard, which contains all performance metrics of the Logistics Platform collected on the infrastructure layer. This includes information about the available CPU, RAM, disk I/O and network capacities of the nodes as well as the current system load. Those metrics are also broken down to showcase the used resource of individual deployments.

This monitoring setup can be recreated by using the configuration files and commands listed on the enclosed DVD.

7.3.2. Monitoring Kafka

As mentioned before, monitoring on the application layer as well as the business and client-software layer was not completed in time in this thesis in order to perform the evaluation of the prototype in Chapter 8 on page 77. Kafka is the only exception to this, as it is the only component operated on the platform, which can obviously not be monitored using Kafka events. The configuration objects for deploying Kafka in K8s allow to enable the use of a *ServiceMonitor*, which collects performance metrics, that can be scraped with the Prometheus Operator⁴. This way, the same

⁴<https://github.com/coreos/prometheus-operator>

Prometheus instance used to monitor the infrastructure layer of the platform can be used to collect the performance metrics of Kafka, which are then passed on to Grafana. The monitor for Kafka created for the Logistics Platform is depicted in Figure 7.4 on the following page. It currently only shows two metrics to provide a simple proof of concept, which was tested using the abstract Kafka publisher and subscriber as mentioned in Section 7.2 on page 72.

7.3.3. Monitoring Other Layers

Due to the limited time, it was not possible to setup and test monitoring on all layers of the Logistics Platform. As suggested in Sub-section 6.3.4 on page 65 Kafka events could be used in order to collect metrics at the application, business and client-software layer as this would result in very little effort for the stakeholders developing components for the system, because they also use Kafka events to communicate workflow-related information. Another possibility for monitoring metrics on these layers is to develop *ServiceMonitors*, which could be consumed by the Prometheus Operator. There are client libraries available⁵ for many different programming languages, which can setup an API endpoint which can be consumed by the Prometheus Operator. As this is more complex to setup, monitoring through Kafka events should be preferred, but the other option remains in case certain components shall not be monitored through Kafka events.

In order to capture the monitoring events in Kafka, it is necessary to deploy and configure Graphite. This tool could be deployed for every stakeholder to listen for monitoring events in their respective topic. Following that approach, every stakeholder receives their own powerful monitoring dashboards, which could be setup automatically when a new company joins the platform. It is also possible to connect Graphite to Grafana for the platform operator, in order to receive a monitoring dashboard which contains monitoring metrics collected at all levels for all the components at once.

7.4. Summary

The prototype of the Logistics Platform was implemented using incremental steps. First the underlying software platform was setup with K8s in order to host all other components of the platform. Then, Kafka was configured to operate within the K8s cluster to allow event-based publish-subscribe communication between components.

⁵<https://prometheus.io/docs/instrumenting/clientlibs/>

7. Implementation of the Platform Design

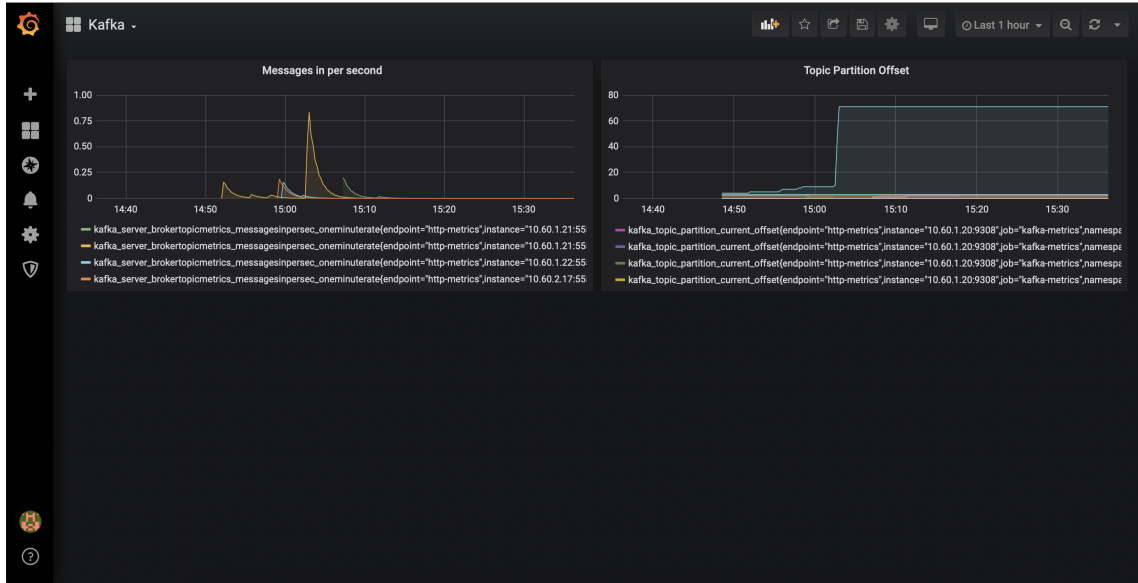


Figure 7.4.: A basic Grafana monitoring dashboard containing performance metrics of Kafka, which were collected by the Prometheus Operator.

At last, monitoring capabilities for the infrastructure layer and parts of the application layer were introduced to showcase how a central monitoring dashboard can later be used by the platform operator.

