TU WIEN Informatics

Microservice Decomposition

A Case Study of a Large Industrial Software Migration in the Automotive Industry

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Heimo Stranner
Matrikelnummer 1326072

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuer: Thomas Grechenig
Mitwirkung: Mario Bernhart

Wien, 9. Juli 2020

Unterschrift Verfasser

Unterschrift Betreuer

TU WIEN Informatics

Microservice Decomposition

A Case Study of a Large Industrial Software Migration in the Automotive Industry

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering and Internet Computing

by

Heimo Stranner
Registration Number 1326072

to the Faculty of Informatics

at the TU Wien

Advisor: Thomas Grechenig
Assistance: Mario Bernhart

Vienna, 9th July, 2020

Signature Author

Signature Advisor

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

Microservice Decomposition

A Case Study of a Large Industrial Software Migration in the Automotive Industry

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering and Internet Computing

by

Heimo Stranner
Registration Number 1326072

ausgeführt am
Institut für Information Systems Engineering
Forschungsbereich Business Informatics
Forschungsgruppe Industrielle Software
der Fakultät für Informatik der Technischen Universität Wien

Advisor: Thomas Grechenig
Assistance: Mario Bernhart

Wien, 9th July, 2020

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

Declaration of Authorship

Heimo Stranner

I hereby declare that I am the sole author of this thesis, that I have completely indicated all sources and help used, and that all parts of this work – including tables, maps and figures – if taken from other works or from the internet, whether copied literally or by sense, have been labelled including a citation of the source.

Vienna, 9th July, 2020

Heimo Stranner

Kurzfassung

Microservices sind eine aufkommende Softwarearchitektur. Im Gegensatz zu einem Monolithen wird in einer Microservicearchitektur eine Reihe relativ kleiner Dienste bereitgestellt, die nur über das Netzwerk miteinander kommunizieren. Sie können unabhängig voneinander bereitgestellt und skaliert werden und arbeiten zusammen, um die volle Funktionalität zu erreichen.

Monolithen leiden regelmäßig unter schlechter Skalierbarkeit und Wartbarkeit. Mehrere Zerlegungen wurden dokumentiert, um die Situation zu verbessern. In industriellen Umgebungen existieren bereits häufig Monolithen, die unter einer Teilmenge dieser negativen Eigenschaften leiden. Befürworter argumentieren, dass nach dieser Architektur die Wartbarkeit von Software im Vergleich zu einer monolithischen Architektur besser ist.

In der akademischen Literatur sind mehrere Migrationen solcher Monolithen zu einer Microservice-Architektur dokumentiert, aber der Ansatz wird nicht immer detailliert beschrieben. Andere Veröffentlichungen stellen Ansätze für solche Migrationen vor, es fehlen jedoch umfangreiche akademische Evaluierungen. Genaue Beschreibungen der verwendeten Ansätze für solche Zerlegungen in Kombination mit groß angelegten Bewertungen im industriellen Kontext sind in der akademischen Literatur selten.

Diese Arbeit beschreibt den verwendeten Ansatz für eine solche Zerlegung in der Automobilindustrie und dokumentiert Änderungen am System. Es werden begrenzte Kontexte verwendet, um zu bestimmen, welche Funktionalität zu einem eigenen Dienst werden soll. Fassaden ermöglichen einen schnellen Wechsel zwischen verschiedenen Implementierungen, wodurch das System ohne unnötige Unterbrechungen schrittweise geändert werden kann.

Alternative Ansätze werden bewertet und Experteninterviews werden durchgeführt, um sowohl die Realisierbarkeit alternativer Ansätze als auch die mit dem aktuellen Ansatz erzielten Fortschritte zu bewerten. Als Ergebnis der Evaluierung stellt sich die Migration als Erfolg dar. Außerdem hat sich die Entwicklungsgeschwindigkeit verbessert.

Schlüsselwörter

Microservices, Monolith, großer industrieller Monolith, Softwaremigration, Zerlegungsansatz, Wartbarkeit, Skalierbarkeit, Expertenbewertung, begrenzte Kontexte, Fassaden

Abstract

Microservices are emerging software architecture. Contrary to a monolith, in a microservice architecture a set of relatively small services is deployed, which communicates with each other only over the network. They can independently be deployed, scaled and work together to achieve the full functionality.

Monoliths regularly suffer from poor scalability and maintainability and several decompositions are documented with the aim to improve the situation. In industrial settings monoliths already often exist which suffer from any subset of these negative qualities. Proponents argue, that following this architecture the maintainability of software is better, compared to a contrasting monolithic architecture.

In academic literature several migrations of such monoliths to a microservice architecture are documented, but the approach is not always described in detail. Other papers present approaches for such migrations but lack large scale evaluations. Precise descriptions of used approaches for such decompositions in combination with large scale industrial evaluations are rare in academic literature.

This work describes the used approach for one such decomposition in the automotive industry and documents changes to the system. To summarize bounded contexts are used to determine which functionality should become its own service and facades allow to quickly switch between different implementations, which allows to gradually change the system without unnecessary disruptions.

Alternative approaches are evaluated and expert interviews are conducted to assess both the viability of alternative approaches and the progress made using the current approach. While still ongoing, the experts agree that the migration is a success.

While the migration is still ongoing, it can already be considered a success as a significant part of the development effort has shifted over to the newer services and the experts prefer working on them. The development speed has also improved drastically.

Keywords

Microservices, Monolith, Large Industrial Monolith, Software Migration, Decomposition Approach, Maintainability, Scalability, Expert Evaluation, Bounded Contexts, Facades

Contents

CHAPTER 1

Introduction

1.1 Problem description

Large monolithic software systems usually have a number of problems. Among the most prominent and pressing ones are increasing difficulties in maintaining and scaling.

Software maintenance is a major part of the software development life cycle. According to Coleman et al. [21] in 1994 there were estimations that about 40 to 60 percent of the total budget of a software system was spent on maintenance. A recent research by Molnar and Motogna in 2017 reports that it is about 50% [52]. Especially, large and complex systems are hard to maintain [69].

Horizontal scaling of monoliths can only be done in discrete steps for the whole application and is not flexible, or difficult to implement efficiently. Relational databases shared among all instances may become bottlenecks [58].

Recently microservices have rapidly gained popularity. They can be seen as a variant of Service Oriented Architecture (SOA) although others claim that it is a new architecture. Technically it can be described as a more fine grained SOA [78]. Primarily, the size and the type of communications between them differentiates microservices from a classic SOA. In a microservice architecture there is a high number of small services and each of them solves only one task [76]. There is no universal consensus how big exactly a microservice should be. The optimal size highly depends on the organization developing them and the problem domain. Microservices communicate over lightweight protocols without complicated logic, commonly referred as „dumb pipes“ [4]. In classic SOAs the usage of complex Enterprise Service Buses (ESBs) that include a lot of functionality, besides transmitting data or messages, is prevalent [7].

A good microservice architecture improves maintainability because one can more easily understand a single microservice in isolation [24]. Every single service can be scaled independently from the other services. Highly scaleable data stores can be used instead

of relational databases. Another benefit is reduced technology lock-in. Different microservices only have to be able to communicate with each other over the network, using a mutually understood protocol. In a monolith all parts of the system are packaged in one executable and there is far less freedom in experimenting with different programming languages or technologies. Using this freedom has some negative aspects as well. Employing a variety of different tools requires greater know how and there is a larger potential for running into problems with one of them. Maintenance can also become more expensive as experts, for all used technologies, are required. A big disadvantage is that this architecture introduces a distributed system, which incurs additional complexity compared to a monolith [7]. In memory calls are more efficient and there is less potential for problems than calls over the network to another microservice.

Creating a well suited software system for a given task is not easy, especially if the system is a distributed system in general or based on a microservice architecture. The granularity of the services need to be determined and the borders need to be defined. It is reasonable to draw borders in places where there are only few and well defined interactions between modules in a system. If there is no prior system or given company structure that can be mimicked, it is challenging to predict what borders will lead to the best results and optimize maintainability, scalability and performance [37].

Conventional criteria for software modularization proposed by academics like coupling and cohesion can theoretically be applied to such microservice decompositions. However they generally do not satisfy practitioners. In practice more diverse characteristics of the software are relevant and not just such one-dimensional considerations. Existing tool support is also seen as far from satisfactory [18].

While monoliths should also be written in a modularized way and not as a large unstructured and hard to understand unit, choosing suboptimal boundaries is much less of a big deal than in a microservice architecture.

1.2 Motivation

Many big corporations already have one or more monoliths in use which suffer from various problems, such as decreased maintainability and poor scalability. It can be desirable to replace them with a microservice architecture but achieving the best results in the most efficient way is anything but easy. Having access to a number of well applicable approaches, that have been empirically evaluated and help making the right decisions, is essential.

Academics as well as practitioners in the software industry proposed a number of different approaches [9, 55]. While there are some academic evaluations of decomposition approaches, they cover academic or small industrial projects that are refactored into a microservice architecture [67]. Decomposing large projects is complicated by having to understand a big system, in order to determine where to draw borders between the newly split parts and the refactoring effort to do so. Most academic approaches which

are focussing on a few formal criteria to decompose monoliths are rarely used in practice. The available tooling support is not considered helpful by practitioners [18].

Conway's law is cited repeatedly in the context of microservice decomposition. It states that organizations are bound to create system designs which resemble their communication structures [4, 24].

Evaluations of proposed approaches in real world scenarios help to compile best practices for practitioners and provide feedback to academics. More data allows them to create better approaches in the future and reach statistically significant conclusions in time. This work aims at providing such an evaluation and thereby increasing the knowledge of what works well when decomposing a large industrial software monolith.

We apply the approach for decomposing a monolith into a microservice architecture to a large Enterprise Resource Planning (ERP) system in the automotive industry. This system is under active development and suffers from poor maintainability and scalability. Its monolithic architecture has been identified as a major problem and as a result the decision to decompose the system in smaller parts and gradually move towards a microservice architecture has been made.

There are a number of properties which the system possesses and they may be important when determining the applicability of an approach to decompose it. Organizational ones are the facts that the software is developed by multiple distributed teams, working from different cities and even countries, and therefore there is a rather high management and communication overhead. This is also discussed by Vallon et al. stating that a variety of synchronous and asynchronous communication methods is employed by teams with asynchronous ones increasing with global separation [70]. The software's operators are not consumers but trained employees at partner companies. Scrum is used as the development methodology. Technical qualities are the used programming language Java and the size of about three million lines of code as well as the used frameworks Spring and Tapestry. Jenkins is used as CI server and source code changes are tracked using Git and Gerrit. There are two major parts of the software dealing with the main process areas of car dealerships, namely the processes of selling a car referred to as *sales* and the processes of providing services to customers afterwards, called *after sales* in this context.

This work will help in systematically analysing the decomposition of large industrial monolithic software systems into microservices. A systematic literature review by Šūpulniece et al. shows a need for more data to develop best practices [65].

Academics can improve the existing approaches or create new ones using the findings of this work and use it in meta studies or systematic literature reviews and in turn develop a comprehensive set of best practices.

For practitioners the gained insights here and even more the compiled best practices in general may be very helpful in determining which approaches lead to good results and are applicable in their situation when decomposing a microservice. Additionally, discovered problems and insights that do not directly stem from the decomposition itself, but are

the properties of the microservice architecture, can also be useful during the development of a new project using the microservice architecture.

CHAPTER 2

State of the Art

This chapter describes the current academic state of the art required to get an idea on how to decompose a monolithic application to a microservice architecture. Firstly the concept of microservices itself is discussed, followed by DevOps. DevOps is highly relevant because manually deploying to the various stages is not efficient and a viable solution will have to encompass some sort of automation. Afterwards reports of successful migrations are given as a general guideline on what works and on the lessons learned from other migration projects. To conclude the chapter a number of academically established patterns for migrating to a microservice architecture is given and discussed.

2.1 Microservices

As stated in the introduction, software scalability and maintainability are crucial topics when deciding on which modern software architecture to choose. Large systems that have tight performance constraints or a large number of requests to handle need to make the right choices in order to handle these topics efficiently. Concerning a better utilization of the available hardware, virtualisation has become ubiquitous [12]. Before the advent of microservices and containerization, most often Virtual machines (VMs) were created by hand and monoliths deployed in them. The drawbacks of decreased productivity and poor modularity are named as popular reasons against a monolithic architecture by Kecskemeti, Marosi, and Kertesz. Such classical VMs can easily be run correspondingly multiple times, but this is frequently too coarse grained, as parts of the system have distinct resource needs. Some parts of the system may be under stress while others are not. Being able to scale the whole application in discrete steps is not as flexible as frequently desired [9]. Splitting monoliths up and creating more fine grained architectures helps with these issues [3, 42]. In general, software systems following the microservice architecture are more scalable and maintainable [4]. Downsides include the more complex interactions between the services over the network which implies a distributed system and the need to define the interaction patterns and borders between individual services.

Splitting up monoliths includes replacing code dependencies with service calls. Representational State Transfer (REST) and message queues are very commonly used as protocols. Hasselbring and Steinacker recommend the usage of asynchronous calls between the services. This way slow or faulty services are less likely to bring down the caller too [39].

Sharing code in the form of libraries is a complex topic. On the one hand, it improves performance compared to a service call. On the other hand, there is the risk that one developer may break another system by changing the shared code. Sharing general libraries, which are not related to the domain and rarely change, is less critical than sharing domain objects. The latter may lead to incompatibilities when one service changes domain objects and upgrades the library, but another service has no interest in these changes and does not yet want to upgrade [9, 39, 40]. Hasselbring and Steinacker also argue that code should not be shared directly between microservices as this introduces unwanted dependencies. Common libraries, especially open source ones, are embraced [39].

One very important aspect, when developing a software system in general and while decomposing a monolith into a microservice architecture in particular, named by Ahmadvand and Ibrahim is security. In a monolithic architecture it is easier to draw clear boundaries of trust and perform the appropriate authorization and authentication there. A very common pattern is a layered architecture, where these security checks are performed on one layer or by a framework and the functionality in need of access control is located on a lower layer. Microservices on the other hand make this more complex. One could check permission on every call but this degrades performance and is not always appropriate when the client is another microservice. A good migration strategy has to deal with these issues [3].

As discussed in more detail in section 5.1.3, transactions are not as straight forward in a microservice architecture if they span across multiple microservices. Distributed transactions are slow and can be a performance bottleneck. Database locks are active for longer durations, since all services first have to agree to commit over a network. This can servilely limit the transaction throughput. Loose coupling and going without distributed transactions is highly desirable but not always possible. Using a waterfall style development methodology for decomposing a monolith, where performance testing is done at the end after a lot development work has been done, is a bad idea. Performance issues may simply be caught at a very late stage and the compensations may be very expensive. Knoche proposes first specifying performance constraints for the relevant actions, considering multiple alternative modernization paths using simulations and choosing the best one [43].

2.2 DevOps

Balalaie, Heydarnoori, and Jamshidi discuss connections between DevOps, and Microservices. DevOps is the approach to combine the development and operation of a software system. This decreases the time between implementing a change and running the new version in production. In practice, this includes automating testing and deployment

processes and automatically setting up environments for the software to run in instead of manually [39]. Deploying a new version of a large monolith takes much more time and may incur a significant downtime, compared to just redeploying one of many small microservices. The goal of fast delivery of changes in DevOps is therefore impeded by a monolithic architecture [8]. Microservices commonly have shorter release cycles because a proper Continuous Integration (CI)/Continuous Delivery (CD) setup makes this easy and there is a desire for fast delivery of changes from a business point of view, as long as the stability is not aversely affected [3]. Since microservices have more services, which need to be deployed, using this architecture without automating, deployment is not efficient. Therefore Microservices strongly encourage the usage of DevOps approaches including CI/CD [8, 9]. Microservices are becoming the prevalent architecture among those leveraging these automations [10].

One general problem when deploying software is that it may behave differently depending on the environment. Sometimes this difference is desired. The test system of a payment provider for example should not perform transactions with real money. Differences between environments are often not desired and one wants them to behave as close to the production as possible, in order to catch as many bugs as possible. „It works on my system“ may be true, but is not helpful. Containers such as Docker provide the ability to create isolated environments that can easily be shared and behave consistently. Dependencies are installed automatically and come bundled in a container image. The approach is similar to virtualisation, but more light weight as the operating system kernel is shared among the host and the running containers, which is not the case for virtualisation solutions [9, 39]. Balalaie, Heydarnoori, and Jamshidi recommend their usage to minimize the possibilities for the environments may behave differently [9].

These containers should be automatically deployed on clusters, using adequate dedicated management tools. Deploying a large number of services manually is not efficient and error-prone [9].

2.3 Successful migrations

Kecskemeti, Marosi, and Kertesz report about efforts to modernize the software architecture behind a system that creates Infrastructure as a Service (IaaS) environments. The paper describes the infrastructure required to operate with microservices in an efficient manner. A crucial part of the system is the automated creation of VMs or containers. Upon creating a VM the dependencies of the software running inside have to be known, otherwise they can not be installed automatically. Previously developers had to create VMs from hand and these dependencies were not always directly available in a machine readable format. It did not scale well for many services [42]. In the newly automated approach tools like Puppet and Chef are used to set up the desired environment inside a VM. As the new alternative Docker is used, Kecskemeti, Marosi, and Kertesz report that the migration was successful [42].

Balalaie, Heydarnoori, and Jamshidi share experiences incrementally migrating a backend

software for mobile applications to a microservice architecture. For them the main motivation to migrate was the technology lock in with the old system. They wanted to use a variety of different tools which are not easily integrated into one monolith. Previously they build a mobile backend, providing general create, read, update and delete (CRUD) operations for user defined data structures by wrapping a relational database [8]. A chat has other requirements. Relational databases do not scale very well, as transactions are very resource intensive and distributed transactions even more [39].

Additionally, they adopted DevOps. Used technologies include Spring Boot, and continuous integration and delivery as well as an edge server were used. The migration was successful [8].

Balalaie, Heydarnoori, and Jamshidi report that in their migration the services were cut along existing boundaries between domain entities. This follows the Bounded Context (BC) pattern employed by Domain-Driven Design (DDD) [8, 9]. The BCs should be incrementally decreased in size. At the beginning the monolith constitutes one BC and these are iteratively split up to match smaller domain concerns [9]. A BC should have relatively few interdependencies with other BCs. Since service calls have worse performance than in-memory calls their number should be limited to achieve a low latency of the overall process [24].

Alternatively, the ownership of data can be used as determining factor where to split the monolith into different services. Since transmitting large amounts of data between services is not efficient, this makes sense particularly in data centric applications. Services may obtain copies of the data owned by another service, but only the owner is responsible for persisting changes. If the data can be separated into different responsibilities this approach can be very successful [9].

In order to not affect existing customers negatively Balalaie, Heydarnoori, and Jamshidi report that the migration had to be as seamless as possible. Therefore an incremental approach was chosen. The first crucial step reported does not concern the decomposition itself but enables it to be efficient. It is setting up a meaningful CI system which later enables CD. In the concrete setup GitLab, Jenkins, Artifactory and the Docker Registry were used to set up a CI pipeline. A later step replaced integration tests of the whole system with consumer-driven contracts. This allows to test every service in isolation and is very important to test changes limited to one service quickly and efficiently, instead of running a much larger integration test of all services. Deployments can then be confidently performed for one service at a time and less communication is required between teams, following this approach. The services were also placed in separate source code repositories, which allows the intuitive configuration for CI builds for each service on its own. Having the built Docker images in a registry makes the deployment to different environments very easy. A developer can run everything locally just with docker-compose, which will start all containers given a configuration file. In production Kubernetes was employed to run containers across a cluster of CoreOS instances [8].

A further step before the modularization of an existing system itself can begin, is to

get to know the current architecture or to make sure the documentation is accurate. With this information better decisions where and how to split the monolith can be made, resulting in a more efficient modularization and better cohesion [9].

An edge server was introduced in order to mask internal implementation details and provide consistent endpoints and interfaces to customers [8]. This is just one example of additional helping service that makes working with microservice architectures more manageable. Others include service discovery and load balancers [9]. Similarly Alpers et al. proposes the usage of a general Application programming interface (API) gateway. It can handle authentication issues and limit the concurrent load on single services [4].

Further Service Discovery, Load Balancers and Circuit breakers were added to the project described by Balalaie, Heydarnoori, and Jamshidi, in order to improve the architecture and make it more performant and resilient [9]. Some sort of service discovery is needed when clients call servers directly, in order for the client or a load balancer to now where the service handling a request is located [9]. Message queues decouple senders from receivers of messages and thus solve the problem without the need for service discovery. Monitoring for every service is required too to provide confidence that everything is working as intended [9]. It also allows to scale services up or down automatically depending on the current load [39].

Circuit breakers detect faulty services by monitoring them and prevent further calls from being routed there. This reduces the number of total calls made to a faulty service. Once it becomes available again the circuit is again closed and calls are made there [9].

They used the common stack consisting of Elasticsearch, Logstash and Kibana. Statistical data was gathered and used to detect later anomalies [8].

On the organizational side Balalaie, Heydarnoori, and Jamshidi report of cross-functional teams being better suited for developing microservices compared to dedicated teams for phases in the classical software development life cycle such as development, quality assurance and operations. Every cross-functional team is responsible for a service [8]. Alpers et al. also stress the importance of one team being able to fully understand and support their service and thus being responsible for it [4].

Balalaie, Heydarnoori, and Jamshidi recommend against the usage of service versioning and favour the tolerant reader pattern. This requires for the versions to be somewhat compatible. Consumer driven contracts can check if the versions are compatible enough to perform the desired actions [8, 44]. The idea is that when reading data, services should be lenient with what they expect and handle data that does not follow the specification exactly as well. Reading an additional field that is unknown to the reader should for example never lead to a crash.

In the case described by Balalaie, Heydarnoori, and Jamshidi the configuration of each microservice was externalized and provided by a dedicated configuration server. This is a good practice since it allows to configure one service differently, depending on its context. The context dependent configuration should therefore not be in the source

code repository. Using this pattern, services can be reconfigured without the need for a redeployment [8, 9].

While Balalaie, Heydarnoori, and Jamshidi report that the boundaries between the newly separated microservices were the given boundaries between the domain objects, Ahmadvand and Ibrahim put much more emphasis on involving all stakeholders on getting the boundaries right [3, 8]. The point is that if only the developers and software architects decide on this issues, some concerns may be missed [3].

Ahmadvand and Ibrahim propose a rather high level but formal approach of specifying a system as a collection of functional and non-functional requirements. The focus among non-functional requirements in this context is on scalability and security specifications. Security requirements are described by misuse cases which the system must not allow. Requirements may have dependencies between them, which correspond to network calls in the microservice architecture. There is a weight associated to these dependencies which describes the expected frequency of calls. Requirement engineers then try to determine the needed scalability for each of the requirements, their impact on the system and define the borders of the microservices by balancing the estimated scalability needs, the dependencies and the security concerns. The approach starts by defining a new microservice, starting from a requirement and adding dependencies into it if there is a high negative impact if not doing so. This impact may be lost performance caused by many calls or by the overhead, generated by strict security constraints [3].

Hasselbring and Steinacker report on migrating business software for the online shop otto.de. They also discuss the ideal size of a microservice and the connection with DevOps. Conway's law is also described as a guide on how to structure the services and scalability is not only considered in the number of users, but also developers and different teams. They decomposed the monolith in different verticals. Those encompass one business area and are responsible for all layers like user interface and persistence of that area [39].

A large industrial ERP systems decomposition is documented by Šūpulniece et al. They stress the problems of poor maintainability and the beneficial aspects of modularization. The system is written in Delphi and they employed an existing parser that compiled graphs from the source code on which further analyses are based. A tool is developed and used to abstract the source code as a graph and further calculations were made on how to decompose the software [67].

Knoche and Hasselbring state scalability as well as maintainability as main reasons to develop systems as microservice architectures or migrating to it. When migrating to this architecture, an incremental approach is generally preferred, but total rewrites are also possible. In the presented case study the decomposed system is an insurance company's customer management application, consisting of about one million lines of Cobol code. The architecture grew extensively over time and some integrated applications accessed the underlying database directly. This adds quite a lot of additional complexity to the migration, as the current architecture of the system has to be reverse engineered and the

new microservice based application has to be compatible to the old monolithic one. An important step in the process was the definition of both external and internal service facades. Then the clients had to be adapted to use these facades especially in the cases where they previously directly accessed exposed implementation details. Knoche and Hasselbring report that the migration is still in progress, but already successfully running in production and so far the results are promising [44].

Gouigoux and Tamzalit report on the technical insights gained by migrating the core business software of a French software editing company from monolithic architecture to microservices. The main reasons for the migration were the desire for a modern web frontend, allowing economic long term evolution and to enable low cost integrations with third party software systems. Microservice architectures are compatible with all three concerns. Crucial questions are about the most appropriate granularity of the services, the deployment and efficient orchestration. They balanced the costs of deployments against the costs for quality assurance, when deciding on the granularity. Good automated tests reduce the costs for regressions, but have to be written causing non negligible costs. Automated deployment has to be set up but decreases the costs of further deployments and also decreases the risks that come with a manual deployment such as human errors, when copying or setting something up wrong. Overall, they assume the cost for quality assurance sinks, when there are more fine grained services as the functionality can be tested in much more isolation and before a deployment only a small set of features has to be tested. The costs of the deployments itself rise of course when the architecture is more fine grained as even in the automated way more jobs to deploy have to be set up and maintained. They stress the importance of using automated deployment when employing a microservice architecture as the costs for performing them manually would be too high in the long run. Overall, they report positive findings [33].

Microservices are a topic under active research and design patterns are available as best practices [77]. Deviations from the given best practices and the results stemming from the deviations are hard to measure. Since the systems dealt with in practice are large, automated approaches, which find such deviations, are desirable. Zdun, Navarro, and Leymann propose to formally check if services are independently deployable and if no shared strongly coupled dependencies such as shared internal libraries or reused models exist [77].

Similarly, Levcovitz, Terra, and Valente propose creating a graph from the different functionaries of a monolithic system. They consider the layers of facade, business functions and database tables. Each facade, business function and database table is a node in a graph and has outgoing edges to all other parts of the system used by it. The graph can be used to decide where to split the monolith. All edges between the new borders have to be replaced by service calls. They should be wrapped in API gateways in order to make service calls transparent for the calling system [48].

Fritzsch et al. compare different approaches on how to decompose a monolithic system. If the monolith consists of different technologies and perhaps is even written in different languages, the borders between these parts are good candidates for the microservice

architecture. Legal or cultural barriers can also be sensible borders in such a system. Very commonly the microservice architecture is structured according to the business capabilities. This is referred to as bounded contexts and can be done with respect to use cases and resources. The idea originated from DDD. The authors point out that there is no universally applicable algorithm on how to optimally split a monolith into microservices and it is thus similar to an art form [30]. There are some approaches that automatically generate some insights on how a monolith could be split. They describe four basic ideas behind this. The first idea is static code analysis, where only the source code is analysed and a suggestion is made based on that. Secondly, there are approaches that additionally use metadata such as specifications in the form of Unified Markup Language (UML). A third class of approaches checks the workload of the system and what performance constraints and data dependencies are in a system and places closely related parts in the same microservice. The last discussed type of approach is more dynamic and describes the system as a whole. It allows to iteratively optimize the current architecture according to multiple objectives such as performance [30].

Gysel et al. discuss Service Cutter an approach distilled from industry experience and prior academic approaches on how to decompose monolithic architectures efficiently into microservice architectures. An undirected weighted graph is constructed based on the code and used to determine which parts depend strongly on each other. Users can specify which criteria are especially important to them and the framework calculates good ways to cut the service, based on these preferences and various input machine readable software artefacts and models of the architecture. Clustering algorithms are employed to find potential cuts in the graph [35].

2.4 Established patterns

Balalaie et al. discuss that microservice decomposition is a complex topic and there are no universal approaches that lead to good and efficient results for every project. Therefore, they propose a repository of patterns that can be used and that are proven to be useful under some circumstances [10].

The patterns are the same as presented by Balalaie, Heydarnoori, and Jamshidi [9, 10]. The latter paper additionally presents positive findings of using the given patterns including medium term retrospectives.

CHAPTER 3

Research questions

A goal of this work is to increase the academic knowledge of how decomposition decisions, or more broadly decomposition approaches, affect the quality and effectiveness of the decomposition itself and the resulting software when applied to a large industrial monolith.

The author has access to such a system and is part of the ongoing development and modularization effort. The software at hand is an ERP system in the automotive industry and is under active development.

It is the author's believe that rather specific academic approaches are not feasible in every organization and have to be adapted to fit into an organization's style of development as well as the organizational structures. More general approaches may be feasible in a huge variety of of organisations and circumstances, but the details are not specified and may have to be concretised.

Because the system at hand suffers from poor maintainability and scalability, management made the decision to decompose it into a microservice architecture. We describe the used approach in detail in section 7.

There are two main research questions, one of which is refined in more detailed questions. We shall mainly focus on the quality aspects of maintainability and scalability because improving them is a major motivation to decompose large industrial systems and were crucial in this case.

- **RQ1:** What are the results of using the current approach to decompose the system at hand?
  - **RQ1a:** How is the maintainability affected?
  - **RQ1b:** How is the scalability affected?
- **RQ2:** What alternative approaches exist to decompose a large industrial monolith?

CHAPTER 4

Methodology

Problem
The problem described in more detail in section 1.1 can be summarized as a lack of well established and evaluated approaches on how to decompose an existing large industrial monolith into a microservice architecture, in order to improve maintainability and scalability. This work is part of the ongoing effort to improve academic knowledge in that problem domain by providing a case study. Case studies are well established in the research community [60].

Research questions
The research questions are listed and explained in chapter 3. They are mainly about evaluating the current approach, evaluating its properties, regarding the desired improvements to maintainability and scalability and assessing potential alternatives.

Solution
In order to answer the research questions about the current approach, a number of developers apply it to the system at hand and we observe changes to the system. Over a period of more than two years the author is part of the group of developers tasked with implementing changes and new features in the system as well as to modularize its old monolithic architecture into a microservice architecture. To check if the existing approach is indeed well suited to large industrial monoliths, the approach is applied to such a large industrial monolith, following the case study methodology [60].

The project, to which the chosen approach is applied, is fixed as it is the only large scale industrial software system available to the author at the point of writing.

During the stated period we observe the progress made. This allows us to check if the approach is indeed well suited to the decomposition of large industrial monoliths following the case study methodology [60]. We also measure parts of the research questions, which can be directly measured, like the duration of a build on the CI server.

Statistics, based on the source code management system and CI server builds, are created and these directly measurable results are considered. This is presented in chapter 7 and section 7.2.

In order to answer the research question about alternative approaches, these need to be found which is done by performing a literature review.

Like any system, the system at hand has some properties. For example, there is no formal specification available, which is very common in the industry according to Ozkaya [57].

These properties rule out some approaches. Criteria required for approaches, which could be used to decompose the system at hand, are established and alternative approaches are explored in chapter 8 by performing a literature review. The currently used approach fulfils both, organizational and technical properties of the system at hand.

We assess the approaches for decomposing monoliths into microservices found by searching for „microservice decomposition approach“ on Google Scholar. Their applicability for decomposing the system at hand is determined, according to the criteria defined in section 8.1 considering the properties of the system at hand along with the results reported by academics. To limit the search extend, only the 25 most relevant entries from the search query are considered.

First, the search query is performed and the 25 first results are checked for their applicability to the problem. Additionally, current practices in the company are taken into account because it is not in the power of the author to change the process in a major way. Many practices are due to management decisions far out of the hands of developers or technical architects. Afterwards potential duplicates are removed and commonalities are found. In order to minimize bias, the assessment process is reviewed and feedback used to adapt the assessment process.

For approaches that are not applicable to the system at hand, we note which properties prevent the applicability. This is important to ensure academic scalability. Criteria which are specific to the system at hand lead to less scalable results compared with very general criteria which are true for almost any project.

Evaluation
In order to evaluate the solutions to both research questions, expert interviews are conducted. Such interviews are well established methods in research [72].

The author works as a software developer in a team tasked with modularizing the software. This team applies the selected approach. The findings and results are documented, especially when applying the approach is not possible for any reason including organizational and technical reasons.

To evaluate the results, structured expert interviews of team members working on the same software are conducted. The interviews are limited to the experts involved in the project. They are qualified to give project specific feedback and not just general feedback to the approach, as already found in the literature for various other systems. External

experts without access to the system at hand would not be able to comment on the changes of said system.

