An Approach for Identifying Microservices using Clustering on Control Flow and Data Flow

Bachelor Thesis of

Niko Benkler

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer: Prof. Dr. Ralf H. Reussner
Second reviewer: Jun.-Prof. Dr.-Ing. Anne Koziolek
Advisor: Dr. Robert Heinrich

02. Month 2019 – 04. Month 2019

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

PLACE, DATE

. . . .

(Niko Benkler)

Abstract

Powered by the rise of cloud computing, agile development, DevOps and continuous deployment strategies, the microservice architectural pattern emerged as an alternative to monolithic software design. Microservices, as a suite of independent, highly cohesive and loosely coupled services, overcome the shortcoming of centralized monolithic architectures. Therefore, prominent companies recently migrated their monolithic legacy applications successfully to microservice-based architecture. They key challenge is to find an appropriate partition of legacy applications - namely *microservice identification*. So far, the identification process is done intuitively based on the experience of system architects and software engineers, mainly by virtue of missing formal approaches and a lack of automated tool support.
However, when application grow in size and become progressively complex, it is quite demanding to decompose the system in appropriate microservices. To tackle this challenge, the thesis provides a formal, graph-based identification approach using clustering techniques. Based on the business point of view, the approach identifies structural dependencies and data object dependencies to build a weighted graph. Clustering tools identify clusters that correspond to possible microservice candidates. To evaluate the quality and the effect of the process, the approach is applied to a case study that has already been decomposed into several microservices.

Zusammenfassung

Angetrieben durch den Aufstieg von Cloud Computing, agilen Entwicklungsmethoden, DevOps und Continuous Deployment Strategien etablierte sich die Microservice Architektur als Alternative zum monolithischen Software Design. Microservices sind eine Ansammlung unabhängiger, in sich zusammenhängende, aber lose gekoppelter Services, die die Defizite zentralisierter, monolithischer Software bewältigen. Namhafte Unternehmen hatten erst kürzlich ihre monolithische Alt-Software in ein microservice-basiertes System überführt. Eine Schlüsselaufgabe dabei ist es, die richtige Aufteilung der Alt-Software zu finden. Dieser Prozess wird Microserviceidentifikation genannt. Bis jetzt wurde er weitestgehend intuitiv und auf Basis von Expertenwissen durchgeführt. Der Hauptgrund dafür liegt vor allem in fehlenden formalen Ansätzen und automatisierter Unterstützung durch Software.
Dennoch wachsen Applikation mit der Zeit und werden zunehmend komplexer, sodass die Aufteilung eines Systems in dieser Komplexität durchaus herausfordern ist. Die Thesis stellt daher einen formalen, graph-basierten Ansatz vor, der mittels Clustering-Techniken mögliche Microservices extrahiert. Der Ansatz basiert auf der Prozesssicht und identifiziert strukturelle- und datenobjektbasierte Abhängigkeiten, um daraus einen gewichteten Graphen zu erstellen. Basierend auf dem zuvor erstelltem Graph ermitteln ermitteln Programme Cluster, die je einem Microservice entsprechen. Um die Qualität und Wirkungsweise des Ansatzes zu überprüfen, wird er auf eine Fallstudie angewendet, welche bereits in Rahmen anderer Arbeiten in Microservices gegliedert wurde.

Contents

List of Figures

List of Tables

1 Introduction

The monolithic software architecture is the traditional pattern to design software, where functionality is bundled in one single, large application [7]. Although monoliths have their strength, like fast development and simple deployment, they become an obstacle when they grow in size and become more complex [24]. Incomprehensible code structure makes it difficult to add functionality, fix bugs and enable new software engineering approaches like Continuous Delivery and Continuous Deployment[25]. Besides, the rise of cloud computing demands a new architecture that can fully exploit the rich set of features given by the cloud infrastructure [20].
Inspired by service-oriented computing, the microservice Architecture is about to become a promising alternative to overcome the shortcomings of centralized, monolithic architectures and consequently gains popularity in both, academia and industry [2]. Benefits like the increase of agility, resilience or scalability [26], the ability to use different technology stacks and independent deployment [3] and the efficient resource utilization in cloud environments [20] explain the usage of microservice-based application by big companies like Google, Netflix, Amazon, eBay [7] and Uber [26].
This thesis describes the current state of the art regarding microservices extraction and provides a systematic approach to decompose a system into microservices.