Unfortunately it was not possible to further implement the Logistics Platform, meaning that there are still a number of open requirements left in Chapter 5 on page 23, that need to be implemented in the future. As the incremental build model was selected for this thesis, it is still possible to evaluate the system design and prototypic implementation in Chapter 8 on the facing page.

8. Evaluation

Evaluation of the platform prototype will be performed by deploying components to the Logistics Platform, which simulate the exemplary workflow as described in Section 4.1 on page 16. This will demonstrate that it is indeed possible to operate interorganizational workflows on a shared software platform.

8.1. Evaluation Goals

In Chapter 4 on page 15 one use case for the Logistics Platform was described in detail: A car manufacturer in Estonia ships car components through inland waterways and needs to know the whereabouts of his shipments, in order to create an efficient assembly schedule. The event-driven approach for the platform suggests that every fulfilled step within the logistical workflow of shipping those components between the two factories should produce an event, which is forwarded to every company it concerns to further drive the workflow or otherwise act as a notification. This would enable the Estonian car manufacturer to track his shipment throughout the workflow and allow him to prepare more efficient assembly schedules.

This chapter will showcase, how a logistical workflow like this can be operated on the Logistics Platform. Therefore, it is first necessary to find the answers to three fundamental questions:

- Which workflow steps should produce events?
- Which event is associated with which workflow step?
- Which stakeholder is allowed to receive which events?

After those questions have been answered, a number of microservices have to be implemented in order to simulate the logistical workflow. After this has been completed, the components can be deployed to the platform prototype in order to simulate the workflow.

8.2. Setting Up the Evaluation

First of all, the different steps contained in the workflow are identified and corresponding events describing the fulfillment of those workflow steps are specified. Also, every event will be assigned a producer and a list of subscribers allowed to receive the event. A generic microservice will be developed, which can parse this information in order to later simulate the workflow. According to the topic structure defined in Sub-section 6.3.3 on page 62 every company is only allowed to publish and subscribe to their own topic, while additional microservices provided by the platform operator will forward events between a stakeholders topic and one internal platform topic. From there, other microservices will forward the events to the topics of those stakeholders, which are allowed to receive the information contained. Forwarding of events between topics will automatically cause a notification event, informing the original event producer that the event was forwarded to a specific topic. This process was already described in detail in Sub-section 6.3.4 on page 65. A list of all the events used for the exemplary workflow in the context of this evaluation can be found in Table 8.1 on page 80. The delivery reporting event is not included here, as it was already specified in Listing 6.3 on page 66 and will be produced multiple times throughout this workflow.

All of the microservices necessary to simulate this workflow can be implemented as one generic application, which will receive its tasks within the workflow using a number of configuration variables. This application will be developed in JavaScript and executed using the Node.js¹<https://nodejs.org/en/> runtime within a Docker container operated on the platform. In order to fulfill all of those tasks, the application must be able to act in three different ways, according to the role that is configured to play and therefore its configuration:

1. Initiating the workflow by producing the event *shipping.contract.initiated*
2. Mediating a specific event between two topics and producing the delivery report event to the topic used by the original publisher.
3. Consuming an event and producing the follow-up event to continue the workflow.

Every microservice is configured to take on the role of one single company involved in the workflow including the platform operator. This configuration determines, which topics the microservice will access and what it will do with the events of a certain type, which are published in a certain topic. When the microservice simulates the behavior of a workflow stakeholder, it is responsible for listening for a

¹`\unskip\penalty\@M\vrulewidth\z@height\z@depth\dp,`

specific event within the workflow to publish the follow up event. The only exception to this is one microservice, which produces the event initiating a new instance of the workflow. This will be done every few seconds to continuously simulate the workflow. All configuration options necessary to instantiate the microservices are set via environmental variables within the K8s configuration objects. This list of variables includes:

- `KAFKA_CLIENT_ID`: A unique name for the Kafka client.
- `KAFKA_GROUP_ID`: An assigned group for the Kafka client. This could be later used, when multiple instances of one microservice share the load of one task.
- `KAFKA_BROKER`: The address of the Kafka broker within the cluster, e.g. *kafka:9092*
- `ROLE`: The String *PLATFORM*, *FINISH_PORT*, *ESTONIAN_PORT*, *FINISH_FACTORY*, *ESTONIAN_FACTORY* or *SHIPPING_COMPANY*
- `EVENT_TYPE`: The type of event to handle
- `SUBSCRIBE_TOPIC`: The name of the topic to subscribe to (only mandatory for platform microservices)
- `SUBSCRIBE_TOPIC`: The name of the topic where an event shall be published (only mandatory for platform microservices)

The application contains different classes, which allow a stakeholders microservice to identify the current step within a workflow based on an event to produce the event connected to the next workflow step when it is configured to do so. All internal processes within a company are completely ignored in this workflow simulation, so all events will be produced immediately to drive the workflow.

The platform microservices in return are configured to listen within a specific topic for a certain event type to forward it to another predetermined topic, while also producing the delivery report event to the topic, where the event was initially published.