The experts are also asked about potential alternative approaches and their assessment of those.

The precise questions which the experts are asked are listed in section 9.2.

The team is distributed and the interviews are executed over mail. The answers given by the experts and conclusions based on them are used to compile the results, presented in chapter 10.

The verbatim answers given by the interviewees are documented in appendix 13.

A conclusion is presented in chapter 13 and sums up the academic contribution of this work, the feasibility of the approach and an assessment of its qualities compared to other approaches.

There is no focus on a metric based evaluation. For such an evaluation a fitting metric would need to be found. Some metrics like coupling and cohesion are known to academics but there is no consensus if they are truly of primary concern regarding maintainability [18]. Finding proper metrics could be a work on its own. Scalability can be measured using load testing. For the system at hand such tests do not exist. Creating them, maintaining them during the modularization period and continually running them is not feasible in the scope of this work.

CHAPTER 5

Concepts

The intent of this chapter is to provide a basic understanding of all the concepts required, in order to understand the following approach selection and evaluation.

In order to achieve this, first a general overview of software architecture and its purpose is given. In that section a more thorough overview over monolithic architecture, SOA and microservices is included. The focus lies, following the topic of this work, on microservices.

Microservices are described in general in section Description and in section History the historical development is discussed. For microservices integration is a central topic as more different services have to be integrated compared with a monolithic architecture. Different approaches on this are presented in section Integration technology. The most prominent style of integrating different services REST is discussed in more detail in REST.

The *micro* part of microservices already suggests something about the desired size of a microservice. The more subtle details are presented in section Size. Apart from the size, also the question on where to split microservices is a crucial topic and is topic of section Grouping of functionality.

Loose coupling allows a lot more flexibility and technical heterogeneity which is the topic of section Technical Heterogeneity. While a lot of flexibility is generally seen as a good thing, not everything that can be done should be done. Shared libraries are one example that are critically discussed in the academic literature in this context. Section Shared libraries gives an overview of this topic.

Manual deployment and monitoring is hardly possible with a microservice architecture. Sections Deployment and Monitoring discuss these topics in more detail. Similarly transactions, testing and application security have to be thought about again and in the context of microservices are the topic of sections Transactions, Testing and Security.

The second part of this chapter is about software maintenance. This part is highly relevant because a migration from a monolithic architecture towards a microservice

architecture, while productively using the software, is a sort of software maintenance. This is discussed in more detail in section 5.2. Special consideration is given to the high costs of this phase of software usage and the difference an agile approach makes. More generally, the different types of maintenance are explained and hints like code smells discussed that indicate possible problems with future maintainability.

Section 5.2.5 is about the related topic of how to modernize legacy software and especially on how to use modularization for this task.

Maintainability and scalability are two major problems encountered in real world monoliths. Since these problems are also encountered in the case study subject, the last part of this chapter is about scalability (section 5.3).

5.1 Software architecture

From a historic point of view the size of software systems grew and continues to increase. Often multiple systems need to be integrated to form even larger ecosystems where data is shared and calculations are performed decentrally. While previously only some calculations where made by computer programs, now the systems are much more integrated and handle not just the algorithmically complex computations, but the whole life cycle of data. Already in 1993 Garlan and Shaw discussed the trend that algorithms and data structures were no longer the major design problems in most systems. Because these systems are growing, more high level concerns need to be dealt with as well if not more prominently [31]. More recently Venters et al. report of the same trend and mentions that modern large scale systems with no scheduled downtime are common. Software architecture allows to discuss core quality concerns of a given system. Maintainability for example is one concern that highly depends on the right architecture [71].

When a software system consists of more than one algorithm the question arises on how they interact with each other and how the data is transported between them. With the number of single algorithms being part of a system rising this question becomes even more critical. Software architecture deals with determining performant and sustainable solutions to these problems. As when constructing a building, architecture deals with the high level concerns and how everything is structured in a software system [31].

In order to better understand and visualize architecture, diagrams are often employed. UML is one standardized language that describes and formalizes such diagrams. Such a formal and standardized description is important so that the diagrams are used in a uniform way and everyone encountering a diagram immediately knows what the syntax is about and can mentally focus on the meaning of the given diagram, instead of the intricacies of the syntax of a given diagram. Often there are informal texts that describe the architecture of a system. This follows less strict rules and allows to intuitively give details about an architecture [31].

When comparing the architectures of different systems there are often common patterns between them that have been further studied in academic literature and by practitioners.

These patterns may deal with issues of the system structure, synchronization and data access as well as protocols for communication. Studying these patterns and assessing their impact on a system allows to reuse them in exactly the right situations, where one can expect positive results [31].

Getting the architecture of a newly build system right is crucial, especially if the system is large. For small systems or if the problem domain is very easy to understand it is most efficient to just start implementing instead of wasting time on obvious design decisions. Once the system gets more complex and larger, the instinct of the developers alone are no longer sufficient and critical thoughts need to be employed on how the system should be designed. If the system gets really complex thinking about the problem may not be enough any more and experience with similar situations becomes a very valuable tool [5]. Choosing an ill suited architecture can dramatically increase further costs and may lead to project failure [31].

Historically one can observe a trend from lower level instructions to higher level programming. In the 1950s computers were programmed in machine language. Gradually higher level languages evolved, abstracting away more and more details of the underlying machine, often sacrificing control [31]. Using many languages, which are considered high level today, one does not need to take care of memory management for example. These abstractions made it feasible to create larger software systems, as the developers could focus on higher concerns and not deal with all the details involved.

Software architecture establishes restrictions on how the program may be written. A simple architecture could divide the program in multiple layers where only functionality of the own layer or of the layer immediately below may be used directly. Such a layered architecture is common and helps to structure a software which leads to better maintainability. Its commonality also allows to reasonably expect new programmers to quickly understand it. For example, a desktop application with a graphical user interface that persistently stores some data could be organized using a layered architecture. Three layers are common and may be associated to the Graphical User Interface (GUI), the service layer and the persistence layer. The uppermost layer, the GUI-layer is responsible for showing the data to the user, accepting user input and calls the service layer. The service layer provides logic and algorithms and calls the persistence layer which in turn gets the data or stores updates to it. This architecture allows one layer to be replaced by other technologies rather easily. Often there are interfaces defined that specify how exactly the functionality exposed by one layer works. If for example the underlying type of database is changed, the persistence layers interface should not change and thus the upper layers should continue to work without changes.

In general, it is one goal of software architecture to allow low cost maintainability. The current needs should be met without sacrificing the possibility to easily make changes in the future. If maintenance operations are easier to perform, the costs will generally be lower. Therefore simple maintenance is very desirable [71].

Today software systems are often complex and consist of multiple components that have

to work together to fulfil useful and sometimes even safety critical purposes. These interdependencies makes the system inherently more complex. The failure of one component should affect the others as little as possible and large chain failures have to be prevented. Good system architectures will take care of this problem [71].

5.1.1 Monolith

Monolithic architecture is often named in contrast to microservices or even in the context of converting a system from a monolithic architecture to a microservice architecture. Following this architecture there is one executable that contains all the functionality and is typically created from one large codebase [30].

Since there is just one executable, the options for integrating different programming languages and frameworks is limited. This makes leveraging new and exiting technologies in the right places more difficult even if they were a perfect match for some use cases. Balalaie, Heydarnoori, and Jamshidi report of them wanting to integrate a chat server with a monolith written in Java. The encountered difficulties and inefficiencies were a motivation to migrate towards a microservice architecture [8].

Using many different technologies however is not necessarily only a good thing. If a firm establishes a company wide default technology stack it is much more likely that developers can easily switch between projects and immediately be productive as they are familiar with the used technology [7]. A lower number of experts is required and if languages and platforms with high availability of programmers are chosen, the costs for the workers will be lower.

Often monoliths provide a lot of functionality, but also have many dependencies and configuration settings that have to be set up in order for it to work properly. Since there is only one deployable, the deployment is often done by hand and thus prone to human errors. Recreating the same environment is relatively hard and time consuming in these cases, compared to the automated DevOps approach where the setup is fully automated. Using a DevOps approach for monoliths is possible and often a good idea to solve these issues. Nevertheless it is not as common as with microservice architectures where it is almost necessary since it is very uneconomic to frequently deploy large numbers of services by hand.

If only few deployables exist which is the case for a monolith, the manual approach to software delivery is much more feasible [78]. Nevertheless, DevOps helps in reducing unnecessary effort and creating more reliable deployments for monoliths. For a microservice architecture the need for reliable deployments and the number of required environments that need to be set up equally is generally larger and so are the benefits of employing DevOps.

Since there is just one executable for a monolith, it can only be scaled horizontally by spawning more instances of the same monolith. In many cases only parts of the application are under stress. Running everything again is then not efficient, as resources like memory are wasted recreating underused functionality [73].

Calling functionality within a monolith is very fast compared to having to perform a network request and possibly wait for an answer. For this reason the raw performance of a request served by a monolith can be expected to be higher than the same request served by a microservice architecture that has to perform network calls. „Chatty“ microservices that have lots of dependencies on other services and perform many calls are not desirable for a microservice architecture however and are a sign of poor coupling.

A monolith is not necessarily a distributed system. Many monoliths interact with external systems, but the network communication is not prevalent in internal communication. Distributed systems come with some disadvantages, such as that the network should not be expected to be reliable and errors of other components should be expected. Monoliths do not have to deal with these issues when performing an internal call [55]. Monitoring is also easier for non-distributed systems as the logs can be gathered and the measurements be performed in one place only.

5.1.2 Service oriented architecture (SOA)

There is no clear consensus on whether microservices are a subset of SOA or a new software architecture [78]. This subsection deals with the classical SOA and explicitly not with microservices.

SOA is an architecture that has services as crucial parts. The proper interactions allow the system to work as one application delivering value to stakeholders. Services can be composed to form larger components [56].

According to Arsanjani, a SOA consists of three major components. There is a service provider, a service consumer and a service broker. These three entities may be operated by different organizations and the possibility to integrate different services provided by distinct teams or organizations is one key idea behind the architecture [56].

A service provider, in addition to providing the service itself, is also responsible for publishing a description to the service broker who maintains a registry of available services. The service consumer accesses the registry provided by the service broker and using the provided information it is able to connect to or to bind the service. The idea is to create an architecture in which the individual services are loosely coupled which allows easier and faster changes to the services [6]. Reusing existing services is highly desirable as it saves effort, since common functionalities only need to be implemented once.

Using a registry provides service discovery and allows to dynamically use other services that perhaps where not even known at the time of writing the service consumer. SOA enables dynamic composition of services when needed using the registry which hosts descriptions of services. These descriptions are formal enough to allow a consumer to directly programmatically use the corresponding service on a syntactical level.

There are public registries that are especially suited for integrations across organizations, but this is not widely used today [6]. The reasons for this include that businesses are heterogeneous and not all business decisions map well to data provided by public

registries. People also commonly prefer to only do business with known organizations, perform due diligence and only depend on services after legal contracts were arranged. Using a full blown registry can be overkill for in house services or the services of a few select organizations. Less formal descriptions such as wikis are commonly used for these purpose [13].

SOA does not make statements about the size of a service and it is absolutely valid to provide the functionality of a very large legacy system as a service. By providing it as a service it is easily accessible to others and that is what this architecture is about [6].

Monitoring and making sure a service is providing a certain quality of service is a first class concern and helps when selecting services to bind and invoke. Concerns include security, availability and performance [6].

According to Orlowska et al. typical web services use Simple Object Access Protocol (SOAP) to communicate and send Extensible Markup Language (XML) messages [56]. To describe a web service Web Services Description Language (WSDL) can be used which provides consumers with the needed data to directly communicate with the provider [56]. An alternative path for integration is an ESB which further decouples the service and the client and forwards the communication. Additionally, an ESB can provide a wide variety of functionality such as converting between message formats [6].

Today, the term web service is often used to describe any kind of service that is accessible over the internet and not just services using the technologies stated. REST services are one prime example [54].

Many promises of SOA were not fulfilled in practice. The prime example is that service registries do not play the role envisioned by academics. Public Universal Description, Discovery, and Integration (UDDI) registries never played an important role in the industry. The envisioned global integration of services has not become reality [76].

The idea to loosely couple different services over well defined network calls however is very much alive. SOAP APIs are widespread, especially in legacy business applications. WSDL is also used to describe the operations such a service provides, but the retrieval happens mostly manually, via company specific or local service registries. More recently, using Remote Procedure Call (RPC) or REST style APIs is gaining popularity, which still follows the same basic idea, to easily allow heterogeneous systems to interact over the network [76].

5.1.3 Microservices

Description

A microservice architecture is a software architecture for distributed systems. Its architecture is similar to, or according to some, a subset of SOA [78].

Fowler describes microservices as an architecture to build applications as a set of small services that are independently deployable of each other and communicate using light

weight protocols. In contrast to SOA there is no complex logic in the communication handling technology as provided by an ESB. Often REST is used as a mean of communication between services as well as with the clients [28].

In contrast to a monolith the functionality is provided by not just one application but by multiple. If a subset of the services does not work, the rest should continue to provide value and work in a potentially degraded mode.

A microservice architecture intends to make it easy to deploy local changes that only affect one service. In contrast, using a monolithic architecture the whole monolith would need to be deployed and potentially tested [28]. However, when changes span across multiple microservices, they all and the APIs between them have to be changed. This can easily mean more effort than if the same change would be made to a monolith where there is no need for explicit APIs. This implies the need for well chosen boundaries.

Scaling can also be done in the same granularity as the deployment. It is common that only parts of an application have high resource usages. In a cloud environment, or when using containerization together with a suited orchestration solution, the parts under stress can often be easily and automatically scaled to meet the demand [28]. Having smaller services often implies having less dependencies per service. Generally, this makes deployment easier, as the need for highly customized and complex setups for all potentially conflicting or incompatible dependencies is reduced.

Microservice architectures are mostly used together with a DevOps approach. There are a lot of independent services and the time to deploy one change commonly decreases. This is seen as a very good thing as faster delivery of features to customers is a common desire of businesses. With the help of DevOps, which reduces the operations overhead, teams that develop a service are commonly also put in charge of operating it and providing support in case of problems. This has two benefits. Firstly, the team needs to answer directly for problems with the service, which leads to a greater feeling of responsibility for the component. Secondly, the developers of a component are very well suited to find bugs as they know all the details about the component and do not only have a black box view on it. Additionally, the team is very motivated to make the service monitoring easy because it is in their own interest and not just an improvement for somebody else [28].

History

Microservice architectures have recently gained popularity [24]. There is a certain hype around the technology and according to Fowler some practitioners consider it to be the default architecture for newly created systems [28].

Google Trends also supports these claims as observable in Figure 5.1.

Classical languages prominently used on a server such as Java, C or Python produce single artefacts that mostly run in one process. Modules are not independently executable, but are compiled in one binary. This lends itself very well to the creation of monoliths [24].

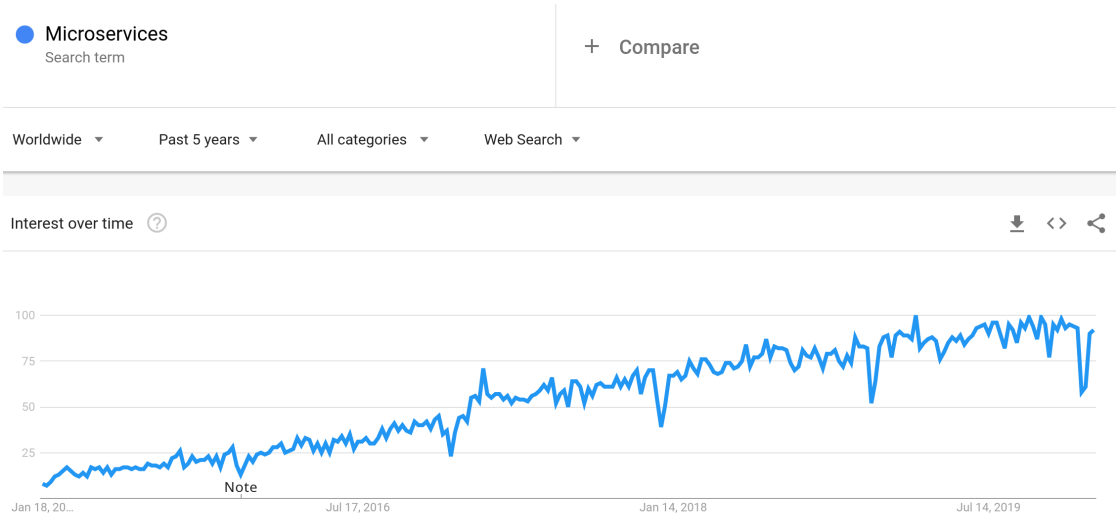

Figure 5.1: Google Trends analysis of the „Microservices“ topic for the last 5 years

Microservices were introduced as means to counteract the negative sides of monoliths, including improving the often poor maintainability and scalability [24].

Large software systems are generally harder to create than smaller ones. Size alone makes quickly changing the source code more challenging as one needs to think about more details to make appropriate changes. Well structured software repositories make this easier. Software architectures were invented and tried over the course of time to create well maintainable software [24].

Today, object oriented software is ubiquitous. Object Oriented Programming (OOP) has made creating programs easier by structuring them into units that interact to achieve the desired functionality. Service oriented computing, as first made prominent with SOA, has taken the ideas and even applied them on another level. The system consist not just of a number of units and classes but different services that interact with each other, primarily over the network [24].

The often complicated and intimidating integration mechanisms and concepts, like discoverability of SOA, were partly responsible for the not too widespread adaption of the architecture, especially for small and medium sized systems [24].

The in many cases unnecessary complexity is reduced by microservices which allow developers to focus on creating a service at a time that solves one problem well [24]. The microservice architecture is also opinionated about the size of each independent module, in contrast to SOA, which is intended to improve the maintainability and scalability.

Microservices take the concept of modularization to the next level. Loose coupling and high cohesion is desired. Every service should be completely independently developed and deployed [24].

Integration technology

While monoliths may also have modules, the coupling of them is often much higher compared to a microservice. Technically there are very low boundaries to violate the architecture and introduce in the short term very useful dependencies but also aversely affect coupling. Therefore a programmer may affect negatively the coupling in a monolith. For microservices this is harder and also much more visible. A monolith uses in memory communication and function calls across module boundaries. This is very fast and efficient. However a major downside is, that the modules are no longer independently deployable [28]. If a change only affects one module, all modules need to be deployed, integrated and tested. These disadvantages are overcome by a microservice architecture. Lightweight messages are passed between the components. Technically REST or RPC style APIs can be used or message queues. Of course this is less performant and efficient. Objects have to be serialized and deserialized in order to be encapsulated in a message. This implies not just the raw transmission overhead, but also some effort for the marshalling.

The communications between the services have to follow specifications or interfaces. These have to be defined explicitly, in order to avoid breaking neighbouring services upon changes. This explicit interface definition should be published and changes handled with care [28].

Fowler talks about the iconic *smart endpoints and dumb pipes*. This points out a major difference to classical SOAs. Following a SOA architecture commonly complex integration strategies are employed such as an ESB. Microservices do not do that. The communication channels do not contain any complex logic and are only responsible for reliable transmission of messages or requests [28]. One problem with a complex integration strategy is that it may become the bottleneck and limit scalability. Additionally, there is a certain vendor lock-in that can be circumvented when simpler mechanisms for communications are employed. It is common for microservice architectures to mix integration methods. REST calls and message queues can very well be combined and used where sensible [28].

„Chatty“ services are discouraged. This are services that communicate a lot with other services. When converting a monolith to a microservice a simple approach is, to simply replace every method call across the newly introduced service boundaries with a remote call. Typically, there are a lot of those calls as they are cheap in a monolith. However in the microservice world, they are rather expensive and should be limited as much as possible. In some instances merging multiple calls into one aggregate call can help with performance issues, in others the structure of the program has to be further adapted [28].

Large data transfers are discouraged as well. The data should be stored near its usage. A single service should store its own data and not need to access a database, shared among different services [28]. Different instances of a microservice may access the same database, but this is not ideal either as the shared database may become a bottleneck. Not having to deal with the disadvantages of decentralized persistence is a bonus of this approach. Especially for traditional relational databases the overhead of distributed

transactions is enormous and will be discussed in more detail in section 5.1.3 in the part about transactions.

A big advantage of not using a large shared database for the whole system, is that every service is allowed to govern its database schema and technology. If transactions and Atomicity, Consistency, Isolation, Durability (ACID) semantics are really important for one concern, then a classical relational database may be a good choice. However its rigidity can limit other use cases. Being able to independently perform database migrations is also a big advantage. These aspects are abstracted away from other services. Historically, other parts of the system may have even depended on implementation details of a foreign module and thus limited its ability to change implementation details. By using only network requests as integration strategy such architecture violations become impossible and the distinction between implementation detail and interface becomes explicit and can not be easily violated.

REST

REST is an architectural style on how to create web APIs. It uses the Hypertext Transfer Protocol (HTTP) verbs GET, POST, PUT, DELETE and PATCH to distinguish types of accesses and transmits state. Fielding coined the term REST in his thesis in 2000 [25]. Since then it has become ubiquitous as an architectural style for writing web APIs, even tough Fielding disagrees with some APIs claiming to be a REST API [26].

While Fielding set some rigorous requirements for an API in order to be considered restful, the wider community of practitioners uses the term more loosely. One important aspect to Fielding is that REST APIs must use hypertext and not out of band communication to communicate available operations and resources. According to Fielding an API should not use a generic media type like *application/json* and require the client to have further a-priori knowledge about the content of the data [26]. A possible alternative would be to use custom media types that convey the content of a request more precisely and include what kind of data is transmitted. This is far from common practice at the point of writing. API specifications including the structure of every JavaScript Object Notation (JSON) object transmitted are ubiquitous among APIs commonly referred to as REST APIs.

In practice it boils down to, GET being safe in the sense that no modifications are made by issuing a GET request, PUT being used for idempotent writes, POST for not idempotent ones and DELETE to delete data. Verbs should not be used in the Uniform Resource Locator (URL) as common in RPC-style APIs, since the state should be set to a new state, but not actions directly triggered on the server side. Fowler has described different levels of compliance to getting all the benefits and of course also disadvantages envisioned by the original proposal of REST. The last level is the usage of Hypertext As The Engine Of Application State (HATEOAS). For Fielding it is a precondition for a REST API, but in practice it is often not used [29]. Using HATEOAS leads to the fact that the client and server are more loosely coupled.

Size

Netflix, one of the early adopters of this architecture, referred to microservices as *fine grained-SOA*. This illustrates well that a major difference to classical SOA is the size of each independently run service [24].

The name *Micro*service itself suggests also that the size of a service is a critical aspect of this architecture. Services that become too large should be split up into multiple smaller services [24].

There is no clear definition of how large a microservice should be [33]. The vague term small does not satisfy the desire for a rigorous definition. A common criteria is that a team should be able to develop one or multiple services. If more than one team is required to create or maintain a service, it is too large [24]. A microservice should also have a single responsibility. Again a responsibility can be defined very narrowly, but also quite broadly. If there are different needs for scalability or different deployment cycles, there should be separate services [33].

There is also a real danger in choosing microservices too fine granular. The results are „chatty“ microservices. These communicate way too much with each other as most relevant business actions span across multiple services and there are various calls made between them. The obvious problem is performance. Additionally, the suboptimally tight coupling also complicates development. It is more complex to develop features that span between several services than it is to remain within a single service. The number of deployments is higher and has to be coordinated for example. As a general rule of thumb it is desirable that most user actions can be dealt with by involving as few services as possible, ideally just one [9, 55].

A range of desired numbers of lines of code is given by Mazlami, Cito, and Leitner. They consider a service with just 100 lines of code very small and with 10000 lines of code very large for a microservice [51].

Conway's Law states that every organization is bound to create systems that follow the communication structures of the organization [24]. This gives an idea how large teams should be and how large a system can be that can be managed by one such team.

Since the name microservices implies that a service should be really small and loosely coupled there is some backlash in the practitioners community. Some claim that true loose coupling can only be achieved by using asynchronous messages and message queues because one service must not know the addresses of the services it depends on. Others find that extreme. The term miniservice is created to explicitly refer to less extreme variants where REST communication is the norm and services are not meant to be tiny, but a pragmatic approach is taken. Miniservices also tolerate some shared data between services which is not accepted in a pure microservice architecture. A major argument for business is that architectural purity is not equal to value [59].

Grouping of functionality

In order to minimize the communication overhead and maximize the commitment to the quality of a service, one team should be responsible for it. This is in stark contrast to the classical development methodology, which has different homogeneous teams that specialize on one layer of a software system. Typical examples for such layered experts are database administrators or user interface specialists. A services problem should be fixable within the team. Otherwise, unwise workarounds are common that deal with issues in the wrong place. Such quick fixes may be faster in the short term, compared to getting another team to act, but the long term consequences are far from ideal [28].

In order to have microservices work well, the teams should consist of several specialists or full stack developers and be able to deal with any layer. Teams are split along business contexts. Such an organization can work well for monoliths as well, but there the boundaries between modules are rarely as strictly enforced. A typical large monolith deals with too many concerns as to be clearly and fully understandable by individual developers [28].

Technical Heterogeneity

Monoliths are commonly only written using one software platform. No technology is optimal for every problem and compromises have to be made. A microservice architecture in contrast, allows different services to easily choose technology stacks independently of each other [28].

It is not wise to always do so and have an extremely heterogeneous technology usage without any standards. The downside of using too many different technologies is that one needs experts on every one and jumping between components can get harder for developers, who may not be familiar with all technologies. A balance needs to be found and the possibility to easily integrate other technologies is considered as a big advantage of a microservice architecture [28].

Shared libraries

Using shared libraries in microservices is a complicated matter. Normally, using libraries instead of repeating common code in every place is seen as a good idea. There exists the principle do not repeat yourself (DRY) which, as the name suggests, states that reoccurring code is a bad thing [55]. It is possible to create quickly a lot of code by copying and pasting with minimal changes. The problem with such an approach is the bad maintainability. If a small change needs to be made to a piece of code that is duplicated all across the program, the change will take quite a bit longer than if the code is nicely structured.

Since a microservice and its clients share at least the models of the objects that are transmitted, using shared libraries for those is an appealing idea at first. The benefits are clear. There is no duplicated code and all clients can start using all the features

of the service soon. Upon changes clients also see these, once the module definitions in the server are updated. The downside of this is that not all clients will have the same release cycle as the server. The coupling between them is much higher than when one decouples the client and server completely. Depending on the set up, changes of the model definitions can break the following builds of the dependent projects, either immediately or as soon as the version of the shared code in the client is upgraded. All of this is highly undesirable. Breaking changes should be avoided. Depending directly on the code of another service makes it much more likely that such breaking changes will occur [55].

Some companies that employ microservices extensively have rules that code may only be shared between microservices if it is an internal or external open source library. Such a library is generally helpful and not just in very specific cases for one service or depending on the domain. Open source also implies a shared code responsibility and when it is published, persons outside of an organization may look at the code. This outside observation can lead to higher standards of quality in the code [55].

Deployment

There is no clear answer if deployment becomes easier or harder when using microservices compared to a monolithic application. There are a lot more deployments if there are several services that change regularly. For a monolith there may have been only one deployable and the deployment process could have been easily and efficiently performed by hand. This is generally not efficient or feasible anymore if a microservice architecture is employed. If there is no up and running existing DevOps infrastructure, the necessary changes of course temporarily complicate deployments [8].

For a microservice architecture a lot of additional settings need to be configured correctly. This applies to all network related components that would not be necessary for a monolith. Load balancers and circuit breakers are prime examples. Depending on the kind of integration, a service may need to know the location of other services or at least the load balancer in front of them. Alternatively, if the integration uses only message queues, only the message broker needs to be known and this is less of a concern. Depending on how often these things change, configuration changes need to be made and this requires some degree of effort, depending on the automation of the infrastructure [8].

For a microservice the only allowed communication with other parts of the system, namely other services, is only allowed using the network. This makes integrating them with others easier. Also the number of dependencies required by a single microservice is probably lower than the sum of all dependencies that a monolith with the same functionalities requires. Generally, microservices and DevOps, continuous delivery or deployment go hand in hand and make each other easier [8].

Monitoring

Monitoring multiple services and servers or containers, especially in a distributed system, is harder than monitoring only a few servers. For this reason microservices face some challenges compared to monolithic applications, when it comes to monitoring. As for deployment the solution to these problems is automation. A continuous deployment approach should include solutions for automatically monitoring a distributed system. Most commonly a central log server is used and the log files are aggregated there. The open source *ELK-stack*, previously an acronym for *Elasticsearch, Logstash, and Kibana* but now the *Elastic Stack* is one widely used solution to aggregate and analyse logs [8].

Log files are not all that is required to successfully monitor in a microservice environment. One problem with just parsing the logs for errors is that if a service crashes really badly or does not even start, the log may not be written and no one may notice for some time, especially if the redundancy of a service masks the problem. Therefore it is a good practice to implement a system that will cause an alert if a service does not regularly state that its health is fine. Circuit breakers that prevent more requests from being routed to a service if it produces over a specified amount of bad responses, have a similar idea but do not solve the problem itself. If a circuit breaker is active in this capacity, an alert is also in order and the service instance behind may need to get replaced [8].

Since scalability is one major reason to use a microservice architecture it is important to know when to scale in which direction. Monitoring of the load of the services gives the answers to this questions.

Transactions

Many business applications work with transactions. This is a mechanism that among other things makes sure if an error occurs during the execution, the whole transaction will be rolled back and it will not be the case that some changes are persisted to the database while others are not. Microservice architectures are distributed systems. As such they inherit all the disadvantages of distributed systems. Distributed transactions for example are much harder than local transactions. One possibility to perform distributed transactions is to wait until all involved actors acknowledge that they can commit the transaction and then centrally give the command for all actors to really do so and otherwise roll the transaction back on all actors. However this is not very performant. It is more advisable to try and manage without distributed transactions that span between different microservices. The downside is that it may be the case that the first service successfully saves some data, but another service fails to act upon them and the initial data is not changed back [44].

Testing

For testing some aspects get easier, but some also get harder by the usage of a microservice architecture. On the unit-test layer not much is changed. When in a monolith, a service or repository method may get mocked. In the microservice a network call can be mocked

just as well. True end-to-end tests are more complex to set up as all services need to be deployed and integrated. This involves more effort than for a monolith. Transactions, which are often rolled back after every test case and allow a resource efficient way to isolate tests from each other in a monolithic integration test, are mostly not available at this level in a microservice architecture [55].

One writes end-to-end tests in order to make sure that not just every part on its own but the system as a whole works as intended. Unit tests alone can not provide that certainty.

Microservices have to define interfaces that other services can use. For tests the interactions expected from other services, according to the interfaces of those, can be mocked allowing to test a service in isolation without all the other services [8]. Consumer-driven contracts define the expectations of a consumer of a service. This is done in the form of a test suite that is run against a version of the service. If the tests are thorough enough that all interactions that a client wants to make with the service are covered and the service passes these tests one, can be reasonably sure that the service fulfills the expectations of the consumers and thus does not break other services hat consume it. The tests are usually run on a continuous integration server against a single service in isolation and thus prevent services from breaking services depending on them. The scope of this approach is similar to an end-to-end test, but since only a single service is tested the speed is much better and the test is less dependent on other services. Depending on other services in a test is complex. It is not clear which version of these services should be used. The latest version in production is not ideal when two new services should be released almost simultaneously [55].

Security

For a monolith the boundaries, which requests and data can be trusted, are often clearer than in a microservice architecture. Generally, all requests coming from users can not be trusted and must be validated. Some services in a microservice architecture may be called directly by a user or an untrusted third party, but also from other services under the control of the same company. In the latter case it is not clear what credentials to use. A service could have its own credentials and then use them for requests that are made on behalf of a user. Alternatively, the users credentials could be used for further requests. The latter approach has the benefit that the user only has the own rights on the second server and not all permissions that the first service has. This can only be done for requests that are sent on behalf of a user and not for example as background batch job. It is a common but insecure practice for microservices to trust each other completely. If one service is compromised under these circumstances, the attacker is put in a too powerful position and can exploit further services [64].

In some circumstances checking the authentication state is not resource efficient. In a microservice architecture where there are lots of small requests, always checking if a token is valid in a central database or a service dedicated for that purpose, is not efficient at all. JSON Web Token (JWT) is one technical solution to this problem. Such a token can

contain arbitrary data that is readable to everyone, but is signed so a user can not change critical information, like the username or roles associated with the token. A service then only needs to check if the signature of the token is correct and if it was issued by a party in possession of a secret, which is much faster than a database lookup or service call [55].