1.1 Motivation

Monolithic software applications develop over time and become more and more complex. The software structure becomes highly coupled and hard to maintain [11]. To tackle this issues, software engineers started to decompose their system into modules and provide the functionality over the network as Web Services [13]. The so-called *Service-oriented Architecture* (SOA) provides logical boundaries between the different software modules to address the design challenge of distributed systems. Nevertheless, Baresi et. al state that the boundaries between modules in SOA are too flexible and the application results in "a big ball of mud" [3]. Microservices make these boundaries physical as each service runs in its own process and only communicates with other services through well-defined lightweight mechanisms like REST [26]. Chen et al. consider the microservice architecture as a particular approach for SOA [7]. Others look at it as an evolution of SOA with differences in service reuse [3] or consider it to be the "contemporary incarnation of SOA" combined with modern software engineering practices like Continuous Deployment [13]. There is no consensus about the relationship between microservices and SOA but they both share common characteristics. The microservice architecture has many advantages over the monolithic style. Sec.2.2 elaborates the main aspects of microservices, including several benefits. Netflix, for instance, is able to cope with one billion calls a day to its video streaming API, by migrating their monolithic system to a

high flexible, maintainable and scalable microservice architecture [7]. Consequently, moving existing applications to a microservice landscape is a upcoming philosophy in academia and industry [2].
Nevertheless, decomposing a system in loosely coupled, fine-grained and independent microservices is a time consuming task that requires tedious manual effort [13] and is technically cumbersome [8]. So far, it is done mainly intuitively and relies on the experience of software architects and system designers. Hence, a formal approach to identify microservices is required.

1.2 Research Questions and Contributions

The microservice architecture is a fast rising approach to structure a system in high cohesive but loosely coupled and independent services. Many companies like Amazon, migrated their monolithic legacy software to microservice in order to fully leverage the benefits of cloud computing and new software engineering approaches like Continuous Deployment [20]. Large applications are decomposed into small, independent microservices where each service can be independently scaled and deployed.
However, one of the biggest problem in designing a microservice architecture is to decompose a monolithic application into a suite of small services while keeping them loosely coupled and high cohesive. This challenging task is also known as *microservice identification* [2].
Baresi et al. state that a "proper" microservice identification defines how a system will be able to evolve and scale [3]. Others claim, that finding the optimal microservice boundaries [12] and service granularity [14] is the key design decision to fully leverage the benefits of microservices.
So far, the partition is performed mainly intuitively based on the experience and know-how of experts that perform the extraction. Hassan et al. criticises a lack of systematic approaches to reduce the complexity of the extraction process [14]. Extracting microservices from monoliths therefore requires tedious manual effort and can be very costly [26] [21]. This leads to the following Research Question (RQ):

RQ1: Which is the most appropriate strategy to extract microservices from a monolithic system?

To identify possible strategies, an extensive literature research is conducted. Suitable strategies and approaches are compared based on criteria identified in the literature research.

RQ2: What formal approach can be constructed to perform the extraction process without detailed know-how and manual effort?

To that end, the most promising strategy identified in RQ1 is used as basis. Thereupon, a formal approach is elaborated that aims to reduce the complexity and manual work that has to be done when extracting microservices from a monolithic system.

RQ3: What is the accuracy of the approach?

Research question RQ3 is tackled by applying the approach to the Common Component Modelling Example (*CoCoME*). The subsequent system decomposition is evaluated by comparing the identified microservices with two other approaches: First, Tyszberowicz et al. [26] provide a decomposition of CoCoME based on their approach. Second, we identified and implemented a microservice-based version of CoCoME manually.