8.3. Simulating the Logistical Workflow

As described in the previous section, multiple microservices will simulate the exemplary workflow by publishing events describing the fulfillment of workflow steps, which will then be forwarded to the topics of other stakeholders involved. Every event described in Table 8.1 on the preceding page will be produced by one

8. Evaluation

Event Type	Producer	Subscribers
shipping.contract.initiated	Finnish Company	Shipping Company
shipping.contract.confirmed	Shipping Company	Finnish Company Estonian Company
voyage.initiated	Shipping Company	Finnish Company Estonian Company Finnish Port Estonian Port
voyage.docking.schedule.created	Finnish Port	Shipping Company
voyage.docking.schedule.created	Estonian Port	Shipping Company
shipping.payload.fetched	Shipping Company	Finnish Company Estonian Company
vessel.docked	Shipping Company	Finnish Company Estonian Company Finnish Port
vessel.loaded	Finnish Port	Finnish Company Estonian Company Shipping Company
vessel.undocked	Shipping Company	Finnish Company Estonian Company Finnish Port
vessel.docked	Shipping Company	Finnish Company Estonian Company Estonian Port
vessel.unloaded	Estonian Port	Finnish Company Estonian Company Shipping Company
shipping.payload.delivered	Shipping Company	Finnish Company Estonian Company

Table 8.1.: Events related to the exemplary workflow as implemented for the evaluation. It contains information on who will produce the event and which stakeholder are allowed to receive it.

microservice acting the role of a stakeholder. The microservice producing the *shipping.contract.initiated*-event in the topic of the Finish port authorities contains a mechanism resulting in the event being published every seven seconds. All other events are produced automatically, because mediators and microservices acting the role of the platform operator and other companies are setup to process the whole workflow. In this setup every microservice fulfills one single role, so it is only concerned of the events of one type within one topic to which they will produce only one predetermined event. The microservices simulating the behavior of the mediator components underline the decentralized governance approach, as they are only responsible for atomic parts within the information flow.

As it is rather difficult to setup the prototype including a full-grown K8s cluster to reproduce this evaluation, the enclosed DVD contains a Docker Compose file, which can be used on every computer, which has a recent version of Docker installed. This container composition file can be used to automatically setup and execute a smaller version of this evaluation, which is missing the monitoring capabilities achieved through Prometheus and Grafana. Still, this can be used to validate, that the microservices are able to operate the workflow by communicating via Kafka events following the Logistics Platform design. The additional tool Kafka WebView² is also deployed in this composition, which allows to browse through the Kafka topics within the evaluation setup to comprehend the information flow. After the composition was started using *docker-compose up* and a few minutes have passed, the tool can be accessed through a web browser on port 8080 of the host machine. The default login credentials for Kafka WebView are *admin@example.com:admin*. Figure 8.1 on the following page shows a screenshot of the Kafka WebView containing a live stream of all events within the Estonian factories topic.

8.4. Monitoring the Logistical Workflow

Monitoring the Logistics Platform was setup on the infrastructure and application layer. The performance metrics of every deployment is automatically collected by Prometheus and displayed on the Grafana dashboard. This includes information on the system resources used by the different deployments within the K8s cluster. Monitoring on the business and application layer requires additional metrics defined by the developers of the microservices operated on the platform. As mentioned in Sub-section 6.3.4 on page 65 this can be achieved through additional events in Kafka, which a company can utilize to describe performance on these abstract layers. Unfortunately, due to time constraints, it was not possible to develop a monitoring

²<https://github.com/SourceLabOrg/kafka-webview>

8. Evaluation



Figure 8.1.: The Kafka WebView streaming the events inside the Estonian factories topic. The highlighted Strings show that some events are related to the same workflow instance.

dashboard which collects and depicts those performance metrics contained within Kafka events.

8.5. Altering a Logistical Workflow

Altering the logistical workflow is also very simple. For example it is possible to forward certain events to additional actors by deploying additional mediator components. As those components are generic, they can be configured and deployed in seconds to adapt to the altered workflow. The companies, whose events are now published to an additional stakeholder receive automatic notifications through the mediator components. As a mediator component will only forward one specific event type one-way between two topics, every company involved in the platform can easily add new components operating in their companies topic, without affecting the flow of other events on the platform. When a new event shall be forwarded to other companies, the stakeholder can contact the platform operator, who will then deploy a new mediator service for this event type. This service is subscribed to the companies topic and moves the concerned events to the platform internal topic. Every company interested in those events can then also contact the platform operator to receive a mediator service, which forwards events to their topic in agreement with the original event producer. When access to events shall be revoked, only one mediator component needs to be removed from the platforms K8s cluster, which automatically

stops information flow at this point.

8.6. Summary

In this chapter the platform prototype developed in Chapter 7 on page 69 was evaluated by developing and deploying components to the K8s cluster, which simulate the exemplary workflow described in Section 4.1 on page 16. All of the microservices performing the workflow are the same Node.js application, which can be configured and deployed to fulfill various roles within the workflow, by including a number of environment variables through the K8s deployment objects. As the implementation of the platform design could not be completed to all extents in the scope of this thesis, further implementation is necessary, before all aspects might be evaluated through future work. This includes monitoring capabilities on the application, business and client software layer as well as aspects related to CI/CD.

9. Concluding Remarks

This thesis proved that a Logistical Platform can be used to perform shared workflows of companies working in the domain of transportation through inland-waterways. Therefore, a concept for such platform was designed, implemented and evaluated.