5.2 Software maintenance

According to Lientz, Swanson, and Tompkins there are three very broad types of maintenance operations that can be performed. There is corrective maintenance, that fixes existing faults, adaptive maintenance, which responds to external changes, and perfective maintenance which aims to improve certain aspects of the system that are already working. Performance improvements or increasing the maintainability are perfective changes [49]. Abbas et al. also cite a fourth type called preventive maintenance, which deals with averting future problems [1].

A maintenance project can be divided in four different and consecutive stages. The first one, during which the first issues with a system are discovered, is the introduction stage. It is followed by the growth stage in which more and deeper technical problems are discovered and hopefully mitigated. The third stage is the maturity stage. At this time major enhancements are performed. In the final decline stage the system is just patched and no major changes are performed any more, as its life cycle comes to an end and it is eventually replaced or not needed any more [1].

Modern systems often interact with many other systems to provide the desired results. Still, goals and desires of stakeholders change and the systems have to adapt. Such complex and interconnected networks of systems are expected to be always available, allowing no planned downtime. In these cases it is hard to completely replace such a central and integrated system quickly and cheaply. Venters et al. name accidental software complexity. This is complexity that is not planned from the beginning of a system, but over time develops as the system becomes more involved in the larger processes. It is particularly caused by accumulated technical debt, too high coupling and thus suboptimal cohesion and missing documentation for design decisions. Many of these issues can be attributed to a lack of time spend on proper development processes. The architecture may erode over time and by the gradual decline in maintainability no change alone is responsible, but it is the sum of many hasty changes. Good architectures allow enough flexibility to perform the needed changes in the future without the need to sacrifice the long term maintainability, by introducing workarounds that negatively affect the whole system in the long run [71].

Sustainability encompasses at least maintainability and extensibility. Even if the main focus of a newly developed system or its architecture is not sustainability, in the interest of long term quality and cost efficiency, Venters et al. recommend explicitly considering sustainability during designing and implementing a software and its architecture [71].

Today many computation systems are well integrated in the daily lives of people. Changes

have to be smooth in order to not upset them. There is also the conflicting desire to evolve quickly and react to change requests within very short time frames. This makes maintenance a challenging topic [71]. Today it is common that software is reconfigured several times or may be updated at any time without much control from the vendor. This is the case on smartphones where apps are commonly updated automatically without much control, but may also not be updated at all. Still the backend systems should work well with every device and software version [71].

One common issue with maintenance releases is that they decrease the performance of the overall system as it was not designed to operate in the new way and inefficient workarounds are employed. A good and flexible architecture makes this case a rare occurrence. How to reliably design systems in this way is still a challenge for software engineers [71].

Venters et al. therefore stress the importance of sustainability when designing new software systems and include maintainability in this consideration. Sustainability in this sense means the ability of a system to endure or its longevity. The given definition for sustainability is that a system is sustainable if over its entire life-cycle it is possible to cost-efficiently maintain and evolve it. Present needs should be met without sacrificing future needs. Software architecture is identified as the main reason for good or bad sustainability of a software system, being even more important than the skill of the implementing software developers [71].

While sustainability and maintainability are extremely important issues there are no widely accepted and proven measurements to quantify these issues [71]. Venters et al. state that this field of research is still active [71].

Venters et al. distinguish between first to third order effects when using a system. First order effects occur when the system is used as it was intended during the initial manufacturing. Second order effects appear when users start to use the system in slightly different ways as intended and finally third order effects include long term usage of a product and all its implied outcomes [71].

During maintenance architectural drift and erosion may change the effective architecture a system has and can affect the maintainability and sustainability. This is mostly caused by unsystematic changes that do not follow the intended architecture. This again may be caused by changes from different programmers, that were not involved in the design decisions of a system and the rationales behind them.

Aspects that affect the maintainability positively include using widely used technologies, that provide regular updates and have a community behind the architecture [71].

Sustainability is inversely proportional to the potential loss in quality and thus can be measured in this manner. The classical measurements for technical debt help to asses the parts of a software system with suboptimal quality. There are tools available that allow to collect these metrics automatically and plan countermeasures. Such metrics can be partitioned in two categories. The first type is code and architecture metrics.

These metrics calculate statistics on the code and employ heuristics to find potential bugs and hard to understand parts of the system. On a higher level this also includes measurements of the cohesion between modules and their interdependence. Architecture knowledge metrics are the second type. They aim to estimate the sustainability of design decisions itself and not its manifestation in the code [71]. Combining multiple metrics often leads to improved results with more reliable results [71].

5.2.1 Costs

Maintaining existing systems that provide vital functions for businesses is crucial for continuing smooth operations. This was already known in 1978. Back then Lientz, Swanson, and Tompkins reported about various estimates what percentage of the total costs were consumed by this part of the life cycle. The estimations range between 40% up to 70-80% of the total resources required with the majority being roughly around 50-60%. The research in the field was not mature according to Lientz, Swanson, and Tompkins in 1978 [49].

More recently, Abbas et al. also report of similar efforts required to maintain systems. Again the estimation is about half of the total effort [1]. This leads to the conclusion that over time the cost efficiency of performing maintenance has not dramatically changed.

Lientz, Swanson, and Tompkins report that over time the budget allotted in many companies for computer systems increases but a large part of it is swallowed by the rising costs of maintaining old systems. This is even often the case when management wants to create new systems [49]. Management often considers maintenance programming only to be a minor problem and not proportional to the costs caused by it even though it is more important than developing new systems. Perfective maintenance plays by far the biggest role being responsible for about $\frac{2}{3}$ of the total costs [49].

5.2.2 Agile and maintenance

In more traditional software development approaches there were phases dedicated to maintenance. In projects developed using agile approaches this is not the case anymore. Depending on the duration of a project there may be the need to perform maintenance tasks during the creation of a program or after a project is finished in a following maintenance project. Many maintenance tasks can not be directly assigned to a user story as they deal with technical improvements that are not directly visible to a user. Depending on the type of maintenance operation and the type of agile development methodology, following the process by the letter and creating the best value for the customer may not be possible at the same time. Fixing an urgent issue immediately even though it is not part of the scrum sprint or creating tasks that do not belong to a user story is quite sensible. Abbas et al. provide a methodology that is based on scrum and considers maintenance as a first class priority. One issue is missed sprint goals when urgent matters were handled first and scheduled tasks did not get completed during the

sprint. When using it, the original sprint gets interrupted and continues after the urgent matter is resolved [1].

5.2.3 Types

As stated previously, there are four different general types of maintenance. These will be discussed in more detail in this section.

Adaptive maintenance

If maintenance is performed in order to react to changed data formats or data in general or to accommodate environmental changes, it is called adaptive maintenance. Depending on the technology, portability may reduce the amount of adaptive maintenance that is necessary to keep a program running for a long time and on different computation platforms [66].

Perfective maintenance

Mostly, there is no directly pressing issue to warrant perfective maintenance. This type of maintenance deals with improving performance and maintainability itself. Therefore, it does make a lot of sense to perform from both, a technological and a long term business point of view, even if there is no urgent matter that forces immediate changes [66].

Corrective maintenance

Even when a software system has been running without known issues for a long time, every now and then new issues are discovered that may only be triggered under very exotic circumstances. These issues still have to be resolved. When the project is already in the maintenance stage, the maintenance action to resolve these issues is called corrective maintenance. Without the discovery of an issue this actions would not be performed, but depending on the importance of the issues the priority may be very high. Often these type of issues are caused by changes to the environment of the software system. This includes software updates of dependencies of the system, which may be necessary for security reasons and can not be delayed [66].

Preventive maintenance

Similarly to corrective maintenance, preventive maintenance does not deal with an issues just as it arises, but further enables the system to work as planned and designed. The difference between the two named types of maintenance is that preventive maintenance is performed before the issue is noted in production and is done in order to avert it from surfacing. An example would be to fix a bug that gets caused at a certain date before this date [1].

5.2.4 Code smells

Poorly written software of bad quality is harder to maintain. There are a number of automatically detectable code smells which indicate that some part of the code base is far from optimal and should be improved. The basic assumption is that parts of the software with many code smells are hard to maintain and are more likely to contain bugs. A code smell would be for example overly complex control flow as is the case with many and often nested control statements. Such code segments are hard to understand for humans and bugs are more likely to occur. Maintaining a long segment of nested if-else statements is also hard and inefficient [62].

According to Kent Beck, code smells help to identify quality issues that lead to bad maintainability. These are code structures that suggest this part of the code should be refactored. There are several more precise definitions of what a code smell is, but the general consensus is captured by this informal description [62].

The common conceptions are that a code smell indicates a problem, represents a poor solution to a problem, violates best practices, impacts quality or is a recurring problem [62].

Several well known code smells are listed by Sharma and Spinellis including: [62]

- **God class**: A class that contains most of the logic of the program and other classes hold merely data for this class. This smell indicates poor modularization.
- **Shotgun surgery**: In order to change one business requirement a lot of different parts of the code have to be changed, instead of just the part where the feature is implemented.
- **Long method**: It is similar to *God class*. One method contains most of the logic and everything happens around this method. It is hard to maintain and understand.
- **Functional decomposition**: If procedural code is written in an object-oriented language. This is often the case with developers coming from a procedural background.
- **Spaghetti code**: Unmaintainable code without much structure. Object-oriented features are not used properly and it is not easy to follow the control flow.
- **Duplicate code**: The same code is repeated in different places. It is often the result of copy-pasting code. Fixes in one place may never be applied to the others.
- **Speculative generality**: There are no real requirements that merit an abstraction, but it is implemented anyway in case one may need it in the future. This never happens though. It makes understanding the code base harder.
- **Swiss army knife**: The designer of a class attempts to foresee all possible use cases and ends up with a class with a too complex interface.

Some authors do not distinguish between code smells and antipatterns. Others see antipatterns on a design level and code smells on an implementation level. One important distinction named by Sharma and Spinellis is that while antipatterns are bad solutions to reoccurring problems, code smells are merely indicators that the code may be bad. Antipatterns may lead to code smells [62].

Sharma and Spinellis also list reasons for code smells. Among the most common reasons are poor technical skills and a lack of awareness what makes code good code. Frequently changing requirements and constraints on the language or platform used also account for many smells. Frequent change requests make it hard to properly plan the code structure. Given only very little time for changes, shortcuts that are suboptimal in the long run are often taken. There are also often knowledge gaps between higher level designers of a system and the programmers implementing details. This is especially true for large systems with a bureaucratic management style. If management favours fast results and quantity of features over their quality, there is a tremendous amount of pressure on programmers not to refactor code, but start with the next feature immediately. In the long run code smells often amass in these organizations code. Understaffing and tight deadlines also come to mind. A team culture that does not focus on quality but on short term throughput has the same effects.

There are many tools written for different languages to detect code smells automatically and direct a programmers attention to them. They may be based on metrics, rules or heuristics, on the history of the code or project, employ machine learning or optimization techniques [62]. Issues like pieces of code with high cycliomatic complexity that are hard to follow can be found using metrics. Metric based approaches are relatively easy to implement, but do not find many issues. Additionally, appropriate thresholds have to be defined. An example is how high the allowed cycliomatic complexity of a method should be. When setting it too high, many cases of bad code will be missed. Setting it too low may lead to refactorings and method extractions that add no value. Heuristics are more suited to finding common mistakes and are often used in combination with metric base approaches. Unfortunately, only a given list of well known and defined problems can be found, but that with high precision and efficiency [62].

History based approaches for detecting code smells are relatively rarely employed in practice. The main disadvantage is that they require rich historic data that is often not available and only aspects of the code which are affected by changes are found. Machine learning approaches are relatively new and require large data sets for training and are currently not widely used [62].

Current tools for detecting code smells often result in a large number of false positives. These are indicated code smells that are not actually bad code. More precise domain knowledge is often required to fine tune the detection process. Metric based approaches have to be fine tuned to reach optimal levels of detection, while reducing the number of false positives for a given environment or project. Some code pieces identified as smells can also be very appropriate in specific circumstances. Developers should not blindly follow all suggestions of a code smell detection tool [62].

Not only production code, but also test code can have smells. These smells do not only hint at suboptimal test code, but often also at poor design of the tested production code and low test thoroughness [63].

While many developers believe the production code is much more important than the test code and only the production code needs to be properly maintained and refactored, Spadini et al. have found high correlations between test smells and bad production code. The code covered by test code with test smells is found to be more likely to be high maintenance code and code with more bugs than usual. Among the smells *Indirect testing*, *Eager test* and *Assertion Roulette* are the ones most strongly correlated with bad production code. *Indirect testing* describes the practice of not testing a unit directly but indirectly by calling other units which in turn call the unit under test. A test is an *eager test* if it tests more methods at once and *assertion roulette* describes the act of asserting multiple things at the same time which makes knowing which precise part of the assertion failed much harder [63].

5.2.5 Modernization

Legacy systems are of high importance to the businesses using them yet are obsolete in one sense or another. Maintaining such systems is often hard and laborious. Implementing new requirements in these systems may not be well suited to the original architecture and thus the code quality decreases and further maintenance becomes even harder. When the original authors retire and other programmers, who are not familiar with the original design decisions, take over the situation gets worse [41].

Maintaining legacy systems is not cost efficient. Developers being familiar with the used technologies become more scarce and the raw speed of implementing changes decreases over time. Jansen et al. state that the documentation of most legacy systems is not optimal and not up to date. This leads to lost knowledge between the new maintainers, who are not part of the original team, and the ones that originally developed the system. Vendors that supply patches and hardware support may go out of business, increase the costs for support or simply stop to provide support altogether. Since they are critical to business success and often have enormous code bases, legacy systems can not be replaced easily. Their low flexibility and poor maintainability make them often incompatible with the future goals of the companies. While their speed and stability are rarely problematic, flexibility often is. This is because keeping up with changing requirements and shifting environments are becoming harder problems over time. Modernizations become more urgent [41].

One common problem when modernizing legacy systems is that the requirements are not clearly known anymore. Since the code is hard to understand and not well structured, reverse engineering the business logic is hard, but often the only way to reliably get all the system functionalities. One migration strategy is then to write thorough test cases for the legacy system and only afterwards create a new system or to migrate to a newer

architecture and to make sure that the test cases do not break during the modernization effort [41].

Staff may not want to cooperate with a modernization project. This resistance is often rooted in fear of not being needed any more or having to retrain and losing job security. If these employees hold crucial information needed for the modernization this is a serious problem and can jeopardise the whole endeavour [41].

Software needs to evolve to keep up with the needs and desires of its users and businesses developing the software. There are different evolution strategies. Getting it right is of utter economic importance. As stated previously maintenance absorbs significant amounts of the budget of most software projects. Common is the claim that more than half of the costs are caused by maintaining a system [45]. Koskinen et al. claim that software needs to be continuously adapted or its usefulness decreases. Just throwing away a software after some time is not economically viable as significant investments were put into it and the more integrated it is the harder a big bang migration away from it becomes. Therefore most companies choose to modernize their systems at least for some time [45].

Modernizations are changes of larger size, that change an existing system either technically or functionally in the business context. While these modernizations are economically very important, they are technically and socially challenging. Users may resist change and management may object to spend money on a running system [45].

Koskinen et al. questioned several Finnish software companies for their experiences. The top level decisions often dealt with are about whether to move to a competitors product or to reduce the usage of an existing product, on the one hand and to invest in the modernization or complete rewrite of the existing product on the other hand [45].

There are often no available software choices of competitors a company can easily switch to. Significant investments in integrating the old solution as well was deliberate vendor lock-in are reasons for this [45].

A complete rewrite is quite expensive and there is a lot of risk involved. Modernizing existing software yields a usable and improved system much more quickly and incrementally. For this reasons complete rewrites of software are often seen as the last resort option when the maintainability of the original system has degraded so far that it is no longer economically sensible to further invest into or the technical difficulties make porting the system to a new environment unfeasible [45].

The three most important concerns when modernizing a software system include the system usability, ending of technological support and changes in business processes according to Koskinen et al. [45]. The latter two force organizations to change the software while the first one partially stems from pressure from users to improve the working efficiency and quality in general [45].

Legal changes often force organizations to adapt software to comply with the new regulations and are considered very important by most participants of the study performed

by Koskinen et al. [45].

One approach for modernization of existing systems is presented by Cánovas Izquierdo and Molina. As a first step a model of the software is created from the source code. These models can later be used for modernizing the existing software suite [16]. Unfortunately there is no full-fledged solution ready to be used by practitioners.

Automation

Since the act of modernizing a piece of software is very labour intensive, there arises desire to make this process more efficient. Automation can often provide more efficient result or simply make the work easier to perform. Unfortunately, there are not a lot of automation tools available to help maintainers performing their work [34].

Grambow, Oberhauser, and Reichert state that the size of average computer applications doubles every four-five years and that old legacy systems become harder to maintain. Humans can no longer comprehend all details of huge software system efficiently and fully. Therefore they search for automated alternatives and improvements to the current best practice, manual maintenance and modernization of software. Maintaining exactly the same functionality even when changing the programming language is very challenging, especially for humans using an unsystematic ad-hoc process for modernization [34].

One obstacle identified by Grambow, Oberhauser, and Reichert is that on the one hand a project is planned rather abstractly and that on the other hand different people who implement the concrete code and perform changes use different tools and methodologies. One early step in modernization is recovering the knowledge of the system architecture and its violations. Often the original authors are no longer available or the code is too old for them to remember the details. In these cases the architecture has to be reverse engineered in order to efficiently transform the system to the newly desired form or to create a newer replacement [34]. Every individual software engineer should be aware of the bigger picture in order not to make locally good looking decisions that later lead to problems when performing operations not thought of by said engineer [34].

Grambow, Oberhauser, and Reichert state that for example code generation tools can help with modernizing software, but focus on narrowing the gap between abstract analysts and implementers. Development methodologies like Scrum or the V-Model XT have not as much impact on the day to day actions performed by the average programmes as desired by them. They state that these processes often only exist on paper, but are not really followed in practice and are often considered as a burden and not as help by the users. The reason they give is that the process is too far detached from the actions performed by these actors [34].

There are intrinsic workflows and extrinsic ones. The intrinsic workflows are part of the process while the extrinsic ones are unforeseen and therefore not part of the process. For intrinsic workflows at least the artefacts created by perhaps different actors should link nicely if there are semantic or programmatic ties. Quality assurance while often just

an afterthought should also be well integrated in the process framework and applied during the whole live cycle. Tasks such as requirements engineering during the knowledge acquisition phase are also crucial to the success of the whole modernization endeavour and have to be integrated and well linked to the artefacts of other actors [34].

Grambow, Oberhauser, and Reichert stress that software engineering as a whole is a highly collaborative process. One key improvement is to automate much of the friction between the different stakeholders. Important parts are automatic notifications when actions concerning a person are performed. Not distracting the participants is also considered very important as people in the „flow“ are substantially more efficient and entering this state of mind is time consuming after every interruption. Additionally, the system envisioned by Grambow, Oberhauser, and Reichert is expected to predict the outcome of an execution of a workflow and to choose appropriate people to perform these. Follow up activities should also be automatically created and maintained [34].

5.2.6 Modularization

Modularization is a powerful tool to deal with complexity in software. When systems are not trivial to comprehend, different aspects are separated in different modules. When done well this allows developers to focus on one module and more easily comprehend it. Depending on the programming language and the overall architecture of a program there can be different forms of modules and interaction patterns between them. One common interpretation in the Java ecosystem, implicitly followed by Candela et al., is of Java packages as modules [15]. In this case communication and integration is mostly done in the same process and thus in memory and the cost and effort for accessing data or methods from another module are very small and not higher than if it was in the same module. Java's package private visibility is one of the few aspects that can make a real difference on the technical level.

A more thorough separation of modules is performed in a microservice architecture. A module here is considered as a service that is runnable in itself and thus also runs in its own process. Different modules or more commonly services communicate with each other only over the network and do not share any other communication channel such as in memory communication. This implies a performance degradation of course. Still the clear separations of the modules or services and their independent deployability and scalability make this architecture quite popular, despite of these low level performance penalties.

Generally speaking, modularization can be described as a graph partitioning problem as code that is put in the same module can be represented as vertices put in the same partition of a graph. Graph partitioning is NP-hard and thus also finding the best modularization is considered intractable to solve efficiently [15].

When a system is developed over a period of time, developers are often faced with implementing new features under time constraints. They may also not know the original design decisions. Then they can and often do make decisions that violate the underlying

architecture. Prime examples are suboptimal decisions that allow changes to be made faster in the short term, but incur technical debt and slow down the development later. Candela et al. discuss this erosion of the original design. They conclude that such short-sighted decisions often increase the coupling between modules and reduce the cohesiveness in them. In the long term these are serious issues that make the system harder to understand and maintain [15].

Manually performing a remodularization that moves code to more appropriate places is very time consuming. One of the reasons is that the suboptimal places in the code have to be found first. Several tools exist that help finding better suited modularizations using clustering algorithms or heuristics. Search heuristics such as Hill Climbing, Simulated Annealing and genetic algorithms are commonly employed [15].

Cohesion describes the amount to which the actions or data provided by one module or unit belong together. In a god class that does all kinds of different, unrelated things cohesion is low. High cohesion is desired [15].

Coupling refers to the relation between different units / modules / classes. High coupling means that two different parts are highly related and this is considered as poor design as low coupling is desired [15].

The goals of high cohesion and low coupling are partially contradictory. Putting all the code in one module would provide optimally low coupling, but also terribly low cohesion as completely unrelated functionalities are joined in one module. The opposite approach using a vast number of modules, each consisting of almost no code, has the inverse effect [15].

While Candela et al. found that generally the cohesion and coupling of examined projects are far from optimal according to theses tools and metrics, the majority of the projects developers (~83%) claim that cohesion and coupling are by far not the only decision criteria when choosing a modularization [15].

Since naming of variables, methods and other code parts contains valuable meta information about the code, some algorithms use this information to cluster similarly named elements more closely together. Similar things can be said about the version history. Code that has previously often been changed together is thus more likely clustered together by some algorithms [15].

Candela et al. analysed 100 different popular open source projects and compared the cohesion and coupling of the program at the point of the analysis and using a genetic algorithm calculated better solutions according to these metrics. In most cases the differences between the original program and the calculated improvements are high. From this they conclude that the effort for a remodularization is also high and the originals are far from optimal. However automatically refactoring the code is ruled out, as the mental model that programmers have is very important and would become obsolete if automated refactorings were performed. The majority of open source programs inspected favour cohesion, meaning their cohesion is higher than the cohesion of the calculated

optimal alternative. This sacrifices coupling however. About a third of the projects had both worse coupling and cohesion as the calculated solution. Candela et al. admit that conscious design decisions such as putting different layers of a multi-tiered application in different modules is often considered bad by their approach. For example, a class of the GUI layer may have very high coupling with a class in the service layer. It may still make a lot of sense to have them in different modules. Generally, such layers increase maintainability and are crucial to the overall architecture. Therefore, they plan future improvements to their approach [15].

The developed approach uses a graph representation of the code of the program and uses clustering algorithms on this graph, in order to find a suitable partition to serve as basis for the boundaries of the newly created microservice architecture. The partitions represent bounded contexts in this sense.

Surveyed open source developers mostly do not use external tools that suggest better modularizations and mostly rely on their Integrated development environments (IDEs) to perform refactorings including remodularizations. Developers are also often not comfortable with tools changing their code automatically [15].

The most important criteria when choosing the modularization for most developers are the layers such as the service layer and usages of classes such as Data Transfer Object (DTO). Considering cohesion is only secondary to most developers. Team boundaries are also commonly used as module boundaries [15].

Developers often consider it more important to group code together that is expected to change at the same time and not to spend too much time on considering the number of calls between modules or their textual similarity [15].

It is a common occurrence that existing systems are not optimally modularized. Especially when the overhead of communication between modules is high, as it is the case in a microservice architecture, this quickly becomes problematic. The modularization should therefore be mostly right in the first place as later changes are more difficult. Mazlami, Cito, and Leitner discuss formal approaches to construct a very good and future proof modularization into a microservice architecture starting from a monolithic architecture. For this architecture it is of particular importance that low coupling between services or modules is maintained as performance suffers severely otherwise [51].

5.3 Scalability

Scalability is one major reason many companies are switching over to microservices or developing green grass projects using this architecture. It is partitioned in two main categories, horizontal and vertical scalability.

Vertical scalability, the less flexible one, describes that the system is able to handle more requests or increase throughput if the system is run on more powerful machines. Adding more Random Access Memory (RAM), using a faster Central Processing Unit

(CPU) or storage space of course increases the power of a node and most likely more data can be processed. There is still a limit. Using more and more powerful hardware quickly becomes prohibitively expensive and eventually not feasible anymore [47].

Horizontal scalability in contrast is about adding more nodes to the system and not making a node more powerful. The concept is often used in the context of cloud computing. It implies the requirement for using a distributed system architecture with all its advantages and also disadvantages. More nodes allow for better resilience, but fully ACID compliant transactions with multiple writing nodes become inefficient and slow very quickly for example [47].

Scalability is related to elasticity. Commonly elasticity in this context is understood as how dynamically and quickly the system is able to adapt to changed demands, that is to scale up or down. Alternatively elasticity can be defined in the ability to prevent states of over- or underprovision. These are states where the load does not match the available resources. An overprovisioned state is a state where there are more resources available than required. In a cloud scenario this is not cost efficient, and one should scale down. The opposite is the case for underprovisioning. In this case the Service Level Agreements (SLAs) are not met any more or the system is not able to handle the load any more [47].

Microservices are mostly named in combination with horizontal scalability, the possibility to easily add more distinct machines to a network in order to satisfy growing needs or to scale down again when the load decreases in order to save resources and ultimately costs which works well in a cloud environment. One big downside of cloud deployments is however that the costs can be hard to estimate. Once the resource requirements for the system grow this can become a serious issue for not yet very profitable businesses [3, 76].

The big benefit that smaller services have is that they can be scaled more precisely. While a monolith can only be scaled in discrete steps for the whole application every single microservice can be scaled independently from each other. There is also no big loss of efficiency as all communications with other services are done via network anyway [3]. Smaller services also tend to be easier to set up in a fresh environment as the number of dependencies is generally lower and thus not as many things need to be configured.

Scalability is an extremely important topic for microservices, as it is one of the primary reasons for most migrations to it or greenfield developments using this architecture. Monoliths can not be scaled as well as microservices, as they are mostly larger and only discrete instances of the monolith can be spawned or shut down.Large instances also imply slower start-up times and more memory usage even if not all parts are under stress. Monoliths often use one big relational database. This also limits horizontal scalability. Microservices in contrast should have independent data stores and may even use completely different persistence technologies and thus can have any kind of scalability of the persistence layer [38]. Monoliths can use any kind of persistence technology as well but it is not as common there to use non-relational databases or the optimal database for each use case instead of one generic one.

Even though scalability is such a central term, there is no consensus on its precise

definition [47]. Overall it is expected that no matter the load thrown at a system, the system can handle it if given enough resources and instances, if it is well scalable. This scaling to the right size is the key part. For monoliths this scaling can only be performed in discrete steps for the whole application. Often it is desirable to scale only the parts under stress.

In cloud environments, especially, it is desirable to measure the achieved scalability and to do so, a specification of this term is required. Lehrig, Eikerling, and Becker perform a systematic literature review in order to find an existing consensus on this term. They cite the following definition: *Scalability is the ability of a cloud layer to increase its capacity by expanding its quantity of consumed lower-layer services* [47]. Another way to express this idea commonly found in the literature is that scalability describes the ability to handle increasing or decreasing workload by allocating more or less resources [47].

Brataas et al. define complex scalability metric functions and evaluate them on projects run at Amazon Web Services (AWS) and on OpenStack [14].

The load or throughput a system is able to handle is often defined in a SLA between a service provider and its consumers [47].

5.4 Bounded Contexts

Developing large software systems requires the participation of multiple stakeholders. Programmers write the code, but they have to know what to write. Domain experts generally have a far greater understanding of the problem domain and need to share their knowledge with the developers. Following DDD one creates software based on a model of the domain.

When the model becomes large it is sensible to split it into smaller parts, the so called *Bounded Contexts*.

In large organizations different sets of people more closely in touch with each other will use slightly different vocabularies or have different priorities. Instead of enforcing one global terminology and its semantic, DDD allows for differences among contexts.

Some entities are shared between Bounded Contexts and others are unique to only one. For example a warehouse will typically care about different properties of a product than the accounting department.

The terminology only has to be totally consistent within a bounded context.

Boundaries between bounded contexts are often driven by different used terminologies and the frequency of interactions between responsible people. Departments therefore often coincide with bounded contexts [27, 55].

CHAPTER 6

System at hand

6.1 Project Introduction

The system at hand is a car dealership management software handling both the sales process of cars and the after sales and maintenance tasks commonly performed by car retailers. Interoperability with vast amount of third party systems is provided both to comply with local regulations and to exchange information with other already existing systems in the car retail ecosystem. In order to satisfy the needs of diverse customers from various countries, the system is very configurable including the ability to enable and disable various components and integrations.

The system supports the whole workflow of a car retailer. As an example, a customer calls to schedule a repair on the newly bought car. The system knows which type of car this is and potentially reports the maintenance to the car manufacturers API. Scheduling appointments is also supported by the system. A representative is scheduled to receive the car before the maintenance operation and to hand it over afterwards. In between a mechanic is scheduled to perform the needed adjustment. The estimated duration for the repair as well as availabilities of mechanics, their legally required breaks and shifts and qualifications, are considered as well as the available space in the workshop at any given time. Parts required for the adjustment are also managed and orders are placed if necessary. In the end a receipt is created that fulfils all accounting and legal requirements.

The created software is meant to be used by trained employees of car dealerships or workshops. This includes salespeople, mechanics and administrative staff. Customers never directly interact with it, but through third party software integration it is possible that data is shared with customer facing applications and they interact thus indirectly with the system at hand.

The company owning and developing the product already has a legacy product with a very similar feature set. The old product is still actively maintained and used by many

car retailers in production. The underlying software technologies are becoming obsolete and the maintainability is very poor. Therefore, both management and customers push towards the new version.

The new project is already in development for about ten years and used in production in pilot deployments but not widespread.

Technology wise Java and the Spring Framework are used. Apache Tapestry is used as the frontend framework.

At the point of writing deadlines for pilot installations in several countries are dated less than a year in the future. This requires the development of country specific features. Some countries have several time zones, which adds complexity, as the time zone can not be configured in one place and be valid for all parts of the system. Other countries use more than one language and features, like creating receipts in a language different from the language used by the operator and those have to be implemented.

6.2 Project and Product History

The product has a well received and profitable predecessor. However, this predecessor is becoming technically obsolete however. It is mostly written in C++ and its frontend is platform specific for Microsoft Windows as there are fat clients installed on every users device. The predecessor system suffers from very poor maintainability. These technical limitations led to the decision of creating a completely new product with similar features and target audience, but using modern platform independent technologies.

The development of the new software began about ten years ago. Back then the project name was different and has been changed once again. At the beginning Subversion (SVN) was used as a version control system, but in 2012 Git took over this role.

6.2.1 Project size

One major issue with the project is slow development time. This is to a large extend caused by the size of the biggest Git repository, containing the code for several executables with lots of shared dependencies and lots of dependencies on other services. Historically, there was one large SVN source code repository. Now there is a desire to reduce the size or use multiple repositories in order to improve the build granularity and development speed. Due to the monolithic nature of the application this is not an easy task. Building the largest project on a developer workstation takes about ten minutes at the point of writing and starting one service takes about another five minutes. CI-builds take between 45 minutes to an hour, depending on server load. These high latencies between making changes and being able to run them or having them tested in a reliable CI infrastructure makes developing much slower than desired.