First, the current state within the logistics domain was researched and summarized in Chapter 1 on page 1. Chapter 2 on page 5 gave a more detailed perspective on the current state of this business area with a focus on interoperability aspects, highlighting the need for a shared software platform capable of operating interorganizational workflows between companies. In order to create said platform, a number of methodologies of Software Engineering were introduced and combined in Chapter 3 on page 11. Following those methodologies, scenarios for the Logistics Platform were elaborating in Chapter 4 on page 15 including a real business case and other perspectives on a potential platform used to operate shared logistical workflows. Those scenarios were translated into technical requirements in Chapter 5 on page 23 manifesting their explicit and implicit contents. The architecture for the Logistics Platform was developed in Chapter 6 on page 33 examining the suitability of different architecture patterns, before creating the detailed design for all the components required for the platform. Chapter 7 on page 69 followed the incremental software development process, which was used to implement the Logistics Platform using a prioritized list of the different aspects. The resulting prototype was evaluated in Chapter 8 on page 77 by developing, deploying and operating microservices on the platform, which simulate the logistical workflow described in Section 4.1 on page 16.

9.1. Limitations

As this thesis is based on one specific business case, it remains in question, if the concept for the Logistics Platform is relevant to other companies as well. The business case provided by (Leucker, 2019) was therefore considered from a more abstract position whenever possible in order to focus on the interorganizational interplay, that could be transferred to other business cases as well. Furthermore, the need for such platform was examined in Chapter 2 on page 5 by analyzing the current

9. Concluding Remarks

state of the business area, showing a high demand for interoperability. Still it is necessary to evaluate the platform design and implementation by simulating other workflows common in transportation through inland-waterways. Contemporaneous to the work on this thesis, (Queßeleit, 2020) examined interorganizational workflows operated at the Lübeck harbor, which could be simulated on the platform prototype by adapting the generic microservice developed in Chapter 8 on page 77.

Due to the limited time, it was not possible to implement all parts of the platform design. Some important aspects like CI/CD are completely missing in the prototype. While monitoring was enabled for some parts of the platform, metrics provided through Kafka events can only be included to the dashboards after additional software is introduced to the K8s cluster. are configured and deployed to the platform. Those parts of the architecture design could therefore not be evaluated by the end of this thesis.

9.2. Future Work

This thesis lay the base for a Logistics Platform that might be used to operate workflows in transportation through inland-waterways spanned across multiple companies. As the lack of interoperability is considered one of the major problems for the logistics domain today, the concepts elaborated in this thesis could contribute to future developments in this business area. Additional work is required to complete the implementation of a prototype, which could be used to demonstrate the impact of a neutral platform for workflows to real stakeholders within the domain.

Eventually funding is required to seriously push the concept of a Logistics Platform further, as multiple companies have to be convinced to join such platform and allowing it to host and operate real logistical workflows. This would require a team of developers and operators to complete the platform implementation and making it production ready. Another important aspect of interoperability concerns a specification for events, which standardize certain workflows and workflow steps within the industry to enable companies to exchange information in a way that can be interpreted correctly by every stakeholder.

A. Appendix

A.1. Functional Requirements for the Logistics Platform

Req. No	Description
R001	The platform <i>must</i> provide mechanisms allowing information exchange between companies participating in shared workflows.
R002	The platform <i>must</i> standardize workflow related information exchange in order to enable interoperability between companies fulfilling similar tasks.
R003	The platform <i>must</i> provide mechanisms that allow a company to limit access to information it provides to the platform.
R004	The platform <i>should</i> ensure that information does not breach the limitation rules defined by a company sharing information.
R005	The platform <i>must</i> log all access to information shared by a company in order to detect breaches and initiate necessary counter measures.
R006	The platform <i>must</i> encrypt all information which is processed on the platform.
R007	The platform <i>should</i> provide configuration files to setup production like environments for teams developing software for the platform.
R008	The platform <i>should</i> contain a Git repository holding all configuration files necessary for creating a production like environment.
R009	The platform <i>should</i> only allow deploying software through the standardized deployment pipeline.
R010	The platform <i>should</i> run automated test suites against new version of software operated on the platform.
R011	The platform <i>should</i> include all those test suites into the deployment pipeline automatically rejecting deployments that fail mandatory tests.
R012	The platform <i>should</i> ensure that no software can be deployed that breaks the functionality of other components.
R013	The platform <i>should</i> allow frequent deployment of software components.

A. Appendix

Req. No	Description
R014	The platform <i>should</i> automatically integrate updates made by different developers and teams on a frequent basis.
R015	The platform <i>should</i> provide tools supporting CI.
R016	The platform <i>should</i> allow automated deployments of software which passes all automated tests inside the deployment pipeline.
R017	The platform <i>should</i> allow automated deployments of software when no interfaces were changed in a non backward-compatible way.
R018	The platform <i>should</i> automatically perform tests which evaluate if the interfaces of a software component were changed.
R019	A component responsible for fulfilling steps of a workflow <i>must</i> provide metrics allowing the platform operator to track the progress of all workflow instances.
R020	The component initiating a workflow <i>must</i> provide a unique identification code for the workflow instance.
R021	A component responsible for fulfilling steps in a workflow <i>must</i> include the unique workflow instance identification code in all communication related to a workflow instance.
R022	The platform operator <i>must</i> be provided with metrics allowing him to track all workflow instances operated on the platform.
R023	The platform <i>must</i> provide tools to the platform operator that visualize the information flow between components that run a shared workflow.
R024	The platform <i>should</i> provide tools that allow all actors to see who has access to the information they share on the platform.
R025	The platform <i>should</i> detect abnormalities inside individual workflow instances.
R026	The platform <i>should</i> automatically inform the platform operator about detected abnormalities.
R027	Every component on the Logistics Platform <i>must</i> write internal failures to a log that can be accessed by the platform provider.
R028	Every component on the Logistics Platform <i>should</i> provide additional metrics (e.g. transaction times) allowing the platform provider to monitor the performance of the application.
R029	The platform <i>should</i> provide a monitoring dashboard for the platform operator containing application layer metrics of all components on the platform.
R030	The platform <i>should</i> provided a monitoring dashboard for every company cooperating on the platform containing application layer metrics of all their components on the platform.

Req. No	Description
R031	The monitoring dashboard <i>should</i> enable filtering mechanisms to isolate certain metrics and / or components.
R032	The CPU load, RAM usage, I/O disk operations and network traffic of all servers used to operate the Logistics Platform <i>should</i> be monitored in order to setup efficient infrastructure.
R033	Metrics collected at infrastructure level <i>should</i> be accessible by the platform operator on a central monitoring dashboard.
R034	The monitoring dashboard <i>should</i> allow to setup alerts related to individual or a group of metrics.
R035	The platform <i>should</i> provide APIs allowing client software to feedback performance metrics.
R036	The platform provider <i>should</i> be able to access the performance metrics provided by all client software.
R037	The performance information of client software <i>should</i> be accessible by the company who owns the software.
R038	The monitoring dashboard <i>should</i> visualize performance metrics provided by client software.
R039	The monitoring dashboard <i>should</i> provide functionalities to filter client software performance metrics.
R040	The monitoring dashboard <i>should</i> allow to setup alerts concerning the performance of client software.
R041	The platform <i>should</i> gather telemetry on all deployment pipelines used by components of the Logistics Platform.
R042	The platform provider <i>should</i> be able to access the telemetry of all deployment pipelines on a central dashboard.
R043	The teams developing modules for the platform <i>should</i> be able to access the metrics measuring the performance and usage behavior of their deployment pipelines.
R044	The platform provider <i>should</i> be able to setup alerts concerning the performance and usage metrics of deployment pipelines.
R045	The teams developing modules for the platform <i>should</i> be able to setup alerts concerning the performance and usage metrics of their deployment pipelines.

Table A.1.: All functional requirements for the logistics platform (aggregated from Chapter 5).

A.2. Enclosed DVD

The enclosed DVD contains a digital copy of this thesis as well as configuration files and instructions on how to setup the platform prototype. A smaller setup for the evaluation can also be found on the disk. This evaluation setup relies on Docker¹ which was used in version 2.2.0.4.

¹<https://www.docker.com>

List of Figures

2.1. Market shares of the different means of commercial freight transportation in the EU	9
3.1. Combined Software Engineering methodology applied in this thesis .	13
4.1. Exemplary logistical process modeled in BPMN	19
6.1. Exemplary structure of a system following the Layered Monolith pattern	36
6.2. Relationships between components in a SOA	41
6.3. The scale cube describes three dimensions of scaling computer software	43
6.4. The flow of an exemplary event through the Logistics Platform	63
7.1. The managed Kubernetes environment hosted on the Google Cloud Platform	71
7.2. Testing the Kafka setup in the Kubernetes environment	72
7.3. The Grafana monitoring dashboard containing metrics collected on the Infrastructure Layer	74
7.4. A basic Grafana monitoring dashboard containing performance metrics of Kafka	76
8.1. The Kafka WebView streaming the events inside the Estonian factories topic	82

List of Tables

5.1. Requirements for information exchange between companies 24

5.2. Requirements for deployment pipelines 25

5.3. Requirements for automated testing 26

5.4. Requirements for continuous integration 26

5.5. Requirements for automated low-risk releases 27

5.6. Requirements for monitoring the business layer 28

5.7. Requirements for monitoring the application layer 29

5.8. Requirements for monitoring the infrastructure layer 30

5.9. Requirements for monitoring the client software layer 30

5.10. Requirements for monitoring the deployment pipeline layer 31

6.1. Communication patterns categorized in two dimensions 44

8.1. Events related to the exemplary workflow including their producer
and subscribers 80

A.1. All functional requirements for the logistics platform 89

List of Listings

6.1. Exemplary event following the event schema for the platform	61
6.2. Schema of the delivery reporting event	62
6.3. Schema of the monitoring event	66
7.1. Authenticating Kubernetes to a private Docker image registry.	71