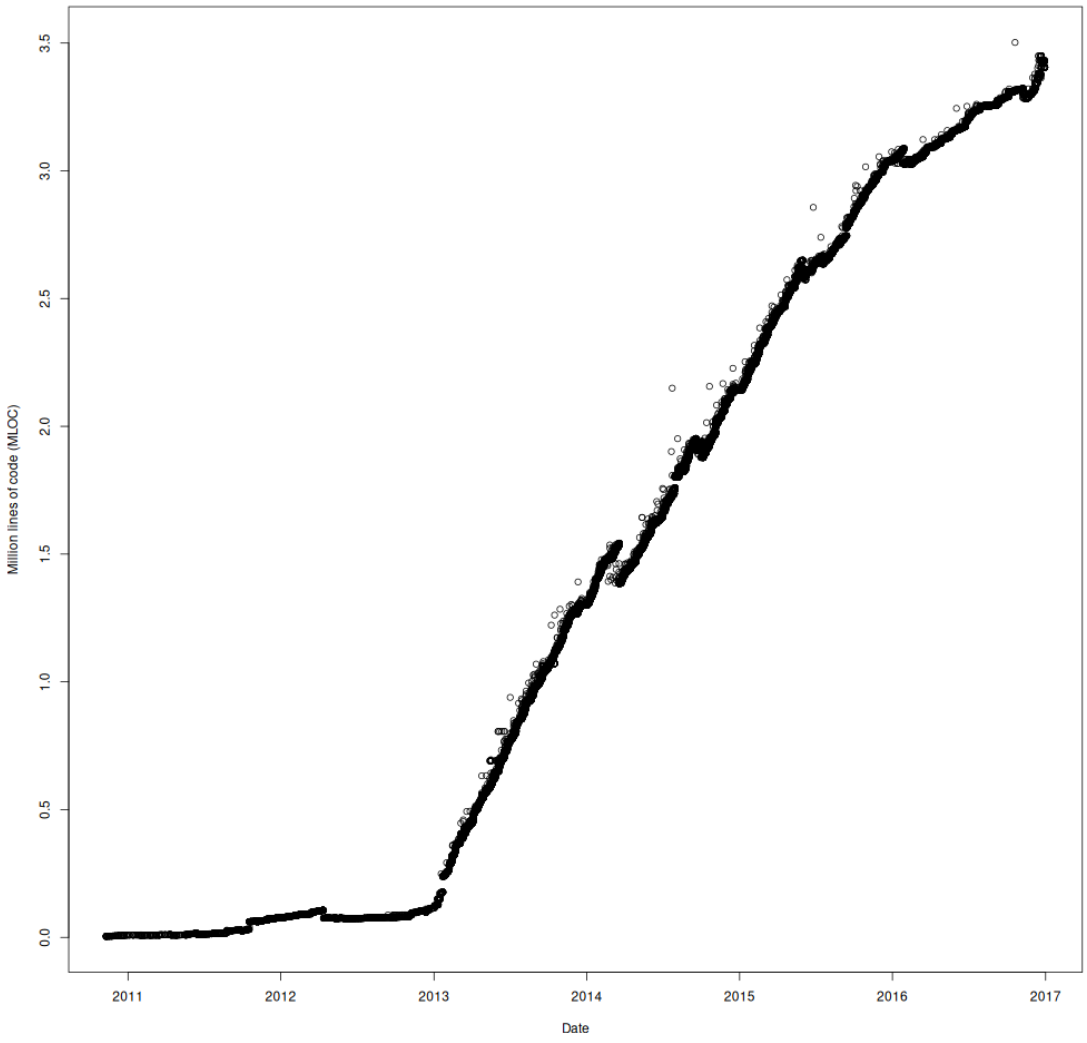

Figure 6.1: LOC history

Figure 6.1 shows the history of the lines of code at the given point in time of the largest code repository. In 2010 the project started and has reached between three and four million lines of code.

There are a few outliners on the left side of the majority. This is caused by commits where the associated date is significantly earlier than the submission date. The project uses Gerrit and every commit is applied on top of the current master branch. Commits that are submitted a bit later, than they are initially created, have this initial creation date annotated and are plotted in this figure at this initial date. However, upon submission they are rebased on top of the current master branch, leading to similar lines of code than their neighbouring commits in the Git graph.

This large source code repository is very actively changed by several teams concurrently. Figure 6.2 shows the amount of commits per day. The number has reduced in the last few years, indicating that the modularization effort already has positive effects.

Most weeks the build of the master branch breaks at least once and has to be fixed as soon as possible, in order to not negatively affect both, the developers pulling from there and the deployment to later stages, such as the quality assurance cloud. Since there are also some infrastructure instabilities leading for example to dropped connections, it is often hard to tell if a red master build is really a regression or just an infrastructure problem. Additionally, the project suffers from brittle test cases that sometimes fail but work most of the time. The most common reasons for this are poorly randomized test cases and tests depending on the current time failing for example around midnight when a couple of minutes into the future or past is not on the same day.

Since a full build of the master branch takes between about 45 minutes to an hour, the amount of commits between two consecutive master builds varies greatly and it is often not trivial to find out which change caused a regression.

Several deployable web applications are built from the same code base through a complex graph of Maven dependencies and Spring configurations. The functional services *sales* and *after sales* have their own technical service. For common functionalities, like configuring global settings, a separate service is also built from this shared code base. Figure 6.4 illustrates this architecture. Users access the services through a reverse proxy that unifies the URLs by providing URL routing among other things.

All these issues led to the desire to decrease the size and coupling of the project and to allow every team to independently commit and „own“ one project, being also solely responsible for the build status and the deployment to later stages. Such a setup also reduces the impact of a broken build as only the developers working on it should be affected.

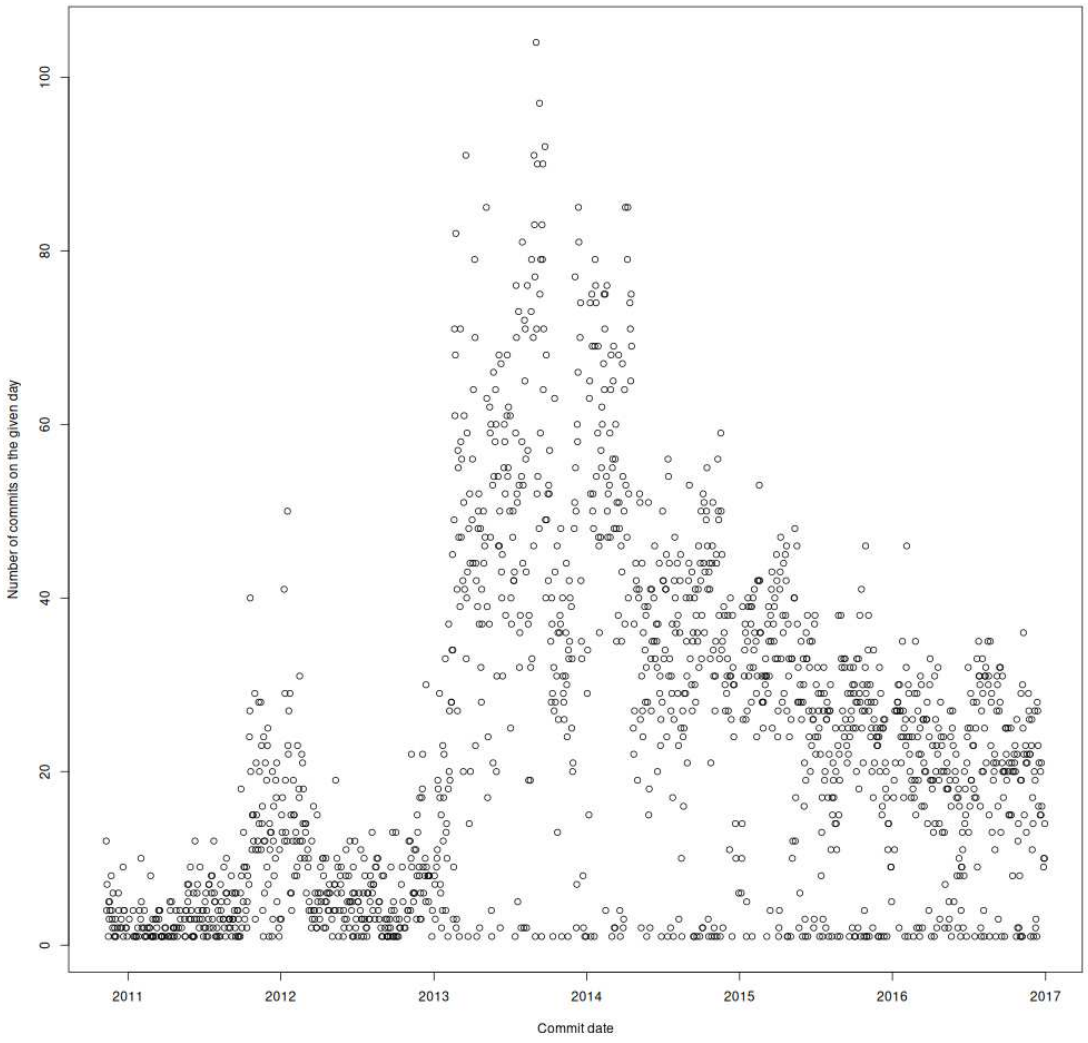

Figure 6.2: Commit history

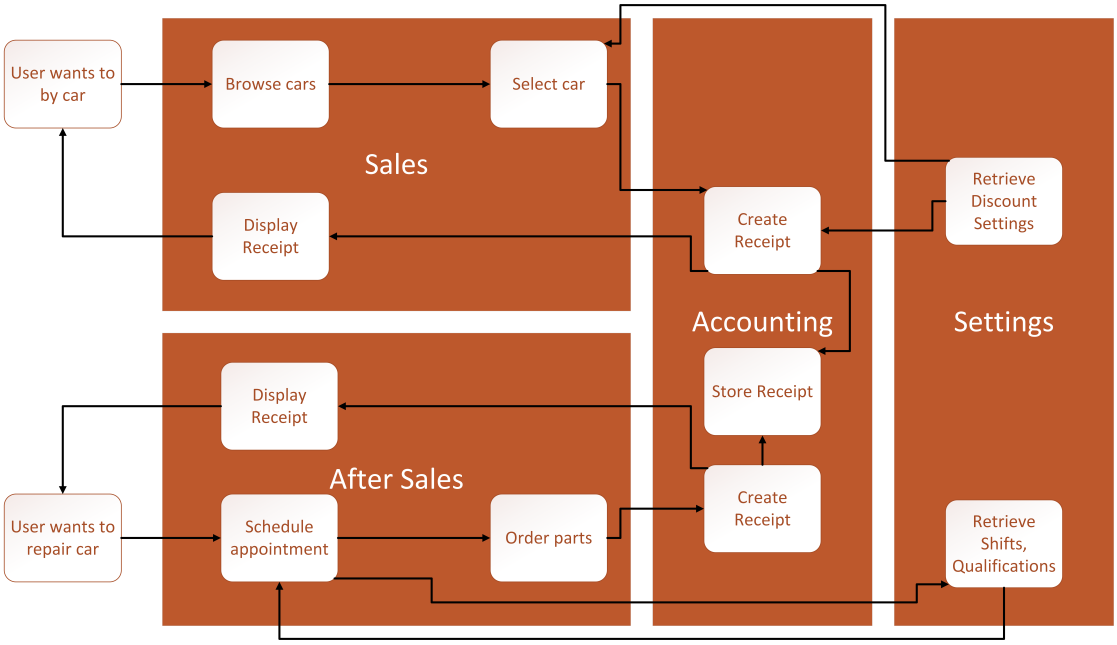

Figure 6.3: Functional architecture

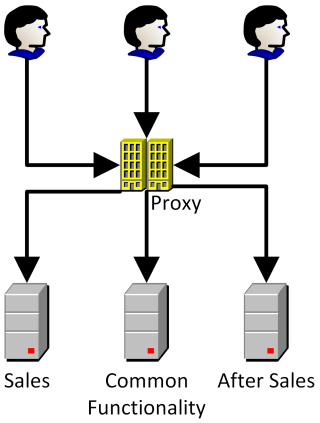

Figure 6.4: Technical server architecture with reverse proxy

Figure 6.5: Project organisation

6.3 Starting position

6.3.1 Organizational

Teams

Speaking on a high level, the project is divided into separate teams working on different parts of the whole software suite. The two biggest parts deal with the sales process of cars and providing services after sales. These two processes are supported by two separate executables. Currently there are six teams working on separate parts of the sales workflow and two teams on the after sales process. These teams vary in their responsibilities. For example, one team of the two after sales teams takes care of the core business processes of providing after sales services, while the other team focusses largely on integrating necessary third party integrations and providing more separate functionalities like the functionality to plan a workshop including appointments.

These two parts are directly visible for customers as they directly interact with these systems on a daily basis. Additionally, there is an accounting team implementing the business logic surrounding receipts and accounting in general. Other system parts call accounting with the information required to print a receipt and eventually get a receipt back. Normal users of the system never directly interact with accounting but use the receipts generated there.

The parts team is responsible for all parts related to maintenance tasks such as ordering new ones or managing their storage.

Since there is quite a lot of common code and services that allow communication between the other services, there are two dedicated teams that have no directly visible business cases and take care of the common infrastructure and the core setup. Managing the frameworks and settings falls in their responsibilities for example.

The test automation team is discussed in more detail in the *Quality assurance* section.

The full team setup also including management layers is shown in figure 6.5.

Over 100 people are involved directly with the development of the software, whether as managers, technical architects, software developers, testers or requirements engineers.

Releases

Both management and the developers consider continuous delivery as desirable in the company. The progress towards fully employing continuous delivery has not progressed very far. Management talks about already employing this development methodology, but admits that the duration between two consecutive deployments to a customer is currently fixed to one month, which equals two Scrum sprints and involves many manual interventions. This does not fulfil the academic definition of continuous deployment described by Chen and Virmani for which every single working commit is immediately deployed into production and to the customers [19, 74].

In this project full continuous delivery is currently not possible as a lot of manual testing is needed, in order to ensure the overall quality of the product. The quality of the automated tests is not enough to cover all partially really complex and highly state dependent checks performed by testers.

Upon every release database migrations as well as custom initializers have to be run in order to prepare the environment and database in the way expected by the applications. This process takes several minutes even if no relevant changes were made.

As stated previously, every month a new release is created and deployed to the pilot car dealerships. From a development perspective a new branch is created in Gerrit and fixes are cherry picked on top of this branch. The master branch requires a lot of fixes and those will be already fixed in the next release. Currently, there is only one release active and in need of fixes at a given point in time. Therefore, the cherry picking has to be done only for one branch.

Quality Assurance

There is a dedicated test automation team that creates End-to-End (E2E) tests for the whole application. This team is also responsible for maintaining a custom test framework that records videos of all test executions. In the case of errors it runs all failed tests another time to reduce the amount of false positives due to brittle tests. The fact that a test is brittle is recorded either way.

Creating unit and integration tests is the responsibility of the teams and in turn of every software developer creating new features or amending existing functionality. The state of unit and integration tests is ensured on every build by Jenkins and it is not possible for normal developers to directly push to master without passing these tests.

The E2E tests are run less frequently and it is very common that these tests fail. Once a day at midnight they are run for the latest master branch. Members of the test automation team then assess if the failures are errors in the test automation or if there is a defect in the production code. In case there is a defect, a ticket gets created and assigned to the tester of the responsible team who in turn assigns a developer to fix it. This of course creates a much slower feedback loop compared to the unit and integration tests.

Every team also has at least one dedicated tester that is responsible for testing all changes made by that team and finding issues that generally impact the functionalities the team is responsible for. Frequently a member of one team breaks functionality of another team. This is partially due to the partially very hard to comprehend dependencies between the parts of the software. Another reason are incompatible version upgrades in shared libraries. Adding new values or removing obsolete ones from enumeration classes broke other services repeatedly in the past. Running only the test cases for the changed parts is not sufficient here, but contract tests can provide more certainty of not breaking dependent services.

Development Methodology

The development methodology is specified as Scrum. The fixed sprint durations and relatively fixed amount of work, that is expected to be done in one sprint, help with planning when features will be available and in which release they will be included. This is important as the pilot customers request some features and expects estimations when it will be done.

Other markets are also in need of specific features. At the point of writing, a country with multiple spoken languages is targeted and features, like multi-lingual input fields, are implemented. In order to meet deadlines it is also critical to determine if the timely implementation of such features is endangered and if this is the case, to take effective actions.

However, all teams do not fully follow the Scrum practices. In some teams there are no sprint retrospectives, the sprint scope is dynamically adjusted and story points are not estimated effectively, resembling the Kanban process more closely compared to Scrum. Other teams follow the process completely. It is the authors opinion that if a team is less stressed to release new features and fix bugs very quickly, the more likely it is to follow the Scrum process fully. If extended periods of time are spend on fixing a very large number of bugs while under release pressure, performing a sprint retrospective or estimating story points can easily be seen as a waste of precious time.

Features

Most general, functionality is already implemented as proven by the productive usage of the software in two pilot deployments at local car retailers. However, this deployment is technically not the most challenging one.

At the point of writing, most new functional feature requests stem from requirements in different countries with additional requirements. Prime examples are for example the support for multiple languages or time zones within one tenant. The previous assumption, that all users and customers for one single tenant use the same language and time zone, is no longer necessarily the case. Legal requirements and commonly used third party services vary also from country to country.

These high level requirements are discussed when making buying decisions and written in contracts with deadlines assigned. The requirements and the deadlines get passed down from the whole project and higher levels of management to the individual teams, responsible for parts of the system, if they need to get active in order to fulfil these requirements.

The overall requirements specification and inquiry process is illustrated in 6.6.

Most of the time it is in the discretion of team leads or requirements engineers to break the high level requirements down into work items, which developers can accomplish within reasonable amounts of time and to provide more details. For questions, contact

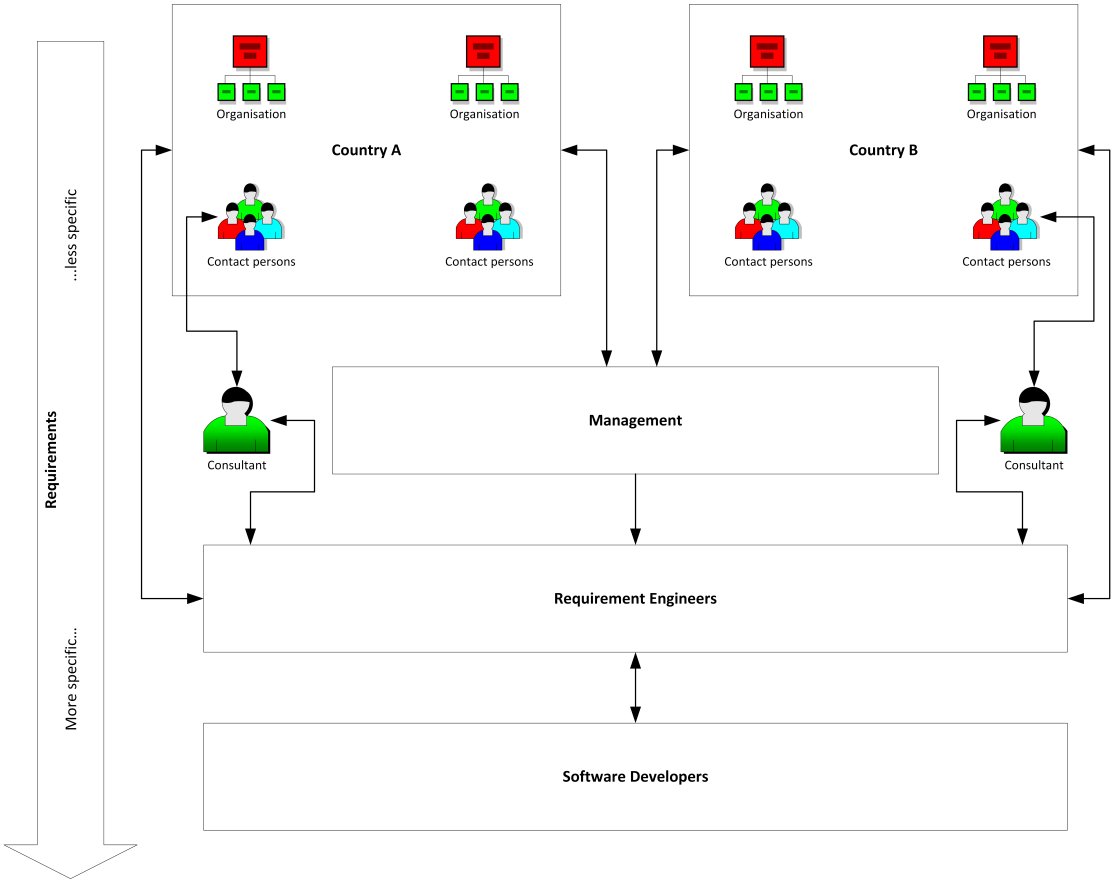

Figure 6.6: Requirement refinements

persons at the customers are either directly contacted or through consultants that have a general understanding of the big picture and the project.

Generally, teams are responsible for providing features in time and report foreseeable issues to management. Management sometimes pushes actively for certain features to be done with a very high priority.

6.3.2 Technical

Programming language

Java is used as the primary programming language. The most notable exception is Typescript for the Angular frontends.

Currently only Java 8 is used and there are no immediate efforts to upgrade.

Used frameworks

The most commonly used framework in the project is Spring. Backend code universally uses it but with different feature sets. Older code parts mainly use spring for dependency injection and configure the beans manually, using XML configuration. Newer parts make use of the newer annotation based configurations. One major difference is that the bean instances no longer have to be manually defined and are inferred by their type.

The Spring framework has many features. Out of these feature set dependency injection is used everywhere and also the transaction (Java Transaction API (JTA)) support is employed ubiquitously. Bitronix is the JTA implementation.

Mostly, functionality is accessible via a graphical frontend, additionally to an API operated by human users. Apache Tapestry is used as User Interface (UI) framework. The frontend suffers from poor maintainability and only few experts are available to help with more complex issues. It is possible to configure the IDE to reload the changes, made to the frontend, in order to test them in the browser without the need for a full recompile and application restart. This reduces the development time greatly but does not always work as intended. Once multiple modules are involved in changes this approach quickly fails and the development times become far from optimal. Additionally many developers raise concerns about the framework configuring too much automatically and making it hard for beginners to comprehend where a certain behaviour comes from.

Gerrit

Gerrit is used as a central code repository. Gerrit is a server that allows to host several Git repositories and for developers to pull changes and push them to it. One specific feature of Gerrit, over other Git servers, is the strict enforcement of code reviews and the linearisation of the history.

Code reviews are a common tool to ensure that only high quality code can be submitted. A common approach uses feature branches and when a developer is satisfied with the changes on this feature branch, a pull or merge request is made, other developers review the code and it is merged once they are happy with the changes. Gerrit works differently. Feature branches occur very rarely and developers do not directly push to the repository at all. Instead Gerrit keeps the changes and displays them for review. Only once they are approved and submitted, they are applied on top of the target branch. This is only possible if there are no merge conflicts between the head of the target branch and the parent commit of the change.

The specifics are configurable, but generally a commit can only be submitted to the master branch, once a developer and a continuous integration server approve the change.

Jenkins

Jenkins is a continuous integration server. This project is configured to have builds triggered automatically, once changes are uploaded to Gerrit and also once the changes

are submitted. The first type of build is important in order to be reasonably sure that the target branch, which is usually master, is not broken by the commit. The latter build of the merged change is important to catch the rare occurrence of a rebase breaking the build or software and to deploy the artefacts. These artefacts can range from Maven modules to fully executable services and docker images.

Jenkins also performs quality analyses and calculates code coverage metrics.

Notifications about the build state are pushed to a chat server. This allows everyone to get quickly notified about failed builds and to discuss the reasons and mention and thus notify people well suited to fix them.

Sonar

SonarQube automatically generates software quality reports based on metrics and heuristics. In Java calling the `equals` method with an argument object of a completely different type might be, for example, a bug. Writing long functions or functions with many possible nested control structures is also a code smell and SonarQube will display according warnings.

The quality across the project, as calculated by SonarQube, varies greatly. Especially larger and more complex parts often have poor quality. SonarQube has about 650 issues qualified as bugs and 70 identified as vulnerabilities at the point of writing. The quality gate is failing, the reliability rating is D and the security rating E. SonarQube estimates that fixing all issues would take about 550 days.

Some parts or packages, especially newly written ones, have better ratings and more of a culture of regularly looking for new issues and fixing them in a timely manner.

Docker

Docker allows easily to package an application and its dependencies into one deliverable format. Since the dependencies and some configurations are included in the package the probability for the software, only working in some environments, is reduced.

Docker or in general containerization is often compared with virtual machines. The creation of a (Docker) container is completely scripted and the image is mostly not kept up to date with updates, but discarded and a new version is build. This is completely in line with the DevOps principles, specifying all configurations as code and not manually patching on the configurations of single machines. On a more technical level containerization is a light weight alternative to VM based virtualisation, as the (Linux) kernel is shared with the host system and only the userland programs are included in the image.

Docker Compose allows to configure and get a cluster of all required services up and running with one command on developer workstations or laptops. The downside is that at the point of writing, if several services are started concurrently, the amount of ram

can quickly reach about 15 gigabytes. Together with a browser and an IDE the resources of a laptop can quickly become scarce.

Due to the monolithic nature of the program and since a single large database is used for persistence, it is not easy to set up a development stage shared among all developers and run only the current actively changed parts locally. Multiple tenants can be configured using different URLs for services but this approach is not well supported.

In production OpenShift is used as container orchestration framework. The same automatically built images are used there that also run on development machines. Helm charts are used to configure the setup.

The logs are aggregated in a log server, based on the Elastic Stack, and can be searched via Kibana. This is very helpful to find the cause of bugs in non-local environments.

6.4 Identified problems

As stated previously in this chapter the major problem currently faced is poor maintainability. It is partially caused by slow feedback cycles. Some services, built from the monolithic code base, are way too large and have a lot of dependencies. This increases the startup time and memory usage dramatically and leads to bad understanding by developers.

The monolithic nature of the software does not cause all the problems listed in this section. Modularizing will not solve all of them. They are listed here primarily for a better understanding of the project environment.

One major source of complexity is the sheer number of configuration options supported by the software. Since the software is supposed to be run in vastly different countries with diverse legal restrictions and technical backgrounds by car retailers, providing varying services, there are a lot of configuration options needed. Configurations are also structured hierarchically. This means that on a global level some configuration values can be set but on a lower level like tenant or site these settings may be overwritten. This is illustrated in figure 6.7. The fact on which level which options can be configured is configurable in itself. Validity dates may exist for configuration options. For some configurations only one value is sensible, like the value-added tax (VAT) in a country. There are more complex options for which a list of active values are sensible. An example is the setting for which car brands are supported. It requires the ability for an undefined number of car brands to be active at any time.

There is a lot of complexity in the details and currently the configurations can be made in two different deployables, build from the largest repository. Not all configurations can be made in either one, some are specific. Unfortunately, this implies that in order to make configuration changes, which developers quite regularly need to do, these two services need to be build and running taking up precious time and resources.

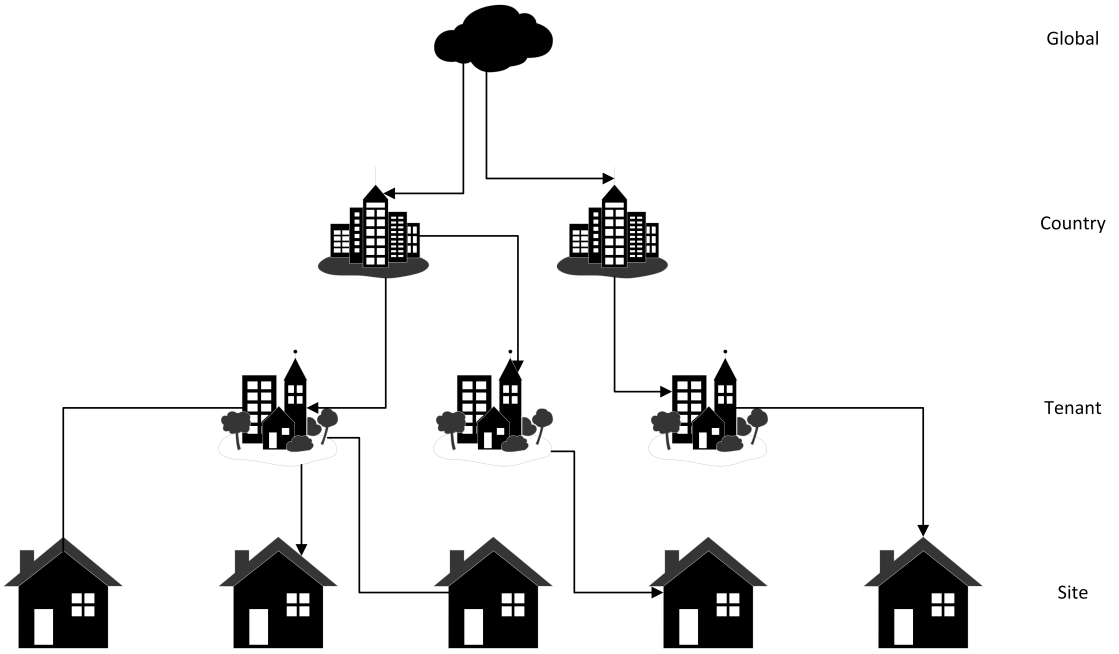

Figure 6.7: The hierarchical organisation and configuration hierarchy

The test coverage in a general sense should also be improved. A long term goal of the project is to employ continuous delivery automated tests that can provide a high level of certainty that the program works as intended are required. At the point of writing the tests are both brittle and far from sufficient to provide confidence. Manual tests performed by human testers find large numbers of bugs in quality assurance deployments. The latency until the next automated E2E test run and following issue delegation also makes it harder to directly revisit the problem and fix it in time by the developer that caused a regression.

The project is running for about ten years now. This is a long time even for a project of this size. Slow development speed is one reason for this. This slowness is partially caused by technical issues, like slow startup times and feedback cycles. Older technologies like Tapestry, which often force developers to recompile the whole application just to make frontend changes, also add to this. Executing all tests locally is not often done and the time until the CI server responds adds latency. In the authors experience the issues there are often small like a newly added but not explicitly declared Maven dependency or Checkstyle issues which break the build. Even if the issue is relatively small, the newly begun work has to be suspended, one needs to switch over to the old work and push another change. If this process is repeated for a few times a lot of time will be lost. Additionally, the context switching is not beneficial for a developers concentration.

There are also some communication problems. Through the various organizational layers some information and possible alternative solutions get lost. There are consultants that

directly interact with users of the system. They interact with requirements engineers which in turn interact with developers. The majority of the developers is not familiar with the details of the domain, be it car retail specific or details in accounting. There have been several occasions in the author's own experience where there were misunderstandings, both between consultants and requirements engineer on the one hand and requirements engineer and developers on the other hand. Both kinds of misunderstandings can quickly cost several hours or even days.

The terminology is often confusing for newer team members. The intuitive meaning of a *ConfigurationValue* compared to *ConfigurationProperty* for example is not self explanatory. In most cases there is a documentation of some sort in Confluence, but since this wiki is very large and often not up to date, finding the right page and its usefulness should not necessarily be taken as a given. The use of specific classes in the grander scheme is rarely documented in the code itself as Javadoc even though the usage of Javadoc is enforced using Checkstyle.

Another issue felt by the author and one colleague is that requirements are only handed top down and there is little flexibility for developers to propose alternative solutions that fit better with the overall architecture. There is very little technical pushback against requirements that are hard to implement, given the current architecture and could be achieved another way. It is often the case that over time even more special cases are revealed that make the original design of components suboptimal. Refactorings are not always performed as not enough time is available, even tough developers are aware of their need, and in some cases the maintainability suffers significantly.

Multiple communication channels are used to interact with team members and distribute information. The two major ways are Microsoft Teams, a chat service and conventional email. Additionally, several other chat servers exist. Some members raised concerns that this heterogeneity makes it difficult to reliably reach everyone in order to spread important technical changes, where others need to take actions on their part. Teams is preferred by company strategy. Various automated messages are sent to its channels which are often not relevant to a single developer. For this reason important messages indicating a need to act may more easily be missed.

CHAPTER 7

Application of currently selected approach

In order to improve the situation, described in chapter 6 and enhance maintainability in the future, the decision was made to move to a microservice architecture.

Technical architects and management decided on the approach which is described in detail in this chapter.

There are efforts to reduce the size of the current monolith and split out separate services, belonging to business units and having their own bounded context. This is intended to increase the speed of development, the maintainability and scalability. Management expects teams to own „their“ part of the whole software system more independently. These extraction efforts can be seen starting from 2017, as the number of lines of code is abruptly reduced several times. Figure 7.1 shows that the number of lines of code has started to repeatedly swiftly decrease by significant amounts. This stems from the fact that some modules have already been moved out to dedicated services.