Abbreviations

API	<i>application programming interface</i>
BPM	<i>business process manangement</i>
BPMN	<i>business process model and notation</i>
CEO	<i>chief executive officer</i>
CRUD	<i>create read update delete</i>
DNS	<i>domain name system</i>
COE	<i>container orchestration engine</i>
HTTP	<i>hyper text transport protocol</i>
IDE	<i>integrated development environment</i>
ILU	<i>intermodal loading units</i>
IPC	<i>inter process communication</i>
IT	<i>information technology</i>
JSON	<i>javascript object notation</i>
K8s	<i>kubernetes</i>
UN/LOCODES	<i>united nations code for trade and transport locations</i>
LXC	<i>linux containers</i>
PRISE	<i>port river information system elbe</i>
REST	<i>representational state transfer</i>
RPC	<i>remote procedure call</i>
SAAM	<i>software architecture analysis method</i>
SOA	<i>service oriented architecture</i>
SODA	<i>service oriented device architecture</i>
VM	<i>virtual machine</i>
XML	<i>extensible markup language</i>

Bibliography

- Abbott, M. L., & Fisher, M. T. (2015). *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise* (2nd). Addison-Wesley Professional.
- Adams, C. (2011). Kerberos Authentication Protocol. In H. C. A. van Tilborg & S. Jajodia (Eds.), *Encyclopedia of cryptography and security* (pp. 674–675). doi:10.1007/978-1-4419-5906-5_81
- Bernstein, D. (2014). Containers and cloud: From LXC to docker to kubernetes. *IEEE Cloud Computing*, 1(3), 81–84. doi:10.1109/MCC.2014.51
- Bohn, B. (2018). Docker Enterprise Edition: Jetzt auch mit Kubernetes | heise online. Retrieved February 24, 2020, from <https://www.heise.de/developer/meldung/Docker-Enterprise-Edition-Jetzt-auch-mit-Kubernetes-4026415.html>
- Bradner, S. (1997). *IETF RFC 2119: Key words for use in RFCs to Indicate Requirement Levels*. Internet Engineering Task Force (IETF).
- Braun, T. (2019). *EventHub Prototype* (tech. rep. No. v0.4).
- Bray, T. (2017). The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259. doi:10.17487/RFC8259
- Butzin, B., Golasowski, F., & Timmermann, D. (2016). Microservices approach for the internet of things. *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, 2016-Novem*. doi:10.1109/ETFA.2016.7733707
- Camunda Services GmbH. (2020). Dealing with Choreography Chaos? Retrieved February 7, 2020, from <https://camunda.com/de/solutions/microservices-orchestration/>
- Chandy, K. M. (2009). Event Driven Architecture. In L. LIU & M. T. ÖZSU (Eds.), *Encyclopedia of database systems* (pp. 1040–1044). doi:10.1007/978-0-387-39940-9_570
- Christensen, B. (2012). Fault Tolerance in a High Volume, Distributed System. Retrieved February 14, 2019, from <https://netflixtechblog.com/fault-tolerance-in-a-high-volume-distributed-system-91ab4faae74a>
- Chung, L., Nixon, B. A., Yu, E., & Mylopoulos, J. (2012). *Non-Functional Requirements in Software Engineering*. International Series in Software Engineering. Springer US. Retrieved from <https://books.google.de/books?id=MNRcBwAAQBAJ>

- Ciupa, I., & Leitner, A. (2005). Automatic testing based on Design by Contract. In *Proceedings of net. objectdays* (pp. 545–557). Retrieved from <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.83.7881%7B%5C%7Drep=rep1%7B%5C%7Dtype=pdf>
- Cloudevents.io. (2019a). CloudEvent specification v1.0. Retrieved March 12, 2020, from <https://github.com/cloudevents/spec/blob/v1.0/spec.md>
- Cloudevents.io. (2019b). Kafka Protocol Binding for CloudEvents - Version 1.0. Retrieved March 12, 2020, from <https://github.com/cloudevents/spec/blob/v1.0/kafka-protocol-binding.md>
- Confluent Inc. (2019). Confluent Community License FAQ. Retrieved February 24, 2020, from <https://www.confluent.io/confluent-community-license-faq/>
- Confluent Inc. (2020). Authorization using ACLs. Retrieved March 27, 2020, from <https://docs.confluent.io/current/kafka/authorization.html>
- Dang, S. (2019). Personal communication.
- Deutsches Institut für Normung e. V. (2011). Intermodal Loading Units.
- Dobbelaere, P., & Esmaili, K. S. (2017). Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper. In *Proceedings of the 11th acm international conference on distributed and event-based systems* (pp. 227–238).
- Encyclopedia.com. (2020). Shipping, Inland Waterways, Europe. Retrieved February 2, 2020, from <https://www.encyclopedia.com/history/news-wires-white-papers-and-books/shipping-inland-waterways-europe>
- European Commission. (2011). White Paper: Roadmap to a Single European Transport Area – Towards a competitive and resource efficient transport system. Retrieved from <https://ec.europa.eu/transport/themes/strategies/2011%7B%5C%7Dwhite%7B%5C%7Dpaper%7B%5C%7Den>
- European Commission. (2015). *On the Digital Transport and Logistics Forum* (tech. rep. No. 1). EUROPEAN COMMISSION DIRECTORATE-GENERAL FOR MOBILITY and TRANSPORT. doi:10.4324/9781849776110-28
- Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T. (1999). *RFC2616: Hypertext Transfer Protocol – HTTP/1.1*. USA: RFC Editor.
- Fielding, R. T., & Taylor, R. N. (2000). *Architectural Styles and the Design of Network-Based Software Architectures* (Doctoral dissertation).
- Fill, H.-G. (2013). What is the fundamental difference between a workflow and a business process in the context of process (or workflow) automation? Retrieved February 19, 2020, from <https://www.researchgate.net/post/What%7B%5C%7Ddis%7B%5C%7Dthe%7B%5C%7Dfundamental%7B%5C%7Ddifference%7B%5C%7Dbetween%7B%5C%7Da%7B%5C%7Dworkflow%7B%5C%7Dand%7B%5C%7Da%7B%5C%7Dbusiness%7B%5C%7Dprocess%7B%5C%7Din%7B%5C%7Dthe%7B%5C%7D>