Unfortunately, this does not imply a complete downwards trend. New features are also added rapidly and the number of lines of code in the largest repository increases slowly, followed by drastic decreases averaging almost no change. However, there is hope that since the code in this repository is already split in Maven modules and an increasing number of service facades replace direct access to other modules, internal matters will be possible to split off even more easily larger pieces in their own service or group of services in the future.

Figure 7.2 shows the sum of lines of code in all other repositories for a given point in time. This sum is by now significantly larger than the count of the monolithic repository. This means that the largest part of the source code is not part of the largest repository anymore.

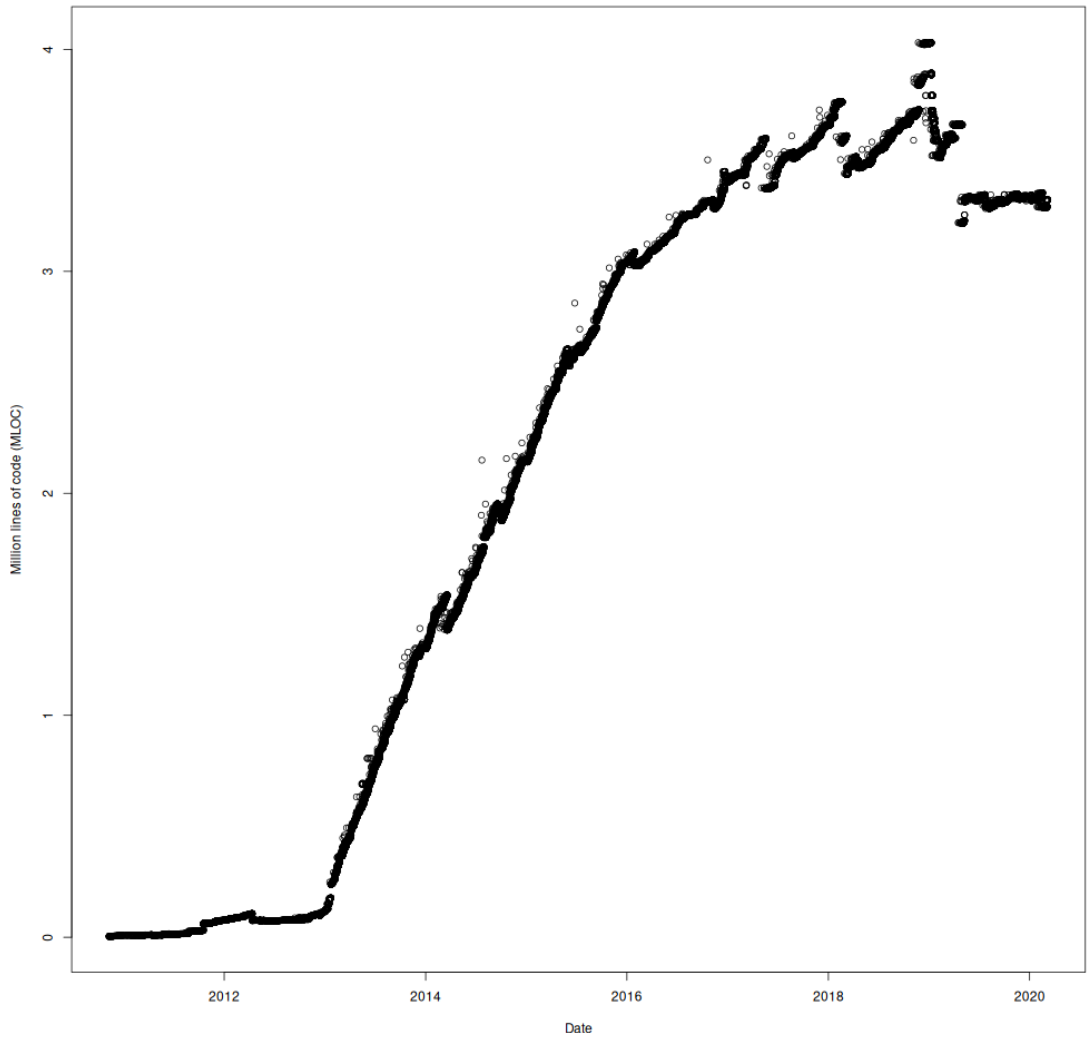

Figure 7.1: LOC history of the monolith

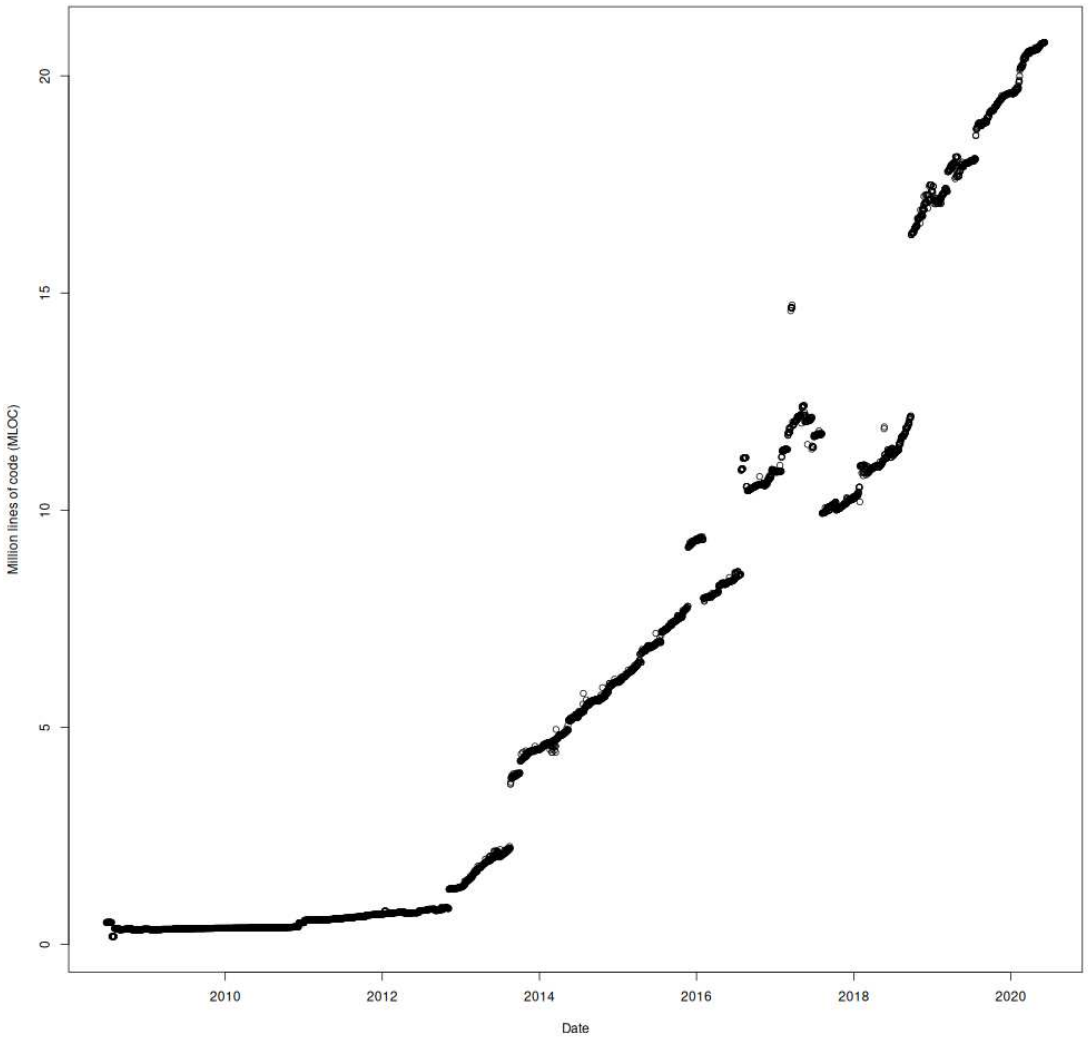

Figure 7.2: Summed LOC history of the other repositories

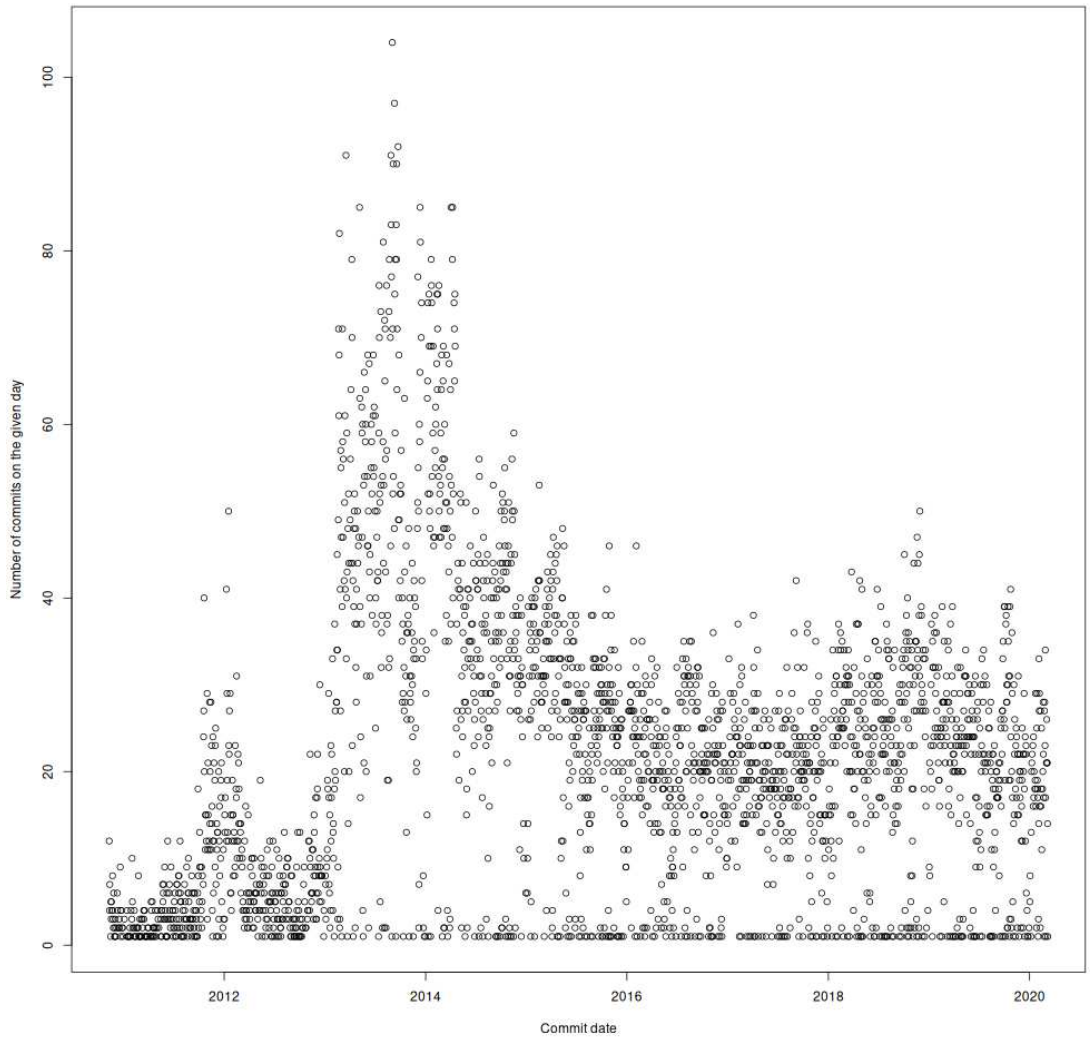

Figure 7.3: Commit history in the monolithic repository

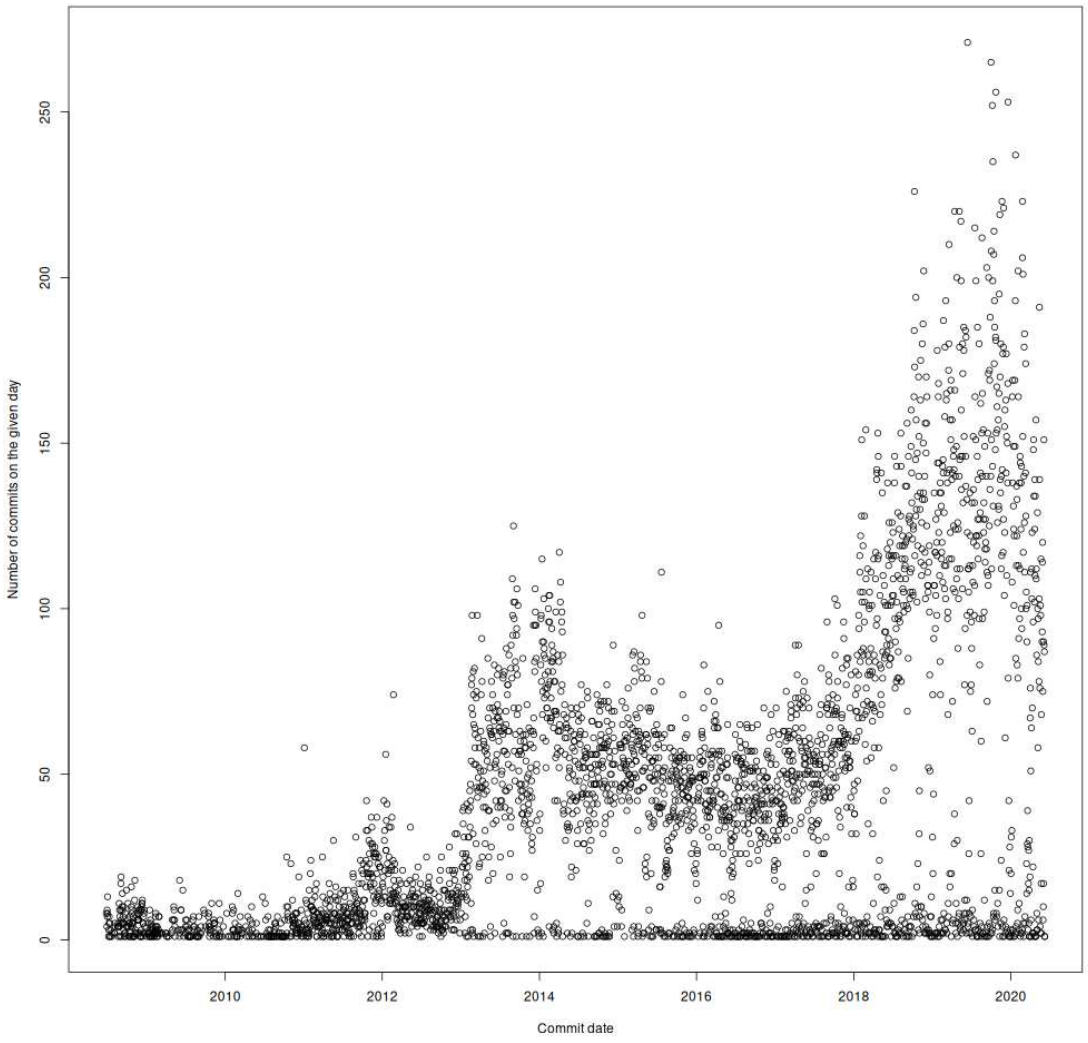

Figure 7.4: Commit history in all other repositories

The commit counts per day remain similar, compared to levels before the microservice decomposition started as visualised in figure 7.3. This is not surprising as the number of lines of code also does not show drastic changes on the average.

Figure 7.4 shows the number of commits per day of the other repositories. These increase significantly, starting from 2017.

Currently, there are 36 different deployables configured to be deployed on the OpenShift clusters. The number will increase in the future as the decomposition progresses and developers split additional services out of larger services or create entirely new services.

A noteworthy positive example of the already achieved progress is the accounting subsystem. The accounting team has managed to completely migrate all of their code to microservices. They are the first and so far only team to achieve this goal.

One of the problems described earlier is the slow development time with Tapestry. Additionally it is hard to find experts that can help solving technical problems. All these concerns and the increasing age of the Tapestry framework lead to the adaption of Angular for new frontend developments. It is not economically viable to immediately switch over to Angular as sole frontend, since substantial investment in the older Tapestry frontends were made. The developing microservice architecture allows to replace the GUI layer of single services with rather low effort and only local impacts.

Developers generally consider the development time with Angular to be much faster and like the development tools. Additionally, the knowledge of Angular among the developers, especially new hires is much higher compared to Tapestry. The clear separation of the backend provided by Spring Boot and the frontend as Angular application is also widely considered a good cut. Data transfer is clearly defined as REST API and can not happen implicitly and hidden as it is the case with Tapestry. The APIs are well defined for the given use case and the amount of on demand data loading is minimal. Thus, the performance is generally better for the new kind of frontend.

Previously, a set of Maven modules named *core* was actively maintained and intended to contain common code. The older code in the project has various dependencies to these *core* modules. It is not clearly defined how domain specific these modules should be. Various developers change these modules without necessarily knowing all use cases. Lately, this has become a very brittle piece of software. Since its usage is very widespread, the potential for breaking another software is large.

There are few meaningful tests that catch introduced incompatibilities. It is only possible to catch incompatibilities in a depending module, when its version is upgraded to the newest release of the *core* module. Therefore a CI job does that automatically upon every release of the *core* modules. Nevertheless breakages are not a rare thing.

These problems lead to the realization of the project, that shared code is a liability, as also described by [39]. Most code that is part of the central infrastructure, is already provided by third party libraries such as Spring Boot in the newer services.

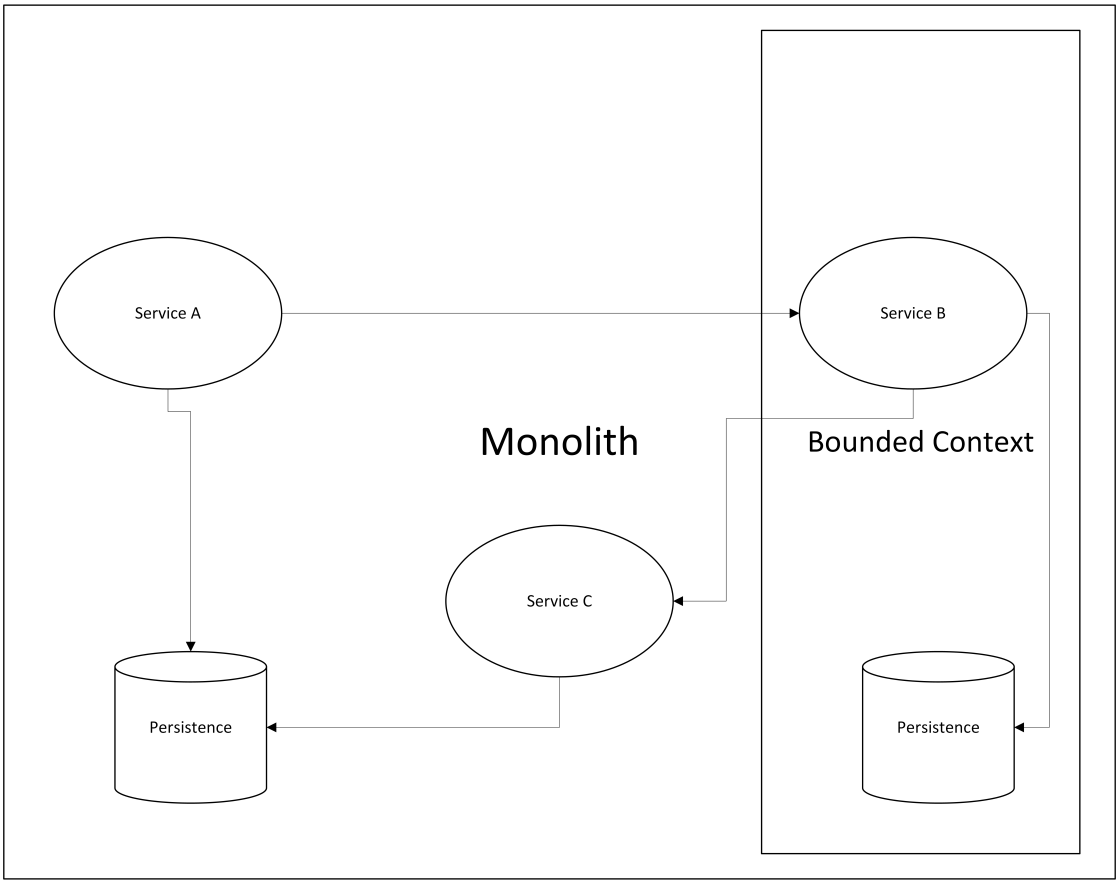

Figure 7.5: State before the migration

7.1 Technical process

7.1.1 Moving existing functionality out of the monolith

In order to move a part of a monolith out into a microservice, architects need to identify a suitable part first. They take business considerations into account, following the approach. They identify a bounded context within the monolith that is suitable to be moved into a dedicated microservice or set of services. To make the migration easier, the bounded context should not be central in the monolith with huge amounts of chatty interactions, but rather some bordering functionality. A bounded context with only incoming calls, but with no outgoing calls into the rest of the system, would be ideal. Technical architects make an intuitive decision. Usage and dependency statistics provided by IDEs help them to make an informed intuitive decision.

Once the decision is made which bounded context should become its own microservice, a process consisting of several steps begins.

An identified context within the monolith and the general state before the migration

can be seen in figure 7.5. There are three services. *Service A* calls *Service B* which calls *Service C*. Only *Service B* is inside the bounded context which should be transformed to a new service.

Definition of Facades

Generally, a facade hides the complex internal matters behind a clean interface. From the outside the system part should only be accessed through this facade.

For the system the desired communication pattern are rest calls. Therefore the used facades have the form of rest interfaces.

Two new Maven modules are created. First there is a module ending with `-public-api`. A second module ending with `-rest-api` is also created. The first one contains so called public DTO, the latter the interfaces of the REST controllers and implementations of clients which use these rest endpoints.

The software system already makes extensive use of DTOs. To distinguish between the internally used ones and the ones that are explicitly meant to be part of a public API, which is exposed over REST, the concept of Public Data Transfer Objects (PDTOs) is used.

The REST controllers implement some interfaces. The mentioned PDTOs are used as data types.

In the project dedicated REST clients are also provided for every REST interface. The idea is to provide commonly required convenience methods, endpoint specific sensible caching and to allow for post processing on the client side. This post processing makes it possible to store data server side and transmit it in a smaller but less easily usable form, while providing easy and intuitive access to users of the REST client.

Technical architects search for common usage patterns and try to define a small but usable REST API that can replace all prior usages.

Facade implementation

As the second major step in the migration process the facades in the form of REST services have to be implemented. Technically this means providing an implementation to the interfaces provided in the `-rest-api` Maven module.

When the same functionality already exists this process is trivial. The difference between the internally used DTOs and the shared PDTOs, which are transmitted over REST, is often minor. In the most common cases only a few fields such as the numeric database primary key are missing in the PDTO version. Numeric Identifiers (IDs) are only unique per database. In order to identify entities globally, Universally unique identifiers (UUIDs) are often used.

In most cases simple mappings and service calls are sufficient, but in others more complex logic has to be implemented.

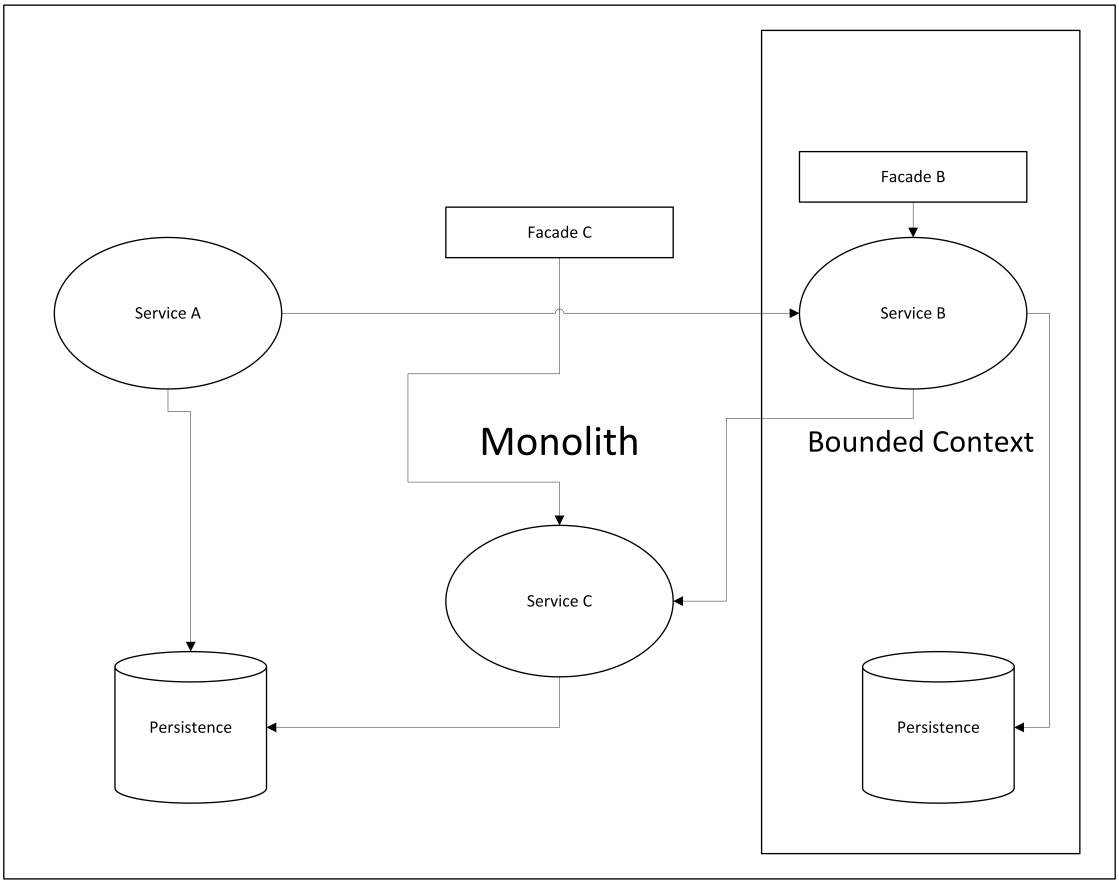

Figure 7.6: Facades are added

Figure 7.6 visualises the state once facades and their implementations are added. In this simple example services *B* and *C* have a 1:1 relationship to their respective facades. In more complex and realistic scenarios with more than one call in each direction this is unlikely. *Service A* does not need a facade since it is not called from the outside.

Using the Facades

The third major step in splitting out a part of a monolith into a microservice is to actually use the facades in the form of REST endpoints and clients.

Developers search the rest of the monolith for usages of the part which is about to be split out of it besides the code contained in the previously created Maven modules. The found usages are direct usages circumventing the defined REST facade and have to be replaced with calls to the provided REST client.

IDEs help finding such usages. The project uses the Maven Enforcer plugin to force the declaration of used dependencies. This is very helpful here as any remaining usages

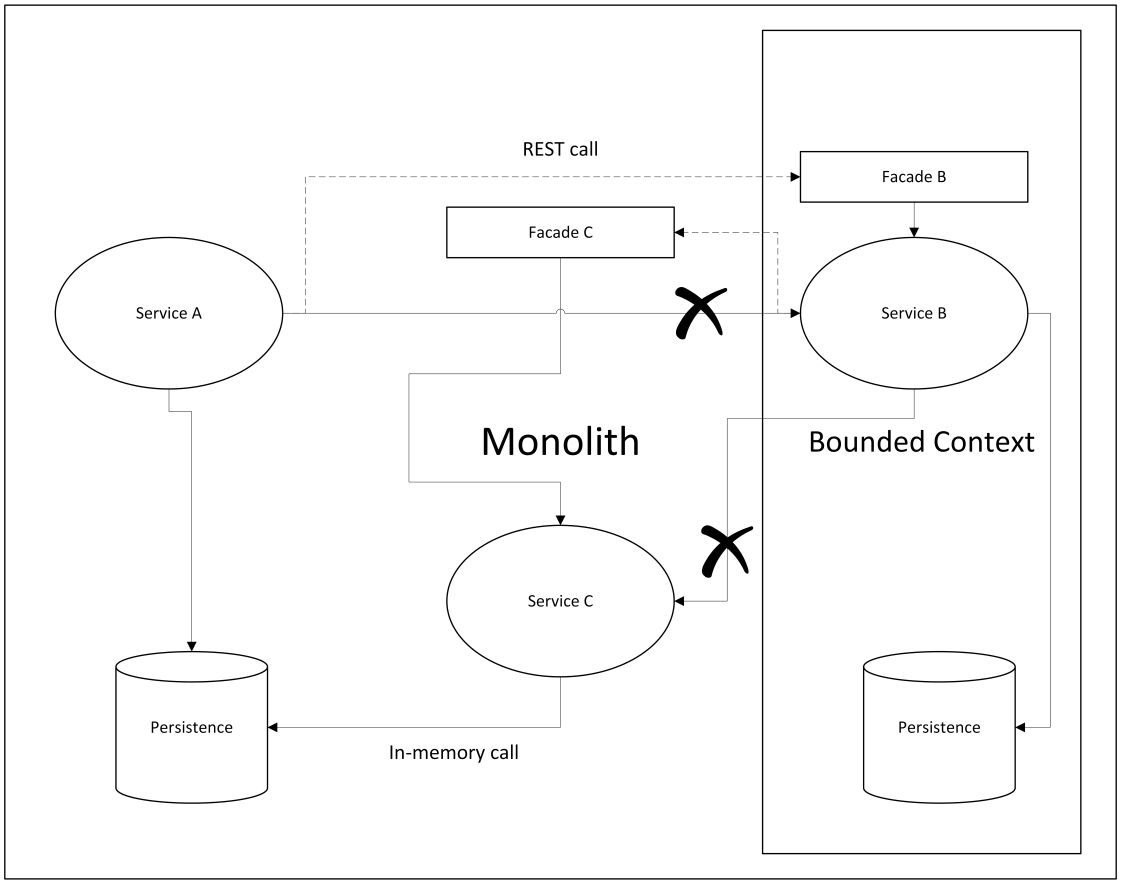

Figure 7.7: Facade calls are added

become more visible and it is apparent which modules are still relying on direct accesses. Without this Enforcer plugin a Maven module does not explicitly need to declare a dependency, if the dependency is provided transitively by another dependency on which the module depends explicitly.

Figure 7.7 shows the process of replacing in-memory calls across the new border with REST facade calls. Normal arrows represent in-memory calls, REST calls have arrows with dotted lines.

Many usages can be replaced very easily by REST calls and the corresponding PDTOs. Transactions become more complex. Distributed transactions are not desirable and also not supported in the system.

In many cases it is possible to move the whole transaction to one service and to provide aggregated data from the other service.

If more complex isolation and transaction semantics are required, a case by case analysis is required and technical architects from all relevant teams discuss a feasible solution.

The visualised case also assumes that the database tables can be cleanly partitioned between the remaining monolith and the new microservice. Since bounded contexts are used, this is likely the case. In the remaining few cases the technical architects make intuitive decisions on how to proceed.

Partitioning tables between the remaining monolith and the new microservice also implies losing the possibility of database joins. That can be solved easily by sacrificing some performance and performing a logical join over REST. In these cases it is important to just perform one REST request with a list of all PDTOs one is interested in, instead of one REST request per PDTO. So far the performance penalty following this approach has been perfectly acceptable.

In this stage of the separation process more and more interactions within the monolith are transmitted over a REST API. The monolith calls itself over REST. This is only a temporary solution since it suffers from disadvantages of both monolithic and distributed architectures such as slow development time, no independent deployment and REST calls are slower than calls performed purely in memory.

As stated previously, it is optimal if the part which is about to become a microservice has no outgoing calls to the rest of the monolith. If it does, these calls also have to be replaced. The same process applies but in the other direction. A REST facade is defined, it has to be implemented, the part which is about to be separated from it - which in this case is the newly created microservice - has to use only these REST calls to interact with the remaining part.

Once there are no more direct calls from the remaining monolith into the separating part and also no direct calls the other way round, this phase is complete.

Sometimes the technical architects that envisioned the REST APIs forget about some details and the interfaces or PDTOs have to be adapted.

Moving out the separated service part

The final part during the separation process results in a new microservice.

A new Git repository is created, containing the Maven modules ending with `-public-api` and `-rest-api`. Another repository is created which contains a new Spring Boot application. All non-public Maven modules of the bounded context within the monolith are moved out of the monoliths repository over to the new repository with the Spring Boot application.

Public and private code and data structures are strictly separated that way. In other words, a clear separation of an API and its implementation is achieved.

Generally, the code in the private part, which contains the actual implementation, changes a lot more often compared to the public repository. Only releases of the Maven modules from the public repository are of concern to developers of other services, as they are only interested in the API and not the implementation details.

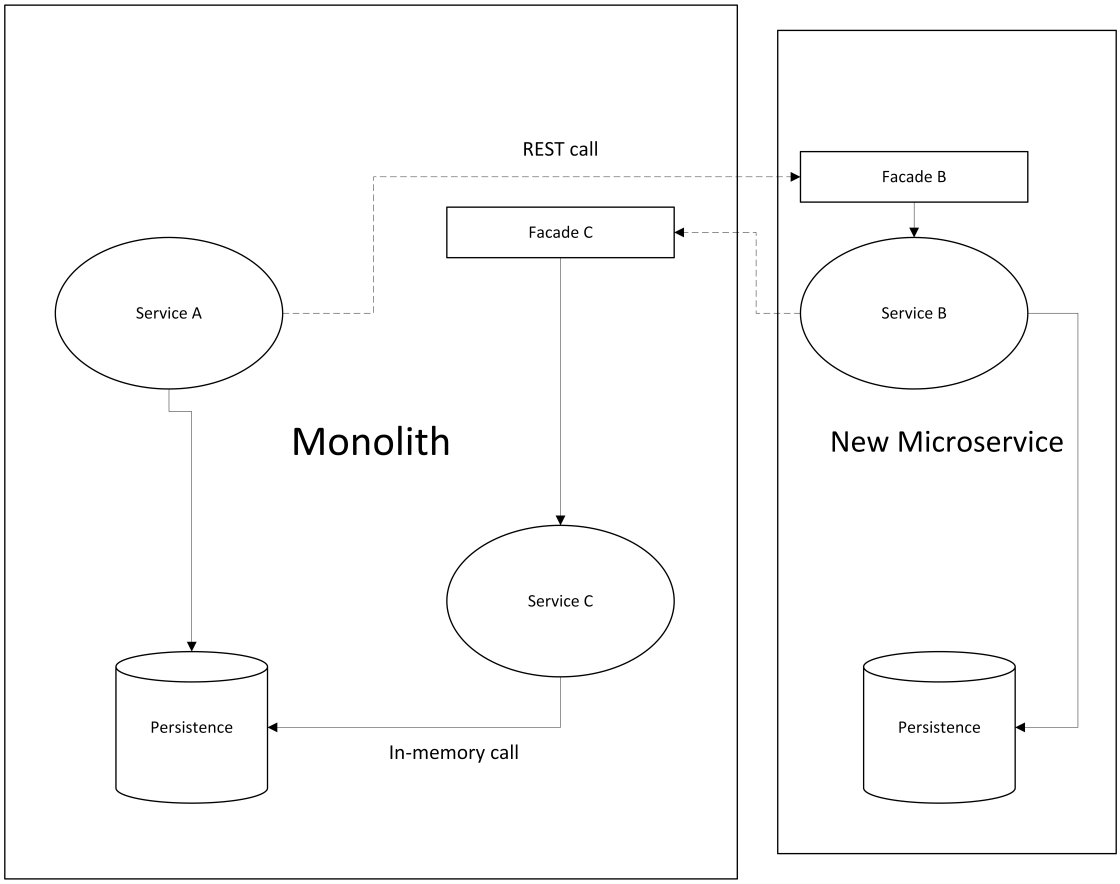

Figure 7.8: Final separation state

Using Jenkins a new release of the public API modules is automatically created upon every push to the master branch. Releasing a set of Maven modules, which are contained in a single Git repository and which share a version number, is easily possible that way.

The new application is also build on Jenkins, Sonar and the artefacts are deployed. Docker images are build and Helm Charts configure how the service should run in a cluster. The URLs for accessing the new microservices functionality is changed, as it is no longer part of the monolith and therefore has a new base-URL.

Figure 7.8 shows the state once the separation is completed. A new microservice exists and interacts with the reduced monolith via REST calls.

7.1.2 Rewriting part of the monolith as a microservice

The software quality depends on the area. Particularly problematic pieces with bad maintainability are not moved out to their own microservice, but they are replaced by a completely new microservice. In the short term it may require more effort to rewrite that

functionality, but the reward is substantial as the bad old code is completely replaced.

Motivating example

An issue already described in chapter 6 is that the configuration of the system is relatively complex and two large monolithic services, built from the largest source code repository, have to be started in order to configure the system fully. These two services handle a lot of completely unrelated functionality, too. The configuration system has grown a lot over time and the system suffers from poor maintainability. Recently a developer needed more than a day to add one simple configuration option.

Therefore a new dedicated configuration service is built. It handles all the complications and provides end users with a web-UI to change settings and developers of other services can easily query the currently for their context relevant values of a setting. Since there is already a lot configured in various installations this old data must be migrated to the new system. Therefore the new service can import the old configuration data. Afterwards the old configuration parts of the old monolith are completely removed. This differs a bit from the normal process, described in section 7.1.1, since the old code is not moved to the new service but is replaced by a completely newly written service.

Facade introduction and implementation

In the beginning there are a set of services which should be removed. In figure 7.9 this is *Service B1*. *Service B2* in the microservice provides similar functionalities and it is destined to replace *B1*.

As a first step facades around the functionalities in the monolith, which are about to get replaced, are required. These facades also have to be implemented. The first implementation delegates to the existing implementation within the monolith.

In figure 7.10 the *Facade E* is added around the functionality of the services, which are about to be replaced. For simplicity's sake the example this is just one service, but in most cases there is more than just one service.

The REST endpoint facade (*Facade F*) is also added in figure 7.10. This is the interface describing the corresponding REST endpoint of the new microservice. These facades have to be implemented.

All direct usages of the soon to be removed services have to be replaced with calls through the facade (*Facade E*) in figure 7.10.

REST client facade implementation

Since a new microservice is not immediately expected to have all functionalities the old monolithic implementation has, the migration process takes some time. During this time one can use Spring profiles to switch between the old monolithic implementation and the new implementation which calls a microservice.

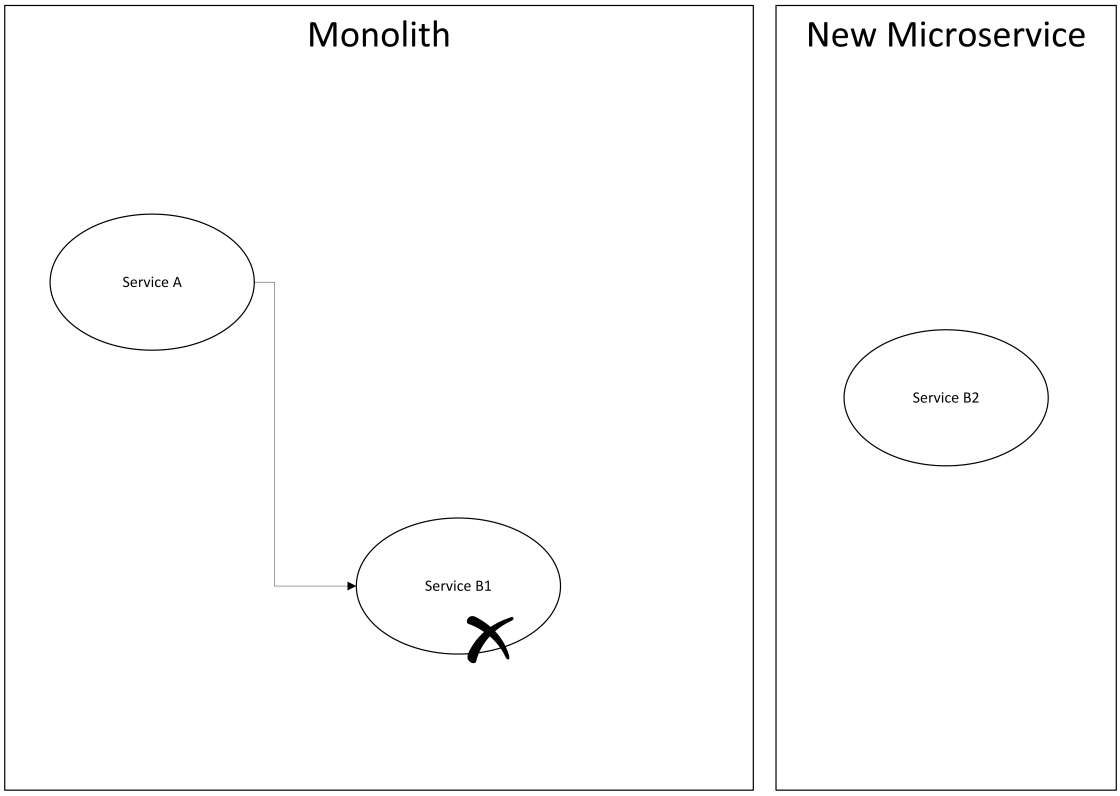

Figure 7.9: Initial state during partial replacement

This is possible by providing a second implementation to *Facade E* which calls *Facade F* via REST. Figure 7.11 visualises this possibility.

In the end developers remove the old implementation (*Service B1*) and the implementation which forwards to the other service becomes the only implementation. The new implementation is then always active, independent from Spring profiles.

7.2 Build times

Fast feedback about made changes is highly desirable. Context switching takes time and if a developer starts working on a new problem, before getting negative feedback to the old one productivity is lost. Low build times contribute to better maintainability and build times are easily measurable attributes of maintainability.

There is a central Jenkins server with multiple build slaves configured that verifies every pushed commit, before it is eventually merged. Theses build times are displayed in figure 7.12 for the large remaining monolithic code base and in figure 7.13 for the already mentioned configuration service. The y-axes represent minutes while the x-axes show the different builds in a historic order. Both graphs contain a number of failed builds,

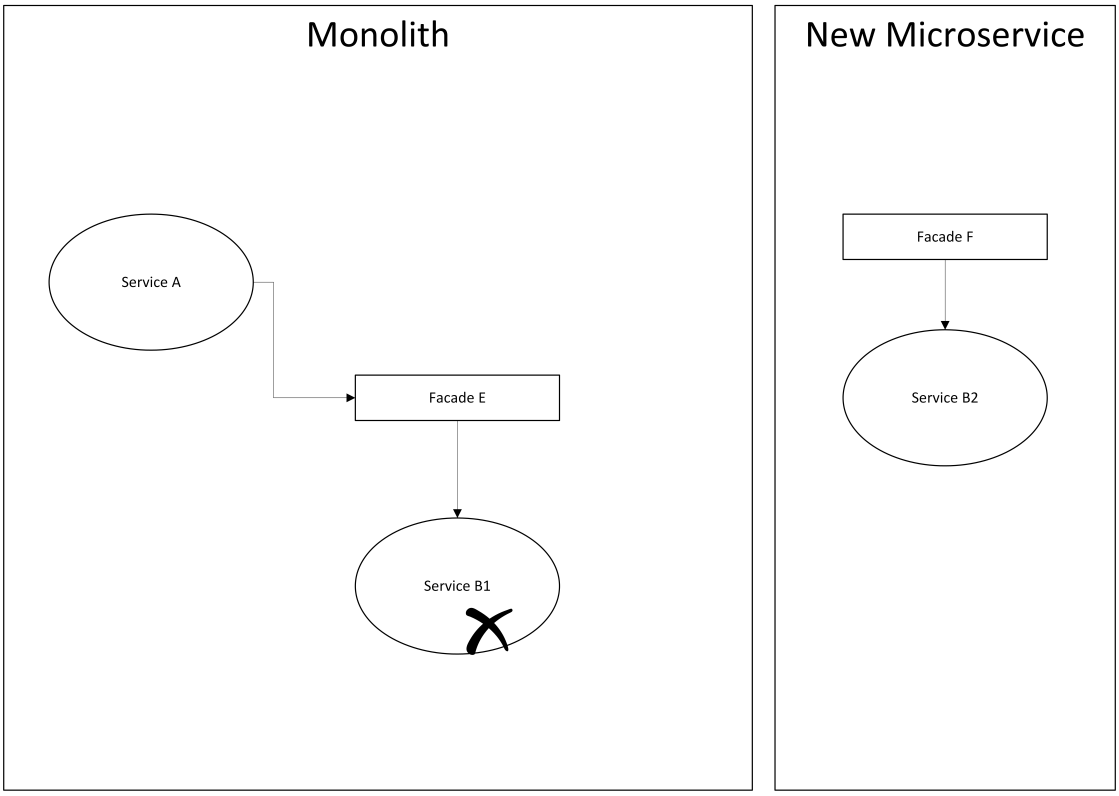

Figure 7.10: Introduction and implementation of facades

indicated by the part of the graph being red. This is not worrying as these builds are not yet on a shared branch and just represent any work a developer uploaded to Gerrit. Grey indicates cancelled builds which mostly occurs if a newer version of a patch set is uploaded and the old build is automatically cancelled.

As the code base is significantly smaller, it is not unexpected that the build times of the microservice are on average far lower than those of the monolithic code base.

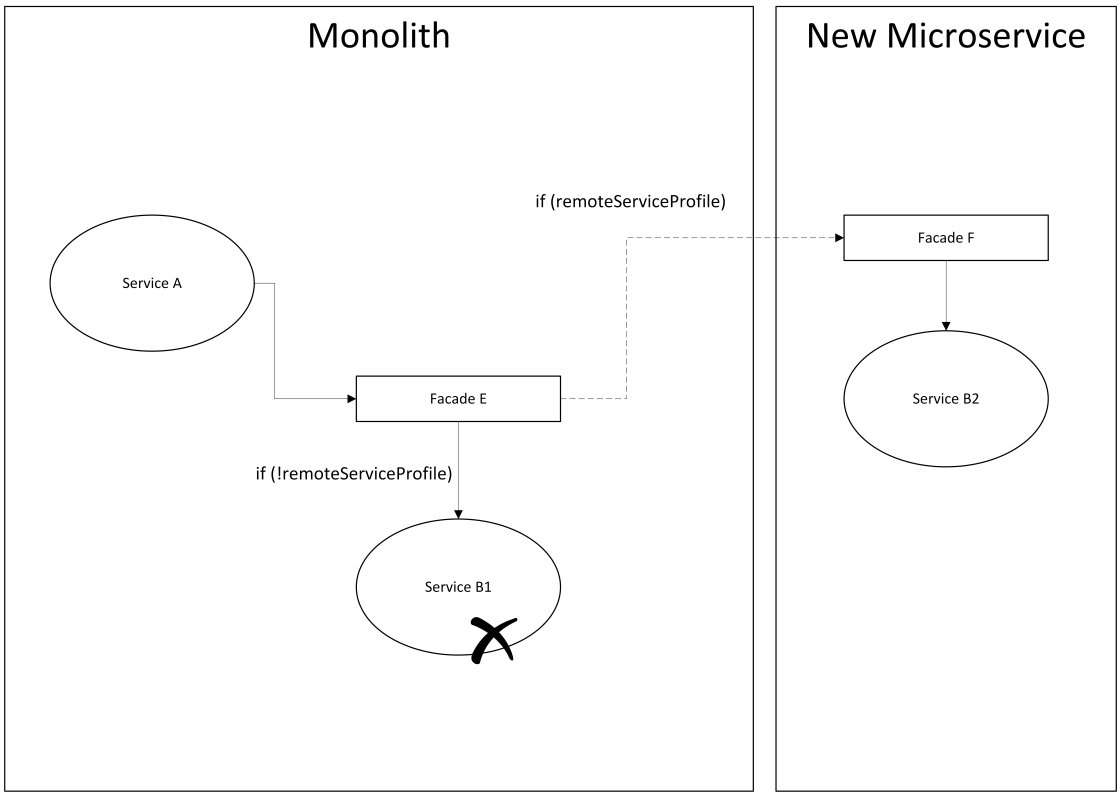

Figure 7.11: Profile controlling local or remote implementation

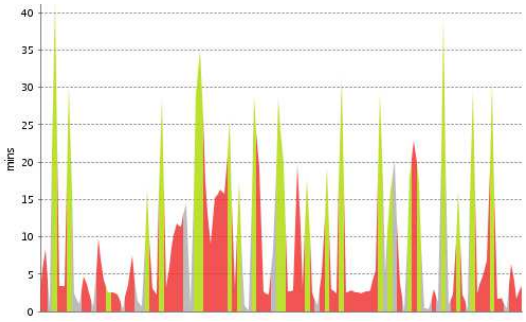

Figure 7.12: Jenkins build times for monolithic repository

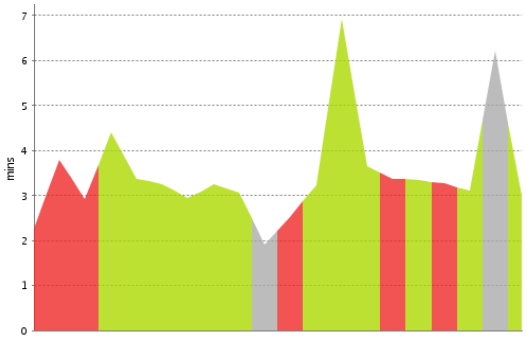

Figure 7.13: Jenkins build times for configuration microservice repository

CHAPTER 8

Alternative approaches

As stated in the previous chapter, there are some issues with the current state of the project. Many of them are caused by the monolithic nature of the project and development style. The strategic decision to move towards a microservice architecture was made.

While a current approach exists, which is used in order to decompose the monolith and improve upon these problems, alternative approaches are also considered. These alternative approaches, including their advantages and disadvantages to a large industrial project, are discussed in this chapter.

8.1 Criteria

To sum up, there are several properties the system has. Only approaches that fulfil these can be considered as alternatives for decomposing the system.

To qualify an approach as alternative to the current approach, it must fulfil the following properties:

Project size: The approach must be viable for a project with slightly over 100 persons directly involved. It is not economically viable to halt all other development activities.

Multiple functional teams: The project has reached a size where by far one team is not enough anymore. Following Scrum, best practices teams are organized cross functional. Different skill sets like requirements engineering, software development and testing are present within every team. The teams are spread over different cities and even European countries.

No formal specification: The requirements for the program are rather loose and there is no specification available. It is not economically viable to create one for this project. It is also not common practice for projects of such a size. This rules out more formal and automated approaches.

Java as programming language: The programming language is, as previously stated, Java. Maven is used for dependency management, and Docker containers are used. These technical facts also may rule out some approaches.

Already in production: Since the project is already running in production, all upgrades have to run smoothly, data has to be migrated and downtime is not acceptable. There are several development and test stages and this requirement is not too hard to meet.

No need for optimal solution: There is no need for an academically provable best solution, according to arbitrary metrics. While for example good overall performance is very important and the resulting modularization should avoid creating slow and chatty integrations, achieving the absolute best theoretically possible performance is not a goal. Maintainability and relatively low effort for the modularization are much higher priorities.

There should be no black box code transformation: Since understanding a large code base requires a very large effort of many people, completely changing the architecture in one large black box step is not acceptable. A huge amount of code understanding and knowledge of the structure would be lost. This is economically not feasible at this point of time.

8.2 Consideration of academic approaches

As stated in chapter 4, the 25 best search results of Google Scholar for „microservice decomposition approach“ are subsequently considered.

The search is not limited to a certain freshness, meaning the search results could have been published at any point in time. In this case the oldest found result in the top 25 is from 2015 bearing testament to the novelty of the topic.

1. „Service Cutter: A Systematic Approach to Service Decomposition“ by Gysel et al. [35]

   The approach described in this paper uses models and use cases to calculate good modularizations. Loose coupling and high cohesion are the key criteria which are optimized. Architects can prioritize coupling criteria and must provide machine readable software engineering artefacts.

   The authors report that one of the downsides of their approach is the time consuming creation of the machine readable inputs in JSON format, from for example use case diagrams [35].

   For the car retail system that is a major problem. First of all, there is no central repository of use cases. Even if there were, the sheer number of use cases would result in prohibitively large effort. Additionally, there is no sensible prioritisation of coupling rules for this project.

   This approach can therefore be ruled out for the system.

2. „The ENTICE approach to decompose monolithic services into microservices“ by Kecskemeti, Marosi, and Kertesz [42]

   This approach is not a general purpose approach, but highly related to the problem domain of creating virtual machine images. The presented work in fact does not deal with the problem on how to split a monolithic service in a way that fulfils various desirable properties, but how to technically handle the explosion of required virtual machine images. Major considerations are how to efficiently compress the large set of duplicate data shared among common VMs [42].

   In the project more modern container based virtualisation solutions are used. Docker in particular solves the problem by using several layers to build images. Common base steps, like installing dependencies, result in one shared layer that only needs to be stored once.

   This approach is not applicable to the work at hand.

3. „Microservices Identification Through Interface Analysis“ by Baresi, Garriga, and De Renzis [11]

   Following an API specification the paper presents an approach that extracts semantic information from there. Endpoints using similar vocabulary or semantically similar are grouped together in a proposed microservice.

   The paper proposes a complex mathematical analysis, but an implementation is also provided and can be downloaded from Github. As input an OpenAPI specification is required. The car retail system currently uses Swagger. In order to get reliable documentation, including meaningful textual descriptions of all API operations in the required format, some manual effort is expected.

   An additional required input for the tool is a so called Schema, a computer readable description of various words and their relationships [11].

   The developed tool has at the point of writing just one contributor and 20 commits all from the end of 2016. The description states that the tool is in an early stage of development [32].

   Overall the approach sounds promising but further refinements are needed. The biggest problems are the need for a good machine readable Schema, basically describing a full human language. Not all endpoints in the car retail project are well documented. For a large industrial size project it is not enough refined yet.

4. „From Monolith to Microservices: A Dataflow-Driven Approach“ by Chen, Li, and Li [20]

   Transmitting large amounts of data between microservices affects the performance negatively. This and the observation that parts of an application, that share data often both semantically and from a standpoint of cohesion belong together, has lead to the presented approach.

The presented approach uses the dataflow in order to calculate a good modularization.

However, in the first step a detailed dataflow diagram is manually created by engineers. Based on this data the algorithm starts the calculations [20].

Creating such a detailed dataflow diagram with the given size of the car retailer management is infeasible.

5. „Microservice Ambients: An Architectural Meta-Modelling Approach for Microservice Granularity“ by Hassan, Ali, and Bahsoon [36]

   This paper does not directly help with deciding how to split a monolith into a set of microservices. Instead it proposes a way to model such a transformation [36]. This is not relevant for our work.

6. „Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems“ by Levcovitz, Terra, and Valente [48]

   The authors of this paper propose building a directed dependency graph that points from higher levels of the code (facades), through business functions down to the database tables that are used to store the data. It is expected that every database table can be mapped unambiguously to one subsystem. This is not the case for the car retail system. Following the approach API gateways are created for all facades. All edges in the graph, on a path from a facade to a database table, are manually evaluated and assigned to business rules. These business rules are then evaluated according to their precedence and if transactions are needed. If several actions have to be performed in the same transaction, they need to be in the same microservice. If a certain precedence is required the gateway can handle this.

   Overall this approach is not viable for the car retail system because neither is it true that every database table can be assigned to a subsystem, nor does a list of business rules exist for which such precedences or requirements for transactions are specified.

7. „Microservice based tool support for business process modelling“ by Alpers et al. [4]

   This paper takes also a look at splitting a service according to business capabilities or use cases. The paper is actually about how to leverage microservice architectures for business process management (BPM) software. This part is not directly relevant for the system.

8. „Requirements reconciliation for scalable and secure microservice (de)composition“ by Ahmadvand and Ibrahim [3]

   The authors state that decomposition decisions are normally made intuitively by developers or architects. This intuitive approach does not capture the needs of all stakeholders. Security and scalability should be first class citizens and included into the approach according to Ahmadvand and Ibrahim.

They propose a methodology to decompose a monolith into a microservice, starting from system requirements. It is expected that for every functional requirement there are load estimations and it is known on which other requirements it depends. Misuse cases, that describe what must be impossible for the system to be secure, are used to describe the security requirements.

Functional requirements with a high need of scalability are in a first step proposed as stand alone microservices. If the overhead of having dependencies in a separate service is too high, they are merged.

The approach is almost entirely focused on requirements engineering and was evaluated using a fictitious case study. It does not consider other important aspects like cost, maintainability and the composition of development teams [3].

Security and scalability requirements are required as inputs. These requirements are not fully available for the car retail system and could only be very rough estimations. Therefore this approach could not be used for the system.

9. „Microservices tenets: Agile approach to service development and deployment“ by Zimmermann [78]

   This paper gives a general overview of the topic of microservices and compares them to SOA. Overall properties, that microservice architects should have, are listed but no detailed approach on how to efficiently reach such an architecture starting from a monolithic architecture is given. As a general hint the usage of DDD and business driven development to conceptualize services is given [78].

10. *Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach* by Daya et al. [22]

    This source is actually a full book and specific to a proprietary cloud platform that is not used by the system. Therefore the source is not applicable.

11. „Designing a Smart City Internet of Things Platform with Microservice Architecture“ by Krylovskiy, Jahn, and Patti [46]

    The authors describe the problems of designing an Internet of Things (IoT) platform and propose a microservice architecture to accommodate the requirements. As a short answer to the central question on where to draw the borders between the individual services, which are part of the microservice architecture, the authors state that one should follow the business organizations. This is in contrast to the classical organization of software teams according to the layer they work on. Cross-functional teams which can fully „own“ one service are deemed optimal. One problem that is combatted with such cross-functional teams is logic which is spread out over all layers. In a cross-functional team, the effort to fix a problem in the right place is not really higher than fixing it at the working place. A bug found by a member of a pure frontend team would be much more difficult to fix in the database layer in the old style of organization.

As additional point Krylovskiy, Jahn, and Patti name that both governance and data management should be decentralized and mostly under control of the team owning a service. As long as the interface remains stable, a team has complete freedom to change and optimize their service, including the local persistence strategy.

Another point which is mentioned is evolutionary design. It is not easy to get the optimal architecture on the first try. A microservice architecture, however, allows to easily replace a single service [46].

The general idea of using business contexts and team borders to decide where the borders of the newly created microservices should be seems very sensible and is feasible for the car retailer management system.

12. „Microservice architectures for scalability, agility and reliability in e-commerce“ by Hasselbring and Steinacker [39]

    This paper also recommends using business considerations for the decision on where to split a monolith. A proper vertical decomposition, according to Hasselbring and Steinacker, can efficiently be reached when splitting along the borders of business services. Following this approach one can also expect good understandability and maintainability of the resulting systems, as their content logically belongs together as it is part of the same business service.

    In the presented case study the authors mention making use of Conway's law and using multiple teams each working on one or more services. In the end they created several verticals, each responsible for one bounded context of the business domain.

    The approach is evaluated on a large scale software migration for *otto.de*. The results are very positive. Scalability has improved a lot and the number of deployments has improved dramatically, while the number of incidents did not increase [39].

13. „Towards an understanding of microservices“ by Shadija, Rezai, and Hill [61]

    This paper, which describes microservices in general, states that they implement a set of business functions. These business capabilities are more important than the source code to decide on a proper decomposition.

    Microservices are also described as business processes implementing business functionalities.

    Teams are often responsible for a business area. This aligns well with that idea that one or more microservices showed be owned by the team and other business areas, for which are other teams responsible, should be different microservices.

14. „Ensuring and Assessing Architecture Conformance to Microservice Decomposition Patterns“ by Zdun, Navarro, and Leymann [77]

    Zdun, Navarro, and Leymann assess the conformance of a given microservice architecture to common patterns that are deemed proper for microservices. An example of such a pattern is *Database per Service*, stating that every service should

have its own database and not use a shared one together with other services. The rationale is that a shared database limits scalability and allows unwanted coupling.

Besides that a microservice architecture should follow the best practices of said architecture, there is no approach described how to decompose a monolithic service [77]. Therefore this paper is not applicable to the problem.

15. *Building Microservices* by Newman [55]

    Newman wrote a book about building microservices. Bounded contexts are described in great detail. For an online retailer these contexts may include the warehouse and the accounting department. Both care about stock items but about very different things about them. For the warehouse it is for example important where the products are stored. The accounting department cares about the price. There should be local and shared models. Exposing all data quickly leads to tight coupling, which is unwanted.

    Newman recommends using the boundaries between bounded contexts also as boundaries between microservices. Only shared models should be used to communicate with other services. Internal data, hidden as this, preserves freedom to change without the fear of breaking other services. Conway's law is also cited to support this approach.

    Coarse grained bounded contexts can often be refined to even finer grained ones. Newman recommends sticking to the more monolithic side first and decomposing later as required, as more information is available. This is in line with the *Monolith first* microservice development pattern. Getting the boundaries right is easier once the bounded context is properly understood. This is often not the case for a green field development project. Therefore it may be beneficial to develop a monolith first and later decompose it into a microservice architecture once the domain and interactions are properly understood, compared to developing a new microservice architecture in the first place and taking a huge risk of getting the boundaries wrong [55].

16. „Microservices-based software architecture and approaches“ by Bakshi [7]

    The authors state that microservices should be built around business capabilities by cross functional teams. Data and logic should be decentralized following the concept of bounded contexts [7]

17. „Architectural Patterns for Microservices: A Systematic Mapping Study“ by Taibi, Lenarduzzi, and Pahl [68]

    The goal of this paper is to analyse different patterns employed by microservice architectures. It does not make recommendations on how to decompose a monolith [68]. Therefore it is not applicable to this work.

18. „Architecting microservices“ by Di Francesco [23]

This paper mostly focusses on how to model microservice architectures. Model driven engineering is cited and positive effects are cited [23]. There are no statements on how to decompose monoliths into a microservice architecture, however, thus making the paper not applicable to the question.

19. „Architecture of an interoperable IoT platform based on microservices“ by Vresk and Cavrak [75]

    Vresk and Cavrak describe the problems inherent to IoT and how microservices can solve some of them. There is no focus on how to decompose monoliths into microservice architectures, but the authors state that microservices may be responsible for single use cases or handling actions common for an entity, like customer management [75]. The paper is not directly applicable to the work.

20. *Microservices From Day One* by Carneiro and Schmelmer [17]

    Carneiro and Schmelmer wrote a book about microservices. Great emphasis is put on the importance of teams being fully responsible for the services they wrote, but having a lot of flexibility on how it is implemented.

    Boundaries between modules with well defined interfaces are seen as an important part of microservices. Teams are responsible for a subsystem which is implemented as one or more microservices. Microservices are split across business boundaries.

    While the decoupling of subsystems is possible in a monolith, it is very easy to violate such architecture decisions and in practice the architecture is rarely upheld. Microservices enforce staying true to the architecture more vigorously, which is seen as a big advantage.

    Carneiro and Schmelmer cite different possible boundaries between microservices, one of them is functionality. Different business functions or areas are put in different microservices. This aligns well with the business organizations within a company.

    When there are different sets of business functionalities with various maturity, separating them in their own services is a good idea. One example are social media sites, introducing a new comment system. It can be expected that it will change a lot during the early stages of adaption and feedback. Having a separate service provides great flexibility, including the possibility to completely rewrite or remove parts while keeping the old established parts of the system stable.

    Analysing storage needs for functionalities can also be used to decide on service boundaries. Some actions may require very strict transaction logic and are well suited to a relational database. Others are better suited with more performant key-value stores.

    Separate services that focus on the same entity make sense if they have different semantics in different bounded contexts. Again this supports the idea of using borders between bounded contexts as service boundaries.

All proposed boundaries follow business considerations. Separate groups of functionalities are split up and handled by individual microservices. These group boundaries can be defined by maturity, storage requirements or bounded contexts [17].

21. „Extraction of Microservices from Monolithic Software Architectures“ by Mazlami, Cito, and Leitner [51]

    The authors provide a formal approach to decompose a monolith into a microservice architecture without the need of much user input.

    The change history of the source code repository is analysed and a graph is constructed including the information when, by whom and what files or classes were changed. Based on that information, the coupling for every two distinct classes is calculated. A clustering algorithm is then applied to the graph resulting in microservice recommendations. Every cluster corresponds to one proposed microservice.

    Different clustering strategies are proposed.

    One idea is to cluster files that frequently change together. Software components should have one responsibility and change only for one reason. It makes sense to group classes that change for the same reasons.

    The bounded context concept is also mentioned favourably. A proposed solution to formalize that concept is to extract the semantic meaning from identifiers and comments in the source code. Statistic analyses which words commonly occur together in a language and allow to calculate the semantic similarities between classes.

    Finding teams that are responsible for parts of a monolith can be achieved by clustering on the contributors [51].