- %7Dcontext%7B%5C_%7Dof%7B%5C_%7Dprocess%7B%5C_%7Dor%7B%5C_%7Dworkflow%7B%5C_%7Dautomation
- Fowler, M. (2014). CircuitBreaker. Retrieved February 14, 2020, from <https://martinfowler.com/bliki/CircuitBreaker.html>
- Hafen Hamburg Marketing e.V. (2014). PRISE optimiert Zu- und Ablaufsteuerung von Großschiffen auf der Elbe und im Hamburger Hafen. Retrieved April 22, 2020, from <https://www.hafen-hamburg.de/de/news/prise-optimiert-zu-und-ablaufsteuerung-von-grossschiffen-auf-der-elbe-und-im-hamburger-hafen---30987>
- Hafen Hamburg Marketing e.V. (2020). Universalhafen Hamburg. Retrieved April 22, 2020, from <https://www.hafen-hamburg.de/de/universalhafen-hamburg>
- Hapner, M., Burridge, R., Sharma, R., Fialli, J., Stout, K., & Deaki, N. (2015). Java Message Service. Sub Microsystems & Oracle, Sun Microsystems & Oracle. Retrieved from https://download.oracle.com/otndocs/jcp/jms-2%7B%5C_%7D0%7B%5C_%7Drev%7B%5C_%7Dmrel-eval-spec/index.html
- Hermes. (2019). *Optimierungsbedarf in der Supply Chain*. Retrieved from <https://www.hermes-supply-chain-blog.com/wp-content/uploads/2019/04/hermes-barometer-10.pdf>
- International Organization for Standardization. (2014). *ISO/IEC 19464:2014 Advanced Message Queuing Protocol (AMQP) (1.0)*. International Organization for Standardization. Retrieved from <https://www.iso.org/standard/64955.html>
- Ionescu, R.-V. (2016). Inland Waterways' Importance for the European Economy. Case Study: Romanian Inland Waterways Transport. *Journal of Danubian Studies and Research, ISSN: 2392 – 8050, Volume 6*, pp. 180–192.
- Jablonski, S. (1995). On the Complementarity of Workflow Management and Business Process Modeling. *SIGOIS Bull.* 16(1), 33–38. doi:10.1145/209891.209899
- Jana, D. (2006). Service Oriented Architecture – A New Paradigm. *CSI Communications*.
- Karagiannis, D., Junginger, S., & Strobl, R. (1996). Introduction to Business Process Management Systems Concepts. In B. Scholz-Reiter & E. Stickel (Eds.), *Business process modelling* (pp. 81–106). Berlin, Heidelberg: Springer Berlin Heidelberg.
- Kasparick, M., Schmitz, M., Andersen, B., Rockstroh, M., Franke, S., Schlichting, S., ... Timmermann, D. (2018). OR.NET: a service-oriented architecture for safe and dynamic medical device interoperability. doi:10.1515/bmt-2017-0020
- Kazman, R., Abowd, G., Bass, L., & Clements, P. (1996). Scenario-based analysis of software architecture. *IEEE Software*, 13(6), 47–55. doi:10.1109/52.542294
- Al-Khanjari, Z., Alkindi, Z., Al-Kindi, I., & Kraiem, N. (2015). Developing Educational Mobile Application Architecture using SOA. *International Journal of Multimedia and Ubiquitous Engineering*, 10, 247–254. doi:10.14257/ijmue.2015.10.9.25