    This approach is highly dependent on the clustering strategy. The presented strategies extract the information which classes change for the same reasons, belong to the same bounded context or are in the same team responsibilities. This is, however, already known by the technical architects and team leaders at the system. Since no automatic actions are taken by the implemented prototype and it only produces recommendations, the value for the system is low. A big bang migration is not feasible and technical architects in each team can evaluate the quality of the current monolithic architecture and thus identify pieces of code which are in the „wrong“ class.

    Classes or files are also treated as atomic units. This is not particularly well suited to the system as there are quite many large classes that handle more different use cases than they probably should.

    While the approach is generally promising there only exists a prototype implementation. The data which should be produced is to a large extend already known as the clustering is based on bounded contexts and organizational boundaries.

22. „GranMicro: A Black-Box Based Approach for Optimizing Microservices Based Applications“ by Mustafa et al. [53]

    The authors state that a service should encompass a set of business processes and that DDD should be the starting point for a decomposition. DDD alone will yield a good basic microservice architecture, but not an optimized one. Their research focusses on the question of the granularity, how many of these business processes should be handled by one single microservice.

    Their tool proposes a service model diagram based on web server access logs.

    Because for the system the migration towards a microservice architecture is just beginning, such optimizations are not yet relevant, but may become important in the future. There is no implementation publicly available at the point of writing and the approach has not been tested on large web applications. Therefore it is not directly applicable to the current state of the system.

23. „Unsupervised learning approach for web application auto-decomposition into microservices“ by Abdullah, Iqbal, and Erradi [2]

    Abdullah, Iqbal, and Erradi describe problems with manually decomposing monoliths into microservice architectures. Scalability and performance are not considered as necessary by manual approaches.

    They propose using machine learning and the web server access logs in order to automatically extract microservices from a monolith in a way that optimizes scalability and performance. Their work goes further as they intend to automatically deploy the microservices to appropriate cloud instances and scale them appropriately.

    The authors only propose such a system and evaluate it on a benchmarking application. There is no tool publicly available that can be applied to arbitrary monoliths [2].

    Since the approach is only evaluated using one benchmarking application, the theoretical applicability to the system at hand is not clear. Because there is no tool available that performs the described steps, the paper is not directly applicable to the system at hand.

24. „A Secure Microservice Framework for IoT“ by Lu et al. [50]

    The authors apply a microservice approach to IoT. Devices can be seen as single services, but there are also major differences.

    The paper does not present an approach on how to decompose a monolithic architecture to a microservice architecture and is therefore not applicable to the question.

25. „Using Microservices for Legacy Software Modernization“ by Knoche and Hasselbring [44]

    Knoche and Hasselbring report of the migration of an insurance company's large business system. The old system was written in Cobol. As a first step they

decomposed the Cobol code and only afterwards migrated to Java, since the language migration is considered a very high technical risk.

They describe employing an external service facade to hide the concrete implementation. This is important, as in their case internal methodologies and even database tables were directly accessed by other programs. The facade is designed domain-oriented from a domain model and is grouped by domain contexts. Clients were migrated to exclusively use the facades for interacting with the system. Afterwards the process of gradually replacing the implementation of the facades with microservices begun. The services are split across the boundaries defined in the facade which in turn is defined based on domain-oriented design decisions. The facades serve as well defined interfaces.

8.3 Categorisation

In this section the previously presented approaches are listed in a table. The first column contains the number from above and the source. In the second column there is a judgement whether the approach could be used to decompose the system or not. Finally, in the third column a short reason for this judgement is given.

Number	Applicability	Reason
1 [35]	Not applicable	Approach requires detailed machine readable use case diagrams, which are not available.
2 [42]	Not applicable	Approach does not deal with splitting a monolithic service into microservices.
3 [11]	Not applicable	The approach is not mature enough and requires too much manual intervention, as automating tools are not available.
4 [20]	Not applicable	Creating the required data-flow diagrams is performed by hand in this approach. This is infeasible for a project of the size of the system at hand.
5 [36]	Not applicable	The paper is not about an approach on how to decompose a monolith into a microservice architecture.
6 [48]	Not applicable	The papers assumption that every database table can unambiguously be assigned to exactly one subsystem is not the case for the system at hand. Additionally, a list of business rules specifying dependencies and transaction requirements, which is required by the approach, is not available.
7 [4]	Partially applicable	The focus of the paper is not on an approach but on how microservice architectures can be leveraged for BPM software. The idea to split along business capabilities or use cases is applicable.
8 [3]	Not applicable	The needed formalized security and scalability requirements are not available for the system at hand.

9 [78]	Partially applicable	A detailed approach is not presented, but using DDD and business driven development is mentioned favourably.
10 [22]	Not applicable	This source is a full book specific to a proprietary cloud platform which is not used by the system at hand.
11 [46]	Applicable	The approach recommends following business contexts and team boundaries. It has no specific requirements which are at odds with the system at hand or the criteria given in section 8.1 and fully describes the approach.
12 [39]	Applicable	This approach again focusses on business considerations and has not requirements which are infeasible for the system at hand. There is also an evaluation with positive results.
13 [61]	Applicable	Again business functionalities are at the core of what should become its own microservice following this approach. There are no requirements which can not be met by the system at hand.
14 [77]	Not applicable	This paper does not describe an approach on how to decompose an existing monolith.
15 [55]	Applicable	The boundaries between bounded business contexts are described as the ideal boundaries between microservices.
16 [7]	Applicable	ABusiness contexts are recommended as basis for service boundaries. There is no conflict with the system at hand.
17 [68]	Not applicable	This paper focuses on common patterns in microservices and not on how to decompose a monolith.
18 [23]	Not applicable	In this paper there are no instructions on how to decompose an existing monolith into a microservice architecture.
19 [75]	Not applicable	The paper is not about how to decompose monoliths.
20 [17]	Applicable	According to this paper the boundaries between microservices should be based on business contexts. Storage needs should also be taken into consideration.
21 [51]	Not applicable	This is a formal approach which uses different clustering strategies calculates a good decomposition. The tooling support is not mature enough yet.
22 [53]	Partially applicable	Business processes and DDD should be the basis for the boundaries between the services according to this paper. They also propose a tool parsing web server access logs. This is not well established and not production ready though.
23 [2]	Not applicable	The authors propose a machine learning based approach which parses web server access logs. Since such a tool is not publicly available at the point of writing it can not be applied to the system at hand.
24 [50]	Not applicable	The paper does not present an approach to decompose a monolith.

25 [44]	Applicable	The authors propose introducing service facades. This is possible for the system at hand.

This section also provides a categorisation of the mentioned approaches, according to their applicability to large industrial systems, the system at hand and their commonalities. In parentheses the numbers of the approaches in the table above are given.

Fully applicable:
To conclude Krylovskiy, Jahn, and Patti (11), Hasselbring and Steinacker (12), Shadija, Rezai, and Hill (13), Newman (15) and Bakshi (16) recommend using business capabilities and organizational boundaries as inspiration for where the monolith should be split into microservices. They describe boundaries between bounded contexts as excellent choices for microservice boundaries because the desired tight coupling inside a proper bounded context and thus a microservice is implied [7, 39, 46, 55, 61]. Carneiro and Schmelmer (20) also recommend microservice boundaries between bounded contexts and additionally more fine grained boundaries between different storage and persistence requirements and maturities [17]. Mustafa et al. (22) recommend DDD in order to get a basic microservice architecture and to analyse usage patterns to further refine the architecture afterwards [53]. While the focus of Alpers et al. (7) and Zimmermann (9) is not about the decomposition, the papers shortly mention the same approach favourably [4, 78].

Knoche and Hasselbring (25) recommend employing service facades as a first step to recover the architecture within a monolith. The decisions in which parts to cut follows DDD [44].

So far these approaches are actually encompassed by the approach which is currently used to decompose the system at hand.

Too formal:
The approaches of Gysel et al. (1), Chen, Li, and Li (4), Levcovitz, Terra, and Valente (6), Ahmadvand and Ibrahim (8) require some sort of formal specification, like a machine readable use case diagram or extensive manual input, like manually created data flow diagrams or formal transaction requirements. [3, 20, 35, 48]. For the system at hand such formal specifications do not exist. Neither is it feasible to manually create the required precise inputs. According to Ozkaya formal models and specifications are rarely employed in practice by large industrial projects [57]. Therefore they are not applicable for the decomposition of most large industrial software systems.

Suboptimal tool support:
The approaches of Baresi, Garriga, and De Renzis (3), Mazlami, Cito, and Leitner (21), Abdullah, Iqbal, and Erradi (23) provide mathematical models as well as tools that at least theoretically allow application to large software systems. Without the tools the mathematical analysis become too expensive to perform manually for any sizeable software system. Unfortunately, these tools are not mature enough yet or even not

available at all for the general use [2, 11, 51]. For this reason their application to the system at hand as well as any other large industrial monolith is not feasible.

Off target:
Kecskemeti, Marosi, and Kertesz (2), Hassan, Ali, and Bahsoon (5), Daya et al. (10), Zdun, Navarro, and Leymann (14), Taibi, Lenarduzzi, and Pahl (17), Di Francesco (18), Vresk and Cavrak (19) and Lu et al. (24) do not actually describe an approach to decompose an industrial monolith into a microservice architecture [22, 23, 36, 42, 50, 68, 75, 77]. Therefore these papers are neither applicable to the system at hand nor to determine an approach to decompose any other large industrial monolith.

8.4 Summary of literature research

In the author's point of view it would be straight forward to choose an approach, given the overwhelming number of papers referring bounded contexts, business contexts or DDD. There is not a single applicable approach in the table above that contradicts that recommendation. When using this concepts, the boundaries between the newly created microservices are specified.

Paper number 25 ([44]) is the only differing approach which is still fully applicable to arbitrary monoliths. It deals with the more low level migration details, how to migrate on a technical level, while disrupting the continuous operation as little as possible. The proposed solution is the usage of facades. This can be nicely integrated with the bounded contexts, deciding where boundaries should be located.

This combination is actually encompassed by the approach which is used to decompose the system at hand and which is described in more detail in chapter 7.

The practitioners who made the decision on how to decompose the system at hand came to the same conclusion which approach should be used as an independently conducted systematic literature review.

All approaches which were ruled out as potential approaches for the system at hand can also be ruled out for most other industrial monoliths. Therefore and since there is overwhelming academic support in the literature for the chosen approach, the academic scalability is not threatened.

CHAPTER 9

Evaluation procedure

In order to evaluate the answers given to the research questions in chapter 3, experts on the system at hand and the modularization procedure are interviewed. This procedure follows the well established methodology of an expert interview described in more detail in chapter 4.

9.1 Expert interviews

As stated in the introduction to this chapter, the expert interviews are a major part of the evaluation procedure.

Suitable experts must be familiar with the system at hand in order to judge the modularization procedure, its results and provide insights exceeding the currently available academic literature. The potential interviewees are therefore only the colleagues of the authors. They are familiar with the system at hand and the progress of the modularization. All of them have multiple years of software engineering experience in the industry and are therefore qualified to share the technical and the domain specific insights. No one else is part of the modularization effort. Therefore other people do not have hands on experience of the modularization process of the system at hand and are not considered relevant for an interview.

The expert interviews are intended to evaluate the chosen approach, applied on the system at hand as well as to ensure the judgements of the alternative approaches are justified. The answers given by the interviewees indirectly answer some of the research questions. Chapter 10 then compiles the given answers into more detailed and interpretative answers to the research questions.

Compared to only the author answering the research questions, the bias is reduced by questioning multiple colleagues. To reduce bias as much as possible the questions are designed to not lead in any way and feedback is obtained.

In order to be able to better compare the opinions of the experts, the interview format is fully structured in the form of a written standardized interview.

Unfortunately, quantitative analyses unfortunately are very limited as the number of possible participants is rather low.

9.2 Questions

The set of questions asked is as follows:

1. What is your role in the project? (junior, senior developer, architect, tester, pm, ...)
2. For how long have you been part of the project? (months, years)
3. For how long have you been a professional in your field? (years)
4. How content are you with focussing on business capabilities and bounded business contexts as basis for choosing the boundaries between microservices? Are there any problems with this approach?
5. Would a more detailed, possibly formal approach, giving low level instructions on a class or method level, be helpful?
6. What is the outcome from using facades during the modularization?
7. Does the modularization impede other teams? If so, how are they affected?
8. How well can multiple teams modularize different parts of the monolith at the same time?
9. Does the modularization impede the continuous operation of the system? If so, how?
10. Is the maintainability changing? If so, how?
11. How does the maintainability differ in the already modularized parts, compared to the remaining monolith?
12. Are there any changes to the maintainability of the remaining monolith? If so, what are they?
13. How does development of parts of the remaining monolith differ from developing a part of a new microservice?
14. Which type of development (microservice / monolith) do you prefer in the context of this system and why?

15. Is the scalability changing? If so, how?
16. How does the scalability differ in the already modularized parts compared to the remaining monolith?
17. Are there any changes to the scalability of the remaining monolith? If so, what are they?
18. Are there impediments stopping the ability to scale any of the services as desired?
19. What are the main obstacles the modularization approach currently faces?
20. Is there anything else you would like to add about the modularization or the project as a whole?

CHAPTER 10

Results

The verbatim answers given by the experts are included in the appendix.

To explicitly answer the first research question (**RQ1**): There are a wide variety of results achieved by following the decomposition approach. This section presents general findings not directly related to maintainability or scalability as they are presented in their own sections.

The experts' answers confirm that the chosen approach is indeed well suitable to improve the situation.

Nevertheless, there are problems with the implementation of it. Non-technical reasons, like a very limited budget and high pressure to release new features quickly, cause the most problems. Since only limited resources are used for the modularization effort, the progress is slow.

Three developers, a technical architect and a software tester were familiar enough with the modularization effort and thus able to complete the interview. Their experience with the project ranges from one to two years and their overall experience as professionals in the software industry from five to ten years.

The experts are content with focussing on business capabilities and bounded contexts as a basis for the modularization. The requirement for good Technical architects (TAs) and Product owners (POs) is stressed and that the technical contexts are also important. Tight coupling of different business contexts makes purely focusing on the business contexts inefficient. Some technical contexts and cross cutting concerns are not bound to a specific business context and still have to be addressed. Following this approach interviewee E mentions that problems may arise early if the monolith is not already internally modularized.

The experts consider more formal approaches infeasible for a project of the given size and budget. Creating formal models and specifications is estimated to be prohibitively expensive. While some help is considered helpful it should not be too restrictive to the

developers. Another two interviewed developers state that the overhead would be simply too high for a sizeable project. Since there is more complexity involved the result is not expected to be any better by interviewee E.

Generally they judge the usage of facades in a positive way, mention that some issues were encountered. A previous modularization effort also used facades. Back then a single team was tasked with creating new facades and migrating code to use these new facades. This team was not aware of the business contexts and therefore making progress without breaking prohibitively, much existing functionality was infeasible. Unfortunately, the quality of the automated tests is not enough to allow such a refactoring. Since the signatures were changed, a lot of test code got obsolete or had to be adapted which is especially hard without domain knowledge. Nevertheless, a lot of tests broke and needed to be fixed in a time consuming manner.

Another previously started refactoring introduced UUIDs as identifiers instead of numeric database primary keys. Unfortunately, this was not done consistently in the whole project. For compatibility reasons it is often necessary to convert between a numeric identifier and the new UUID. This conversions can lead to dramatic performance penalties if done inefficiently like one lookup for every element in a collection. It is possible to query for a collection of one type of identifiers, however, often it is not immediately apparent that one iterates over many elements, when multiple method calls lie between the iteration and the converting database or REST call.

There is some confusion between services and facades and when to call which from a given context and subtle differences in method parameters.

Some facades differ from corresponding services in their type of identifier and when migrating such a facade to use REST calls, inefficiencies become more visible and pressing compared to pure database queries.

In the new approach the signatures of facades stay the same to the old implementation and a compatible implementation is provided ,which delegates to the REST server.

Interviewee E states that facades are a well known and good strategy which allows to switch between multiple implementations easily and quickly.

There is no absolute consensus on how large the impediments are for other teams. Interviewee A, for example, believes the impact is not that large, while Interviewee D mentions introduced bugs which were not caught by insufficient automatic test quality, which is backed up by interviewee B. Additionally, it becomes harder for other teams to quickly change code, when it is split across multiple applications as the API have to adapted additionally to the applications themselves. Further, the modularization itself sometimes forces other teams to wait for not yet fully implemented new features of the modularization taskforce or use the legacy way, knowing that they have to change it soon after. According to interviewee E the state of the internal modularization of the monolith makes a crucial difference. If the monolith is already properly used, for example Maven modules, it becomes easier to use extract services along these borders which hopefully

coincide with the bounded context boundaries. When there are little to no such internal structures, it becomes harder to extract microservices and also for different teams to coordinate the extraction effort.

The modularization effort should be prioritized across teams, to limit the negative impact but also to achieve maximal long term positive effects. Interviewee E states that for that cross cutting issues should be a priority.

Another important point mentioned by Interviewee C is that all affected teams have to fully test the application, which puts additional stress on the testers.

The experts state that scaling the modularization effort up to multiple teams is easily possible, as long as clear lines between different areas exist. During the modularization a lot of files are changed and this leads to hard to solve merge conflicts, if multiple persons are active on the same part of the source code. This not only applies to other modularizing teams but active functional development as well.

Even tough bugs are regularly introduced by the modularization effort, the overall stability and availability of the software is not really affected. Manual tests across several quality assurance stages normally catch introduced bugs very quickly. Before a major release no critical changes are pushed on the master branch, in order to give testers a good chance to ensure the overall quality. If serious issues are found and can not be fixed before the release date, it is delayed. Operational requirements should be taken into consideration according to interviewee E, in order to prevent issues in this regard.

Recently new feature flags have been introduced to allow to quickly enable and roll back changes to some modularized configuration changes.

10.1 Maintainability

A major reason for the whole effort is to improve maintainability. Therefore it is crucial that improvements are achieved in this regard.

RQ1a asks how the maintainability is affected. There is a general consensus that the situation improves a lot, but for the system at hand there is still a lot to do.

For the new modules the feedback of the experts is very promising. Smaller modules are easier to understand, start and debug. Testing a smaller application also becomes easier and more efficient.

As testers primarily use quality assurance deployments and do not need to work with local code, deployment speed is an important part of the latency with which a tester can start to examine changes, which also improves maintainability. The newer smaller modules can be deployed much faster than the remaining monolith.

This is also the case for releases, as interviewee D mentions.

A negative aspect mentioned is that some actions, like version updates, have to be repeated for many applications instead of only one monolith. This is not hard but cumbersome. Maintenance is greatly reduced per module but is required for more modules.

Developers understand smaller modules more easily and thus new developers get productive faster. This in turn improves the maintainability of these parts.

When the boundaries of a module are left, debugging becomes harder compared to a monolith, where it is possible to debug across the whole application inside an IDE.

Also when developers change the API between modules, the effort is generally larger as in a monolith, where such an API does not formally exist. As long as changes are internal to an application, the maintainability improves but as soon as multiple applications are affected the advantages diminish and new problems arise. For backwards compatible changes a new API version has to be released in addition to the code changes. Other applications have no stress to upgrade. If a change is not backwards compatible, however, all applications depending on the functionality have to upgrade at the same time or API versions have to be used. The effort is multiplied by the number of usages. For this reason it is crucial to keep the number of API changes minimal. Interviewee E also stresses this as a crucial point and favourably mentions stable contracts between services.

The maintainability of the remaining monolith becomes slightly better as its size decreases, as reported by interviewee A.

In the project some migration efforts have been started and abandoned half way. The mentioned move from numeric identifiers to UUIDs is such an example. A half finished modularization approach can make it temporarily harder to work with a system as the new way of doing things is not fully introduced and the old one still exists. As long as the old variant is not removed and still has to be maintained, the effort in fact increases. Such situations are very displeasing and should be overcome as soon as possible, allowing to remove the old code and leverage the benefits of a smaller monolith.

Integration tests, that test the interactions between services, are important and domain knowledge over several services help here. Interviewee E predicts problems if external services are mocked in all tests.

Similarly, there is the danger of no longer knowing enough about the surrounding systems and thus having knowledge concentrated in single teams. This runs contrary to the idea of the desired cross functional teams.

10.2 Scalability

RQ1b is about how the scalability is affected. Scalability is also a widely given reason for a microservice architecture. In the project mixed results are reported. Theoretically, once the modularization is finished it should allow to scale all applications very well, according to interviewee D, but it is not experienced in the project yet as the system overall is perceived as rather slow, regardless of the load and more replicas do not help directly

with latency problems. If parts of the system are identified as under particular high stress, it is advantageous if this is a microservice, as administrators can independently scale it up.

Interviewee C does not see any changes to the scalability so far and interviewee B cites the remaining shared database as limiting factor for scalability, but that the situation has already improved with monolithic applications.

Interviewee A also addresses the aspect of scaling to more teams and favourably mentions that many teams can more easily create their own services in a microservice architecture, without negatively impacting other teams. Every team can own a set of services and interact via well defined APIs.

There were efforts to make the remaining monolith less stateful, to more easily allow running replications but this has not progressed very far. Currently, it is not possible to just load balance any request to any instance of the remaining monolith. A user's request in a session always has to be sent to the same server as the user is authenticated there and data is kept in the server side session. This makes load balancing more complex and limits the scalability. It is not directly possible to move some users of an already overloaded instance.

While the scalability of a single service in isolation becomes easier, the whole orchestration becomes more complicated. Once this is properly taken care of the benefits of being able to scale every service according to its needs becomes prevalent. Container platforms and other operational details are therefore crucial for the success of the scalability improvements.

The system at hand repeatedly suffers from network issues with the used OpenShift container orchestration platform. Updates of this core system component also lead to downtime already.

The available hardware and budget also limits the scalability.

10.3 Satisfaction

Since the codebase has grown a lot over several years with around 100 developers contributing, the modularization approach is described as taxing by interviewee A. A restrained budget and poor documentation are also reasons any major changes, like the modularization, are hard. Interviewee B criticises working on too many things at once instead of completely finishing an effort and then moving on. The half finished migration to UUIDs is an example.

For budget reasons there is currently only one team performing directly the modularization. While multiple potential new microservices were identified, at the point of writing only the extraction of the configuration microservice is actively in progress. Several other microservices exist which have already been extracted and still others were written from scratch.

The states in between starting the extraction of a microservice and the final code clean-up are very undesirable as temporarily more code has to be maintained. Therefore all approaches from this category can not be considered viable alternatives for now.

Developers perceive working with the newer, smaller services very positively. The main reasons are easier understandability, greatly improved development speed and faster feedback cycles. These qualities have a large positive impact on maintainability. Interviewee E even calls the split inevitable but warns of overdoing it and creating too much tiny services. Those are less maintainable when any reasonable work spans multiple services and the API has to be changed all the time.

Working with a smaller service also allows to upgrade the code more easily as fewer dependencies have to be taken into consideration. This leads in the long term to a more modern application with which developers are more satisfied and prefer working with.

From a tester's perspective the microservice approach is also preferable, as a microservice can be tested more easily. Test automation is simpler as services can be tested in a more isolated way than the monolith.

Interviewee D is not always happy with the size of the microservices and criticizes the additional overhead of having multiple services. This can lead to a slowdown due to excessive splitting and the previously mentioned additional work required if APIs change. Smaller services, but not tiny services, are therefore preferred by interviewee D.

10.4 Alternative approaches

The second research question (**RQ2**) considers alternative approaches.

In chapter 8 those are analysed. Overall those approaches that are theoretically applicable to the system at hand were already applied.

A few other approaches are too formal and require precise inputs which are neither for the system at hand nor for most other large industrial monoliths available [57]. The interviewed experts also agree that such approaches are not good candidates for large industrial systems. Some even doubt that the results of such approaches were better if their required inputs were available.

There also exist papers of approaches which include tools to be executed, in order to compute good decompositions. None of the examined approaches in this category provided a fully functional tool which is considered by the authors of the tools as production ready. Therefore all approaches from this category can not be considered viable alternatives for now.

Overall, the already used approach of using bounded business contexts to determine the boundaries between the services and using facades, in order to quickly exchange the implementation from in memory calls to REST calls can, therefore be considered as the only viable approach, for systems for which no formal descriptions are available.

CHAPTER 11

Future work

Since the decomposition of the system at hand is not completed at the point of writing and it is expected that this will take a long time, other analysis, once they are done, could provide further insights. Long term effects of the modularization could be observed and the scalability improvements better evaluated, when the system is deployed with a much larger user base and thus higher scalability demands.

More metric based approaches could be used to evaluate the achieved improvements. This potentially includes searching for code smells in the monolith and the created microservices over time. Generally for this a metric that can accurately describe the differences in maintainability in a monolith and a microservice needs to be found. Load tests can be performed during various stages of the development though this would be complicated by the parallel surge of features which causes a slowdown and the modularization which is intended to improve the scalability.

Future work may include performing more case studies, further evaluating the chosen approach for other large scale industrial systems and combining the knowledge in larger meta analyses that generate results with statistical significance.

Since some alternative approaches found in the literature lack good tool support, their widespread application to large systems is prevented. Fully developing these tools, integrating them with practitioners tools like IDEs and performing case studies to evaluate their practical applicability could be further studies.

CHAPTER 12

Threats to validity

For case studies it is always crucial to ensure the academic scalability. The results should not be specific to the system at hand but have to be generally applicable.

The literature review and the judgement of alternative approaches comes to the same result if conducted independently from the system at hand. Any approaches that are not deemed applicable as potential alternative to the current approach are also not applicable to most other large scale industrial monoliths. This crucial aspect is described in greater detail in section 8.4.

Other approaches on more formal methods are deemed unsatisfactory by the experts and Ozkaya [57].

The expert interviews are structured in such a way that questions are not specific to the system at hand. Answers are also given in a form which is independent from the system at hand.

A threat to validity is that since the author is one of several developers of the system, there may be a bias about how useful the performed work is. In order to minimize this threat feedback for the whole process, methodology is used to improve it and expert interviews are conducted to evaluate the results. In order to prevent biased questions or at least limit the impact, feedback on the questions is gathered before the interviews and on the whole interview process afterwards.

Runeson et al. lay out four different types of validity. The first one is *Construct Validity*. A problem in this area arises if the applied measurements do not really measure what the author of the study intends. This may be the case if an interview partner interprets questions or answers differently from the interviewer. All answers given by the interviewees in this case study were quite clear and further enquiries were not necessary. Nevertheless, feedback is gathered in order to gain a third perspective and discover potential problems [60].

The second aspect is *Internal Validity*. This deals with the question of whether causal relations are correctly inferred or could be explained by others, perhaps not considered variables. Since we do not seek causal relations this is not concerning [60].

As third aspect Runeson et al. name *External Validity*, which deals with whether the findings can be generalized and are of relevance to other people. Šūpulniece et al. state that there is not a lot of research conducted on large industrial cases [67]. This means that there is need for more case studies and thus interest in the current work. Statistically significant results can not be concluded from this case study alone [60]. As stated previously in this chapter, the study can be generalised to other systems as the literature research would have reached the same conclusion for other typical monoliths. The system at hand is considering its relevant properties in no way special for the industry as a whole. The expert interviews can also be applied or generalised for a wide range of industrial monoliths.

Reliability is the final aspect which means the ability to replicate the study by other researchers [60]. In order to enable this the thesis will contain all taken steps. Since the source code is proprietary and actively developed the exact same setup can never be achieved again. Nevertheless, the same approach can be applied to other projects.

CHAPTER 13

Conclusion

Microservices are an emerging software architecture. Relatively small services interact with each other in order to achieve the whole functionality, while staying as independent from each other as possible in order to scale and develop each service on its own. This architecture lends itself well to writing large distributed systems.

A monolith has a very different architecture. A lot of unrelated functionalities are present in one executable. Once this executable reaches a certain size, both maintainability and scalability may become problematic.

Historically, a large ERP in the automotive industry was developed as monolith. It suffered from poor maintainability and scalability. In order to improve the situation, management made the decision to migrate to a microservice architecture.

The chosen approach is to use bounded business contexts, in order to determine what should become a microservice. Developers then either rewrite relevant parts of the system as a microservice or extract them from the existing monolith. In order to minimize the disruption and allow gradual changes, they use facades to switch between different implementations of the same functionality as desired.

This work analyses the achievements of this approach, in particular changes to the maintainability and scalability of the system at hand and, following a case study methodology, ascertains general conclusions for any large industrial system decomposition.

Alternative approaches are evaluated in the form of a literature review. From this we conclude that the chosen approach indeed follows recommended academic practices and that there are currently no alternative and practically viable approaches available.

During the application of the current approach, changes to the development system are measured. For example the latency between a version control system (VCS) commit and feedback from a CI server is greatly reduced as well as the local build times. The number of lines of code and commits per time in the smaller source control management (SCM) repositories, compared to the monolithic one, increases steadily.

Developers started to move out pieces of the monolith into microservices while also rewriting some parts in the form of a completely new microservice. When a new microservice is created, which replaces old functionality, both implementations have to be maintained until the migration is complete. Therefore it is desirable that every migration finishes soon.

Performing changes to a microservice is faster compared to a monolith. When working with the monolith, the IDE often lags, startup times are high and newer developers need to understand a lot about the architecture in order to be productive. The newer microservices in contrast are easier to understand, state of the art technologies are used and developers get faster feedback from local tools as well as the CI server.

To evaluate the results, we conduct expert interviews of stakeholders, involved in the development of the system. They are familiar with the modularization effort.

They agree that maintainability for the already modularized parts has greatly improved. Drastic changes in scalability are currently not visible, but they agree that in general the smaller services are much more scalable.

The developers greatly prefer working on smaller services and are content with the applied approach. More formal approaches are largely disliked by the interviewed experts. They are deemed hardly possible to apply for such a large project, too restrictive and it is doubted if the results would be better.

Overall, the migration and the chosen approach can be considered a success, even tough the migration is still ongoing and a lot of work still has to be invested in order to completely migrate to a microservice architecture. Improvements are already visible and greatly appreciated by developers and testers.

List of Figures

Acronyms

ACID Atomicity, Consistency, Isolation, Durability.

API Application programming interface.

AWS Amazon Web Services.

BC Bounded Context.

BPM business process management.

CD Continuous Delivery.

CI Continuous Integration.

CPU Central Processing Unit.

CRUD create, read, update and delete.

DDD Domain-Driven Design.

DRY do not repeat yourself.

DTO Data Transfer Object.

E2E End-to-End.

ERP Enterprise Resource Planning.

ESB Enterprise Service Bus.

GUI Graphical User Interface.

HATEOAS Hypertext As The Engine Of Application State.

HTTP Hypertext Transfer Protocol.

IaaS Infrastructure as a Service.

ID Identifier.

IDE Integrated development environment.

IoT Internet of Things.

JSON JavaScript Object Notation.

JTA Java Transaction API.

JWT JSON Web Token.

OOP Object Oriented Programming.

PDTO Public Data Transfer Object.

PO Product owner.

RAM Random Access Memory.

REST Representational State Transfer.

RPC Remote Procedure Call.

SCM source control management.

SLA Service Level Agreement.

SOA Service Oriented Architecture.

SOAP Simple Object Access Protocol.

SVN Subversion.

TA Technical architect.

UDDI Universal Description, Discovery, and Integration.

UI User Interface.

UML Unified Markup Language.

URL Uniform Resource Locator.

UUID Universally unique identifier.

VAT value-added tax.

VCS version control system.

VM Virtual machine.

WSDL Web Services Description Language.

XML Extensible Markup Language.

Bibliography

[1] Muhammad Abbas et al. „Scrum Software Maintenance Model: Efficient Software Maintenance in Agile Methodology“. In: *2018 21st Saudi Computer Society National Computer Conference (NCC)* (2018), pp. 1–5. DOI: 10.1109/ncg.2018.8593152.

[2] Muhammad Abdullah, Waheed Iqbal, and Abdelkarim Erradi. „Unsupervised learning approach for web application auto-decomposition into microservices“. In: *Journal of Systems and Software* 151.February (2019), pp. 243–257. ISSN: 01641212. DOI: 10.1016/j.jss.2019.02.031. URL: https://www.sciencedirect.com/science/article/pii/S0164121219300408.

[3] Mohsen Ahmadvand and Amjad Ibrahim. „Requirements reconciliation for scalable and secure microservice (de)composition“. In: *Proceedings - 2016 IEEE 24th International Requirements Engineering Conference Workshops, REW 2016* (2017), pp. 68–73. DOI: 10.1109/REW.2016.14.