- Kim, G., Debois, P., Willis, J., & Humble, J. (2016). *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press.
- Klyne, G., Clearswift Corporation, Newman, C., & Microsystems, S. (2002). *RFC 3339: Date and Time on the Internet: Timestamps*. IETF.
- Kotonya, G., & Sommerville, I. (1998). *Requirements Engineering: Processes and Techniques* (1st). Wiley Publishing.
- Kubernetes.io. (2020a). Cluster Networking. Retrieved March 27, 2020, from <https://kubernetes.io/docs/concepts/cluster-administration/networking/>
- Kubernetes.io. (2020b). Kubernetes Documentation. Retrieved March 27, 2020, from <https://kubernetes.io/docs/>
- Larman, C., & Basili, V. R. (2003). Iterative and Incremental Development: A Brief History. *Computer*, 36(06), 47–56. doi:10.1109/MC.2003.1204375
- Leach, P., Microsoft, Mealling, M., Refactored Networks, L., Salz, R., & DataPower Technology, I. (2005). *RFC 4122: A Universally Unique Identifier (UUID) URN Namespace*. IETF. Retrieved from <https://tools.ietf.org/html/rfc4122>
- Leucker, M. (2019). Personal communication. Unpublished.
- Lewis, J., & Fowler, M. (2014). Microservices. Retrieved February 17, 2020, from <https://martinfowler.com/articles/microservices.html>
- Maguire, E., Moreno, K. W., Moreno, H. S., Gagnon, R., McGrath, S., Millar, B., & Pasternak, Z. (2018). Logistics, Supply Chain and Transportation 2023: Change at Breakneck Speed. *Forbes insights*. Retrieved from http://forbesinfo.forbes.com/1/801473/2019-09-23/27ns/801473/8485/Penske%7B%5C_%7DREPORT%7B%5C_%7DFINAL%7B%5C_%7DDIGITAL.pdf
- McIlroy, M., Pinson, E., & Tague, B. (1987). UNIX Time-Sharing System: Foreword. *The Bell System Technical Journal*, 1902–1903.
- Nygard, M. T. (2017). *Release It!: Design and Deploy Production-Ready Software (Pragmatic Programmers)*. Pragmatic Bookshelf, The. Retrieved from <https://www.xarg.org/ref/a/0978739213/>
- OASIS Committee. (2012). Reference Architecture Foundation for Service Oriented Architecture Version 1.0. Retrieved February 18, 2020, from <http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/cs01/soa-ra-v1.0-cs01.html>
- OASIS Standard. (2019). *MQTT* (5th ed.) (A. Banks, E. Briggs, K. Borgendale, & R. Gupta, Eds.). OASIS. Retrieved from <https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html>
- Object Management Group. (2011). Business Process Model And Notation. Retrieved from <https://www.omg.org/spec/BPMN/2.0/>
- Panel, U. S. N. M. C. A., & of Naval Research, U. S. O. (1956). *Symposium on Advanced Programming Methods for Digital Computers: Washington, D.C., June 28, 29, 1956*. ONR symposium report. Office of Naval Research, Department of the Navy. Retrieved from <https://books.google.de/books?id=tLo6AQAAMAAJ>

- Parbel, M. (2018). Apache-Kafka-Unternehmen bricht mit Open-Source-Lizenzierung | heise online. Retrieved February 24, 2020, from <https://www.heise.de/developer/meldung/Apache-Kafka-Unternehmen-vertraut-auf-neue-eigene-Open-Source-Lizenz-4253562.html>
- Philip, A., Afolabi, B., Adeniran, O., Ishaya, G., & Oluwatolani, O. (2010). Software Architecture and Methodology as a Tool for Efficient Software Engineering Process: A Critical Appraisal. *Journal of Software Engineering and Applications*, 03(10), 933–938. doi:10.4236/jsea.2010.310110
- Queßeleit, P. (2020). *Recording of processes and coding recommendations for a digital representation of the Port of Lübeck* (Doctoral dissertation).
- Richardson, C. (2017). *Microservices Patterns*. arXiv: 1-933988-16-9. Retrieved from <http://www.ncbi.nlm.nih.gov/pubmed/20608803>
- Rodgers, P. (2005). Service-Oriented Development on NetKernel- Patterns, Processes & Products to Reduce System Complexity Web Services Edge 2005 East: CS-3. In *Cloudcomputingexpo*, SYS-CON TV.
- Schröder, H. (2020). Personal communication.
- Smith, T. (2017). Are You Building Microservices or Microliths? Retrieved December 6, 2019, from <https://dzone.com/articles/are-you-building-microservices-or-microliths>
- Stomp.github.io. (n.d.). STOMP Protocol Specification, Version 1.2. Retrieved March 2, 2020, from <http://stomp.github.io/stomp-specification-1.2.html>
- Stopford, B. (2018). *Designing Event Driven Systems*. O'Reilly Media, Inc.
- Tabbaa, B. (2019). Anti-Patterns of Microservices. Retrieved December 6, 2019, from <https://itnext.io/anti-patterns-of-microservices-6e802553bd46>
- Tengstrand, J. (2016). The Micro Monolith Architecture. Retrieved December 6, 2019, from <https://medium.com/@joakimtengstrand/the-micro-monolith-architecture-d135d9cafbe>
- United Nations Economic Commission for Europe. (2019). United Nations Code for Trade and Transport Locations - 2019-2. Retrieved February 17, 2020, from <http://www.unece.org/cefact/locode/welcome.html>
- Vermeulen, C. (2019). How to monitor your Kubernetes cluster with Prometheus and Grafana. Retrieved from https://medium.com/@chris%7B%5C_%7Dlinguine/how-to-monitor-your-kubernetes-cluster-with-prometheus-and-grafana-2d5704187fc8
- Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., & Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. *10th Computing Colombian Conference*, 583–590. doi:10.1109/ColumbianCC.2015.7333476
- W3C. (2008). Extensible Markup Language (XML) 1.0 (Fifth Edition). Retrieved March 13, 2020, from <https://www.w3.org/TR/xml/>
- Workflow Management Coalition. (2014). What is BPM? Retrieved December 3, 2019, from <http://www.wfmc.org/what-is-bpm>