[4] Sascha Alpers et al. „Microservice based tool support for business process modelling“. In: *Proceedings of the 2015 IEEE 19th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations, EDOCW 2015* January 2017 (2015), pp. 71–78. ISSN: 2325-6583. DOI: 10.1109/EDOCW.2015.32.

[5] Doug Arcuri. *Observations on the testing culture of Test Driven Development.* 2018. URL: https://www.freecodecamp.org/news/8-observations-on-test-driven-development-a9b5144f868/ (visited on 06/01/2019).

[6] Ali Arsanjani. „Service-oriented modeling and architecture“. In: *IBM developer works* January (2004), pp. 1–15. DOI: 10.1109/SCC.2006.93.

[7] Kapil Bakshi. „Microservices-based software architecture and approaches“. In: *IEEE Aerospace Conference Proceedings* (2017). ISSN: 1095323X. DOI: 10.1109/AERO.2017.7943959.

[8] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. „Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture“. In: *IEEE Software* 33.3 (2016), pp. 42–52. ISSN: 07407459. DOI: 10.1109/MS.2016.64. arXiv: 1606.04036.

[9] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. „Microservices Migration Patterns“. In: 1 (2015), pp. 1–21.

[10] Armin Balalaie et al. „Microservices migration patterns“. In: *Software - Practice and Experience* 48.11 (2018), pp. 2019–2042. ISSN: 1097024X. DOI: 10.1002/spe.2608.

[11] Luciano Baresi, Martin Garriga, and Alan De Renzis. „Microservices Identification Through Interface Analysis“. In: *Service-Oriented and Cloud Computing*. Ed. by Flavio De Paoli, Stefan Schulte, and Einar Broch Johnsen. Cham: Springer International Publishing, 2017, pp. 19–33. ISBN: 978-3-319-67262-5. DOI: 10.1007/978-3-662-44879-3. URL: https://link.springer.com/chapter/10.1007/978-3-319-67262-5{_}2.

[12] Paul Barham et al. „Xen and the Art of Virtualization“. In: *Proceedings of the nineteenth ACM symposium on Operating systems principles - SOSP '03* 37.5 (2003), p. 164. ISSN: 0163-5980. DOI: 10.1145/945445.945462. URL: http://dl.acm.org/citation.cfm?id=945445.945462.

[13] Bogdan. *Is the public UDDI movement dead or, was it ever alive?* 2012. URL: https://stackoverflow.com/questions/1525045/is-the-public-uddi-movement-dead-or-was-it-ever-alive (visited on 06/01/2019).

[14] Grunnar Brataas et al. „Scalability Analysis of Cloud Software Services“. In: *Proceedings - 2017 IEEE International Conference on Autonomic Computing, ICAC 2017* (2017), pp. 285–292. DOI: 10.1109/ICAC.2017.34.

[15] Ivan Candela et al. „Using Cohesion and Coupling for Software Remodularization“. In: *ACM Transactions on Software Engineering and Methodology* 25.3 (2016), pp. 1–28. ISSN: 1049331X. DOI: 10.1145/2928268.

[16] Javier Luis Cánovas Izquierdo and Jesús García Molina. „A domain specific language for extracting models in software modernization“. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5562 LNCS (2009), pp. 92–97. ISSN: 03029743. DOI: 10.1007/978-3-642-02674-4-7.

[17] Cloves Carneiro and Tim Schmelmer. *Microservices From Day One*. 2016. ISBN: 9781484219362. DOI: 10.1007/978-1-4842-1937-9.

[18] Luiz Carvalho et al. „Analysis of the Criteria Adopted in Industry to Extract Microservices“. In: *Proceedings - 2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry and 6th International Workshop on Software Engineering Research and Industrial Practice, CESSER-IP 2019* (2019), pp. 22–29. DOI: 10.1109/CESSER-IP.2019.00012.

[19] Lianping Chen. „Continuous delivery: Huge benefits, but challenges too“. In: *IEEE Software* 32.2 (2015), pp. 50–54. ISSN: 07407459. DOI: 10.1109/MS.2015.27. arXiv: arXiv:1011.1669v3.

[20] Rui Chen, Shanshan Li, and Zheng Li. „From Monolith to Microservices: A Dataflow-Driven Approach“. In: *Proceedings - Asia-Pacific Software Engineering Conference, APSEC* 2017-Decem (2018), pp. 466–475. ISSN: 15301362. DOI: 10.1109/APSEC.2017.53. URL: https://ieeexplore.ieee.org/abstract/document/8305969.

[21] Don Coleman et al. „Using Metrics to Evaluate Software System Maintainability“. In: *Computer* 27.8 (1994), pp. 44–49. ISSN: 00189162. DOI: 10.1109/2.303623.

[22] Shahir Daya et al. *Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach*. IBM Redbooks, 2016.

[23] Paolo Di Francesco. „Architecting microservices“. In: *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings* (2017), pp. 224–229. DOI: 10.1109/ICSAW.2017.65.

[24] Nicola Dragoni et al. „Microservices: yesterday, today, and tomorrow“. In: (2016), pp. 1–17. ISSN: 2327-4662. DOI: 10.13140/RG.2.1.3257.4961. arXiv: 1606.04036. URL: http://arxiv.org/abs/1606.04036.

[25] Roy Thomas Fielding. „Architectural Styles and the Design of Network-based Software Architectures“. In: (2000), pp. 76–106.

[26] Roy Thomas Fielding. *REST APIs must be hypertext-driven*. 2008. URL: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (visited on 02/27/2019).

[27] Martin Fowler. *BoundedContext*. 2014. URL: https://martinfowler.com/bliki/BoundedContext.html (visited on 06/04/2020).

[28] Martin Fowler. *Microservices*. 2014. URL: https://martinfowler.com/articles/microservices.html (visited on 02/26/2019).

[29] Martin Fowler. *Richardson Maturity Model*. 2010. URL: https://martinfowler.com/articles/richardsonMaturityModel.html (visited on 02/27/2019).

[30] Jonas Fritzsch et al. „From Monolith to Microservices: A Classification of Refactoring Approaches“. In: *CoRR* abs/1807.1 (2018), pp. 128–141. DOI: 10.1007/978-3-030-06019-0. arXiv: 1807.10059. URL: http://arxiv.org/abs/1807.10059http://link.springer.com/10.1007/978-3-030-06019-0.

[31] David Garlan and Mary Shaw. „AN INTRODUCTION TO SOFTWARE ARCHITECTURE“. In: *Advances in Software Engineering and Knowledge Engineering*. 1993, pp. 1–39. ISBN: 9810215940. DOI: 10.1142/9789812798039_0001. URL: http://www.worldscientific.com/doi/abs/10.1142/9789812798039{_}0001.

[32] Martin Garriga. *Microservice (de-)composition tooling*. 2016. URL: https://github.com/mgarriga/decomposer (visited on 08/08/2019).

[33] Jean Philippe Gouigoux and Dalila Tamzalit. „From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture“. In: *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings* (2017), pp. 62–65. DOI: 10.1109/ICSAW.2017.35.

[34] Gregor Grambow, Roy Oberhauser, and Manfred Reichert. „Providing Automated Holistic Process and Knowledge Assistance During Software Modernization“. In: *Computer Systems and Software Engineering* (2017), pp. 351–395. DOI: 10.4018/978-1-5225-3923-0.ch015.

[35] Michael Gysel et al. „Service Cutter: A Systematic Approach to Service Decomposition“. In: *Service-Oriented and Cloud Computing*. Ed. by Marco Aiello et al. Vol. 8745. Cham: Springer International Publishing, 2016, pp. 185–200. ISBN: 978-3-319-44482-6. DOI: 10.1007/978-3-662-44879-3. arXiv: 9780201398298. URL: http://link.springer.com/10.1007/978-3-662-44879-3.

[36] Sara Hassan, Nour Ali, and Rami Bahsoon. „Microservice Ambients: An Architectural Meta-Modelling Approach for Microservice Granularity“. In: *Proceedings - 2017 IEEE International Conference on Software Architecture, ICSA 2017* (2017), pp. 1–10. DOI: 10.1109/ICSA.2017.32. URL: https://ieeexplore.ieee.org/abstract/document/7930193.

[37] Sara Hassan and Rami Bahsoon. „Microservices and their design trade-offs: A self-adaptive roadmap“. In: *Proceedings - 2016 IEEE International Conference on Services Computing, SCC 2016* (2016), pp. 813–818. DOI: 10.1109/SCC.2016.113.

[38] Wilhelm Hasselbring. „Microservices for Scalability“. In: (2016), pp. 133–134. DOI: 10.1145/2851553.2858659.

[39] Wilhelm Hasselbring and Guido Steinacker. „Microservice architectures for scalability, agility and reliability in e-commerce“. In: *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings* (2017), pp. 243–246. DOI: 10.1109/ICSAW.2017.11.

[40] Robert Heinrich et al. „Performance Engineering for Microservices“. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion - ICPE '17 Companion* (2017), pp. 223–226. DOI: 10.1145/3053600.3053653. URL: http://dl.acm.org/citation.cfm?doid=3053600.3053653.

[41] Slinger Jansen et al. „How do professionals perceive legacy systems and software modernization?“ In: (2014), pp. 36–47. DOI: 10.1145/2568225.2568318.

[42] Gabor Kecskemeti, Attila Csaba Marosi, and Attila Kertesz. „The ENTICE approach to decompose monolithic services into microservices“. In: *2016 International Conference on High Performance Computing and Simulation, HPCS 2016* (2016), pp. 591–596. DOI: 10.1109/HPCSim.2016.7568389.

[43] Holger Knoche. „Sustaining Runtime Performance while Incrementally Modernizing Transactional Monolithic Software towards Microservices“. In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering - ICPE '16* (2016), pp. 121–124. DOI: 10.1145/2851553.2892039. URL: http://dl.acm.org/citation.cfm?doid=2851553.2892039.

[44] Holger Knoche and Wilhelm Hasselbring. „Using Microservices for Legacy Software Modernization“. In: *IEEE Software* 35.3 (2018), pp. 44–49. ISSN: 07407459. DOI: 10.1109/MS.2018.2141035.

[45] Jussi Koskinen et al. „Software modernization decision criteria: An empirical study“. In: *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR* (2005), pp. 324–331. ISSN: 15345351. DOI: 10.1109/CSMR.2005.50.

[46] Alexandr Krylovskiy, Marco Jahn, and Edoardo Patti. „Designing a Smart City Internet of Things Platform with Microservice Architecture“. In: *Proceedings - 2015 International Conference on Future Internet of Things and Cloud, FiCloud 2015 and 2015 International Conference on Open and Big Data, OBD 2015* (2015), pp. 25–30. ISSN: 0006291X. DOI: 10.1109/FiCloud.2015.55. arXiv: arXiv:1011.1669v3.

[47] Sebastian Lehrig, Hendrik Eikerling, and Steffen Becker. „Scalability, Elasticity, and Efficiency in Cloud Computing“. In: 1 (2015), pp. 83–92. DOI: 10.1145/2737182.2737185.

[48] Alessandra Levcovitz, Ricardo Terra, and Marco Tulio Valente. „Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems“. In: *CoRR* abs/1605.0 (2016). arXiv: 1605.03175. URL: http://arxiv.org/abs/1605.03175.

[49] Bennet P. Lientz, E. Burton Swanson, and Gerry Edward Tompkins. „Characteristics of application software maintenance“. In: *Communications of the ACM* 21.6 (1978), pp. 466–471. ISSN: 00010782. DOI: 10.1145/359511.359522. URL: http://doi.acm.org/10.1145/359511.359522.

[50] Duo Lu et al. „A Secure Microservice Framework for IoT“. In: *Proceedings - 11th IEEE International Symposium on Service-Oriented System Engineering, SOSE 2017* (2017), pp. 9–18. DOI: 10.1109/SOSE.2017.27.

[51] Genc Mazlami, Jurgen Cito, and Philipp Leitner. „Extraction of Microservices from Monolithic Software Architectures“. In: *Proceedings - 2017 IEEE 24th International Conference on Web Services, ICWS 2017* (2017), pp. 524–531. DOI: 10.1109/ICWS.2017.61.

[52] Arthur Molnar and Simona Motogna. „Discovering maintainability changes in large software systems*“. In: *ACM International Conference Proceeding Series* Part F1319 (2017), pp. 0–5. DOI: 10.1145/3143434.3143447.

[53] Ola Mustafa et al. „GranMicro: A Black-Box Based Approach for Optimizing Microservices Based Applications“. In: *From Science to Society*. Ed. by Benoît Otjacques et al. Cham: Springer International Publishing, 2018, pp. 283–294. ISBN: 978-3-319-65687-8. DOI: 10.1007/978-3-319-65687-8. URL: http://link.springer.com/10.1007/978-3-319-65687-8.

[54] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. „An Analysis of Public REST Web Service APIs“. In: *IEEE Transactions on Services Computing* PP.c (2018), pp. 1–14. ISSN: 19391374. DOI: 10.1109/TSC.2018.2847344.

[55] Sam Newman. *Building Microservices*. 2015, p. 280. ISBN: 978-1-491-95035-7. DOI: 10.1109/MS.2016.64. arXiv: 1606.04036.

[56] Maria Orlowska et al. „Service-Oriented Computing“. In: *ICSOC 2003, First International Conference, Trento, Italy, December 15-18, 2003*. January. 2003, pp. 25–28.

[57] Mert Ozkaya. „Do the informal & formal software modeling notations satisfy practitioners for software architecture modeling?“ In: *Information and Software Technology* 95.May 2017 (2018), pp. 15–33. ISSN: 09505849. DOI: 10.1016/j.infsof.2017.10.008. URL: https://doi.org/10.1016/j.infsof.2017.10.008.

[58] Jaroslav Pokorny. „NoSQL databases: A step to database scalability in web environment“. In: *International Journal of Web Information Systems* 9.1 (2013), pp. 69–82. ISSN: 17440084. DOI: 10.1108/17440081311316398. arXiv: /doi.org/10.1108/17440081311316398 [https:].

[59] Jennifer Riggins. *Miniservices: A Realistic Alternative to Microservices*. 2018. URL: https://thenewstack.io/miniservices-a-realistic-alternative-to-microservices/ (visited on 02/26/2019).

[60] Per Runeson et al. *Case Study Research in Software Engineering*. 2012, p. 216. ISBN: 9781118104354. DOI: 10.1002/9781118181034. URL: http://www.wiley.com/WileyCDA/WileyTitle/productCd-1118104358.html.

[61] Dharmendra Shadija, Mo Rezai, and Richard Hill. „Towards an understanding of microservices“. In: *2017 23rd International Conference on Automation and Computing (ICAC)*. 2017, pp. 1–6. DOI: 10.23919/IConAC.2017.8082018.

[62] Tushar Sharma and Diomidis Spinellis. „A survey on software smells“. In: *Journal of Systems and Software* 138 (2018), pp. 158–173. ISSN: 01641212. DOI: 10.1016/j.jss.2017.12.034.

[63] Davide Spadini et al. „On the relation of test smells to software code quality“. In: *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018* (2018), pp. 1–12. DOI: 10.1109/ICSME.2018.00010.

[64] Yuqiong Sun, Susanta Nanda, and Trent Jaeger. „Security-as-a-service for microservices-based cloud applications“. In: *Proceedings - IEEE 7th International Conference on Cloud Computing Technology and Science, CloudCom 2015* (2016), pp. 50–57. DOI: 10.1109/CloudCom.2015.93.

[65] Inese Supulniece et al. „Decomposition of Enterprise Application: A Systematic Literature Review and Research Outlook“. In: *Information Technology and Management Science* 18.1 (2015), pp. 30–36. ISSN: 2255-9094. DOI: 10.1515/itms-2015-0005.

[66] E. Burton Swanson. „The dimensions of maintenance“. In: *Proceedings of the 2nd international conference on Software engineering* (1976), pp. 492–497. DOI: 10.1145/390016.808430. URL: http://dl.acm.org/citation.cfm?id=800253.807723.

[67] Inese Šūpulniece et al. „Source Code Driven Enterprise Application Decomposition: Preliminary Evaluation“. In: *Procedia Computer Science* 77 (2015), pp. 167–175. ISSN: 18770509. DOI: 10.1016/j.procs.2015.12.377.

[68] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. „Architectural Patterns for Microservices: A Systematic Mapping Study“. In: Closer (2018), pp. 221–232. DOI: 10.5220/0006798302210232.

[69] Johannes Thönes. „Microservices“. In: *IEEE Software* 32.1 (2015). ISSN: 07407459. DOI: 10.1109/MS.2015.11.

[70] Raoul Vallon et al. „Systematic literature review on agile practices in global software development“. In: *Information and Software Technology* 96.April 2017 (2018), pp. 161–180. ISSN: 09505849. DOI: 10.1016/j.infsof.2017.12.004. URL: https://doi.org/10.1016/j.infsof.2017.12.004.

[71] Colin C. Venters et al. „Software sustainability: Research and practice from a software architecture viewpoint“. In: *Journal of Systems and Software* 138.January 2018 (2018), pp. 174–188. ISSN: 01641212. DOI: 10.1016/j.jss.2017.12.026. URL: https://doi.org/10.1016/j.jss.2017.12.026.

[72] June M Verner et al. „Guidelines for industrially-based multiple case studies in software engineering“. In: *2009 Third International Conference on Research Challenges in Information Science* (2009), pp. 313–324. DOI: 10.1109/RCIS.2009.5089295. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5089295.

[73] Mario Villamizar et al. „Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud Evaluando el Patrón de Arquitectura Monolítica y de Micro Servicios Para Desplegar Aplicaciones en la Nube“. In: *10th Computing Colombian Conference* (2015), pp. 583–590. ISSN: 9781467394642. DOI: 10.1109/ColumbianCC.2015.7333476.

[74] Manish Virmani. „Understanding DevOps & bridging the gap from continuous integration to continuous delivery“. In: *5th International Conference on Innovative Computing Technology, INTECH 2015* Intech (2015), pp. 78–82. DOI: 10.1109/INTECH.2015.7173368.

[75] Tomislav Vresk and Igor Cavrak. „Architecture of an interoperable IoT platform based on microservices“. In: *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2016 - Proceedings* (2016), pp. 1196–1201. DOI: 10.1109/MIPRO.2016.7522321.

[76] Zhongxiang Xiao, Inji Wijegunaratne, and Xinjian Qiang. „Reflections on SOA and Microservices“. In: *Proceedings - 4th International Conference on Enterprise Systems: Advances in Enterprise Systems, ES 2016* (2017), pp. 60–67. DOI: 10.1109/ES.2016.14.

[77] Uwe Zdun, Elena Navarro, and Frank Leymann. „Ensuring and Assessing Architecture Conformance to Microservice Decomposition Patterns“. In: *Service-Oriented Computing*. Ed. by Michael Maximilien et al. Vol. 9435. Cham: Springer International Publishing, 2017, pp. 411–429. ISBN: 978-3-319-69035-3. DOI: 10.1109/ICEmElec.2014.7151186. arXiv: 9780201398298. URL: http://link.springer.com/10.1007/978-3-662-48616-0.

[78] Olaf Zimmermann. „Microservices tenets: Agile approach to service development and deployment“. In: *Computer Science - Research and Development* 32.3-4 (2017), pp. 301–310. ISSN: 18652042. DOI: 10.1007/s00450-016-0337-0.

Answers given by the interviewees

This appendix contains the answers given by the domain experts to the questions from section 9.2.

Chapter 10 discusses their meaning and summarizes these answers.

Answers

Interviewee A

1. I'm a developer.
2. Since May 2018 (1 year 3 months)
3. ~8 years
4. I think it's a good idea (as long as it is done properly and the process is guided by good POs/TAs)
5. Really depends. Sometimes this could be helpful (a technical description can make the development much easier, but it shouldn't be too detailed -> could restrain the developer)
6. A more clear domain specific abstraction can be realized. At least as long as it is being used consistently (sometimes services/helper take over where they probably shouldn't). Needs to be clear what a Facade should provide
7. Hard to say from my standpoint (since I'm MOD), but i guess the impediments are minimal (but still there).
8. Depends on how easily it is possible to draw strict lines where those teams should split code into modules...sometimes this is very hard / impossible. For complex projects that depend on a lot of different domains, 1 team is probably preferable.
9. I don't think so.

10. Yes. The modules are way easier to grasp and understand. Therefore a developer can much quicker begin with a task since it's easier to understand the domain (not so much in a monolith)

11. It's much more maintainable. Easier introduction to each module, faster build times, less overhead setting up and launching a module. The monolith could take 1-2 days alone to properly set up for a beginner.

12. It also gets slightly better in the monolith: faster build times, less setup time for (now handled in modules) configurations. Code analysis also gets easier (don't end up in parts where you shouldn't be)

13. Monolith: lots of changes required due to extreme complexity. Microservices: can be refactored more easily and can be made less complex from the get go.

14. Microservices: Way more scalable. Monolith almost needs a LHC to run properly ;)

15. Yes. It is fairly easy to add new modules. New teams can expand the feature-set by adding new modules without additional overhead (at least once everything is setup/configured properly to support those modules)

16. It's way easier to add new functionality without breaking any existing code/logic.

17. Couldn't say...probably.

18. In a project of this size: sure. But it is manageable.

19. Huge codebase that has grown over several years by ∼100devs...it is really hard/taxing to modularize that amount of code.

20. MOD FTW! Every developer who has worked his share in the all-repo will be glad to work in a new module

Interviewee B

1. Senior Developer

2. ∼1 year

3. ∼5 years

4. Mostly content. However since business contexts don't necessarily have to be equal to technical contexts, there's sometimes a lot of tight coupling between services. However I also think that splitting by technical contexts would have its own set of problems as well.

5. I think the overhead for something like this in a seizable project would be too high.

6. In my opinion, it led to some confusion where you were never quite sure whether to use the facade or the service, sometimes both had a method you needed, sometimes only one of them. Sometimes you just had the UUID of the object but the methods just took long ids, or vice versa, so there was quite some unnecessary mapping. This also led to some insidious performance problems, where those mappings (which are usually a REST call or a query) are done in mapper methods and then used for mapping collections.

7. Definitely. Modularization without breaking something/doing breaking changes is incredibly hard, so often times, we had to fix bugs that turned up because of modularization or update dependencies to be compatible with the new breaking changes.

8. Not very well if there is ongoing development in the parts that are being modularized. It usually leads to hard-to-fix merge conflicts as well as mixed codebases, where the newly developed code is built onto the old code base before modularization (long id/uuids mostly)

9. Since our application is very large and not 100% covered by automated tests, bugs sometimes slip through modularization efforts.

10. With the move to a more micro-service oriented architecture, maintenance usually becomes less effort for one module, but has then to be repeated for modules. What would be needed here are some further abstractions over the microservices in general to reduce this not complicated, but cumbersome part of maintaining microservices.

11. It's usually not as much effort compared to the monolith, but one that has to be repeated multiple times. Also sometimes finding bugs in the modularized parts becomes harder when they go over module boundaries. Debugging well, as well as discoverability, are usually harder than in the monolith

12. Sometimes modules get modularized, but not fully, where some parts are already pulled out of the monolith, but some functionality is still in the monolith as well, which makes debugging and finding stuff harder than they should be.

13. Development in new microservices is a lot faster and more developer friendly (some of the technologies used in the monolith are rather bad (Tapestry, bitronix, ...)). Code, build, test cycles are a lot quicker in the microservices than in the monolith. Unfortunately it can sometimes be quite hard to figure out which modules you need to be running for your application to work though, but this applies to the monolith and microservices as well. Startup times for typical services are ~30 seconds for microservices and 5+ minutes for applications in the monolith.

14. Obviously microservices, since it's quite a lot faster to see results and you don't work on woefully outdated code.

15. Yes, the new microservices are mostly built in a way to allow to run multiple replicas, which should make them easier to scale. They still all use the same database though, so this would probably bottleneck pretty hard. The old monolith holds a lot of state, so running multiple replicas of it is not advisable.

16. See question above.

17. There were some efforts to make the monolith more stateless to be able to run replicas of it as well, but I don't know the status of that.

18. Yes, the monolith holds a lot of state, so you cannot just load balance every request independently, but rather have to have to knowledge of authentication to keep the same user on the same node.

19. Budget and time constraints, as well as spreading themselves too thin by working on too many fronts at once.

20. In my eyes no modularization is better than half-finished, then abandoned modularization efforts like the UUID introduction.

Interviewee C

1. tester

2. 1,5 years

3. 5 years

4. For the system we are working at, including the available documentation it is the better way for modularization

5. -

6. Other teams are still able to develop new features.

7. Not that much

   During modularization also the integration must be taken into account including new features developed parallel to the modularization including testing.

8. With the approach we are using it is quit good possible for multiple teams to „cut" modules out of the monolith and integrate them in the rest of the monolith without stopping other teams from developing.

9. System is still operating - releases and testing must be planned when finally integrating the new module.

10. So far for new modules - yes: bug fixing and adding new functionality is way faster and better testable.

    Adding new functionality to the monolith - takes way longer and also needs way more testing resources. Same with test automation. Building time and deployment take its time too. On the other hand - software update for modules have to be done for each module separate. Changes in the system structure also affect all modules have to be done for each module too.

11. Defect testing is quit easier in the module also adding new features is within the module can be developed and tested faster and easier.

12. See above

13. From the testing perspective: Testing takes longer as the monolith needs time for all the calls, unknown changes done by other teams can result in failing tests.

    Microservice - faster and easier to test - also need to test integration to the monolith including REST calls.

14. I will talk about testing in this case too I prefer the module - most test cases can be automated - not so many side effects by changes from other teams. Con.: Automation of GUI integration tests are time consuming in development and maintaining.

15. Not really. Regarding the time i am working on the project and the paused modularization.

16. -

17. -

18. Hardware and infrastructure problems. Also budget question and new features are more important than modularization at the moment.

19. Bad documentation of older code. New functionality is added to the monolith. Speed - performance when working with the system.

20. -

Interviewee D

1. Senior Developer

2. 2 years

3. 10 years

4. This might work when starting with a new microservice-based project, however in a large existing application, it might not be possible.

   Also, in addition to business context-based microservices, there are also technical „contexts“ or cross-cutting concerns that are not bound to a specific business context.

5. No in this case, due to the size of the project. There are probably thousands of services with disputable responsibilities. Creating a clear formal model of the responsibilities would require a refactoring/restructuring of the application that would exceed the effort of the transition to microservices by far.

   A formal approach might work in a project with perfect/clean code, which I haven't seen yet.

6. Facades were used in the project before. The initial approach was to replace existing facades using new ones and simplify the codebase that way. This slowed down the modularization drastically. The main reason was the lack of business expertise by the developers, Simplifying the existing facades without the business knowledge proved to be impossible, with the introduction of major bugs as a result. Without business understanding, it is not possible to introduce a new facade-layer.

   The new approach is to replace existing facades 1:1 by implementing Adapters to new modules. This makes for faster development cycles. Method signatures stay the same, existing tests can be reused.

7. Yes it does.

   a) By introducing bugs into the existing code base for badly or insufficiently tested code.

   b) Other teams no longer possess the possibility of changing modularized code. Debugging times are longer.

   c) Other teams sometimes have to wait for new „not-yet-modularized/ready“ functionality or implement legacy code that will be removed shortly afterwards.

8. Due to the large codebase, this has not been an issue in the project.

9. No. Technically, the modularization is structured in a way that existing functionality is replaced. Operations is not impeded, however with the introduction of major bugs, parts of the application might not be usable.

   In newer approaches, the introduction of feature toggles made this approach ever better, since it is possible to „rollback“ to the previous, non-modularized state, without code modifications.

10. Yes.

    - Faster startups, due to smaller code-base.

- Faster re-deployments, du to fast build-times and smaller code base.
- Faster releases, due to smaller code-base.
- Slower code changes when multiple business-contexts are concerned, because of multi-repository dependencies.

11. The maintainability depends on whether APIs are changed and whether the changes are backwards compatible or new API versions are introduced.

    If the API doesn't change, maintainability got better, because the build-test-release cycle in a smaller application is much faster.

    If the API changes but the modifications are backwards-compatible, the maintainability is slightly worse. First, the changes have to be implemented and a new API version is released. The client dependent module than has to update its client-version and build/test/redeploy. This doubles the time required before modularization.

    If breaking API changes are introduced, all dependent modules must update their client versions until normal operations are restored, the maintenance effort is multiplied by the number of affected modules.

12. see previous answer.

13. The build/test/release cycles are much longer due to the larger codebase and longer startup-times. Developers are also confronted with an „in between“ state, where parts of the code-base are already modularized and other parts are not.

14. I prefer a combined approach of „smaller monoliths“. I do not always agree with the size of the „microservice“ because, IMHO, the additional overhead of implementing „microservices“ does not justify gain in deployment speed. I believe that development speed is sometimes even slower due to modularization.

    My preferred approach would be to start with a monolith and split up contexts „when needed“ based on development speed. Once the development is starting to get slow due to the size of the application, split it up.

15. Theoretically yes, because parts of the application can be scaled independently. However, in practice, this is not yet happening.

16. Theoretically, if slow/demanding parts of the application are identified, they can be scaled-up independently of the rest of the application.

17. Due to new projects being implemented as separate „microservices“, the demand of existing monoliths is not increasing. However the overall demand is increasing because the requirements for all microservices combined are much higher than for previous monoliths.

18. Not that I currently know of.

19. Budget. There is only a single small team, dedicated to modularization. Multiple possible „microservices“ were identified but due to the size of the team, they can only be implemented one-by-one instead of in-parallel.

20. no :)

Interviewee E

1. TA

2. 2 years

3. 10 years

4. I find choosing boundaries on the basis business capabilities/contexts a good strategy for extracting possible microservices from a monolith. Problems may arise early on if the current monolith is not organized in internal business context related modules already.

5. I think that the added complexity will normally not lead to a better solution in the end. In the contrary this approach might lead to more problems developing and maintaining the project.

6. Using facades with defined interfaces enable switching different backend implementations quite effectively. This is a well known and proven strategy.

7. Modularization topics may be prioritized depending on the effect that they have on teams that need to develop business capabilities. Doing the most cross cutting concerns early on in the modularization path would benefit the outcome in the long run.

8. If the monolith is already internally modularized it should be possible to modularize different parts at the same time with multiple teams. If the internal separation/modularization is not well defined in the first place it may lead to problems when multiple teams try to modularize the monolith in microservices.

9. Continuous operation of the system should not be impeded. But operational requirements should get analysed and prepared extensively before starting the transition to a more modularized system.

10. Separated modules will be easier to maintain internally. But there need to be stable contracts in place to successfully operate the interacting system of different business modules. Knowledge that extends the given borders of the business module will be crucial for long term stability of the system. Transfer of crucial cross module knowledge should be a priority to minimize maintenance problems. Also there need to be integration tests in place that assert the overall system behaviour. Mocking the ideal external world in the separated modules will certainly lead to problems otherwise.

11. Splitting a monolith in microservices will lead to improved internal maintainability of already modularized parts but will also increase complexity of the system as operating the system will be more complex. Problems arise typically in handling transactions over multiple business modules. Complexity of operating the system will increase therefore increase the maintenance burden in a different region of the system.

12. Maintainability of the remaining monolith may decrease internally but as the complexity of system orchestration will increase also the system maintainability may increase as a whole.

13. Development of a part in the microservice will need to take more time to adhere to or introduce new stable interfaces. Subtle changes in a microservice will still have the power to influence the whole system.

14. As this system was complex and burdening the development cycle it was inevitable to split it into simpler modules. But splitting into too many small microservices in a complex business setting increases the overall system/operational complexity too much. It is beneficial to first split the monolith into smaller (maybe still somewhat monolithic) modules with defined interfaces and then decide if it would be really necessary to split further into small microservices. So working in the old monolith was associated with a very long build time and feedback loop that was really burdensome. Working within a smaller module or microservice was speeding up build time and feedback loops and therefore increasing the overall development efficiency.

15. Scalability of a single microservice should generally be easier but scalability of whole the system may also get more complex especially if business processes depend on the interaction of multiple involved microservices. Operational details like network stability and performance may become crucial for the overall system health and performance.

16. Scalability will be limited more and more by operational details like network and container platforms.

17. -

18. Operational impediments like network architecture/design can limit scalability.

19. Increased operational complexity with OpenShift and the network architecture. Increased organizational complexity and hidden/siloed knowledge in the teams developing the microservices.

20. -