1.3 Thesis Outline

The proposal is structured as follows:

- Chapter 2 presents the background information on monolithic software architecture and microservice-based architecture. For the latter one, benefits and challenges are elaborated.
- Chapter 3 introduces the common case study *CoCoME* that is used to apply and evaluate the approach. Special attention is given to the system specifications.
- Chapter 4 outlines the current state of the art concerning microservice identification. First, the process of literature review is presented. Secondly, the most promising strategies and approaches are described and further compared using adequate criteria.
- Chapter **??** sketches a first attempt for a formal approach, that is further elaborated during the ongoing work on this thesis.
- Chapter 6 explains how the elaborated approach can be applied and evaluated using the case study *CoCoME*
- The proposal is concluded by chapter 8, where a schedule and accompanying milestones are defined.

2 Background

2.1 Monolithic Software Architecture

The monolithic software architecture is a well-known and the most widely used pattern for Enterprise Applications, which usually are built in three main parts (top to bottom): The client-side user interface (Tier 3), the server-side application that contains the entire business logic (Tier 2) and the persistence layer handling the database access (Tier 1). Fig. 2.1 illustrates the architectural difference between a standard three tier application and a exemplary microservice-based architecture. The server-side application - *the monolith* - is a single unit and deployed on one application server [24]. The software structure, if well defined, is composed of self-contained modules (i.e. software components), where each module consists of a set of functions [8]. The monolith implements a complex domain model, including all functions, many domain entities and their relationships. For small applications, this approach works relatively well. They are simple to develop, test and deploy [26]. Fast prototyping is supported by the current frameworks and development environments (IDE), which are still oriented around developing single applications [24].

But once they grow in size, they become exceedingly difficult to understand and hard to maintain without reasonable effort [26] [12]. A complex and large code base prevents a fast addition of new features and makes the application risky and expensive to evolve [19]. Alterations to the system, even though they might be small, result in a redeployment of the whole monolith application due to its nature being a single unit [26]. Moreover, it is difficult to adopt newer technologies without rewriting the whole application, as monolith are build on a specific technology stack [24] [21].

Scaling is only possible by duplicating the entire application, namely *horizontal scaling*. Consequently, large portions of the infrastructure remains unused, if only parts of the application need to be upscaled or even used [17] [11].

Chen et al. provide a short résumé:

"Successful applications are always growing in size and will eventually become a monstrous monolith after a few years. Once this happens, disadvantages of the monolithic architecture will outweigh its advantages."

- Rui Chen [7]

2.2 Microservices

"The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API."

- Martin Fowler [10]

2.2.1 Definition

The above quotation is a widely adopted definition of the term *Microservice*, provided by M. Fowler and J. Lewis [10], the pioneers of the microservice architecture. However, the term is not formally defined. Amiri et al. describes microservices as a collection of cohesive and loosely coupled components, where each service implements a business capability. The author introduces three principles upon which the architecture is build: *Bounded Context, Size, Independence* [2].
The first principle is about related functionality, that is combined in a single business capability - the *bounded context* [26]. Each capability is implemented by one microservice. The *Size* of a microservice is defined by the number of features it provides (namely bundled functional capabilities) [18]. There is no consensus about the "proper" size of a microservice [23], but several guidelines exists: Services should focus on one business capability only [2]. Others state, that the size of a microservice should not exceed a level, where it cannot be rewritten within six weeks [18]. However, the sizes vary from system to system [26] and even different sizes for each microservice in a specific system are possible [23]. The bottom line of *Independence* is in Amiri's description of microservices as "a collection of high cohesive and loosely coupled components" [2]. High cohesive services implement a relatively independent piece of business logic (at the most one business capability). Further, microservices should hardly depend on each other, which is the idea of being loosely coupled [7].
Communication between microservices is achieved by lightweight message passing mechanisms such as *REST*. Each service exposes a well defined interface (*API*) with endpoints that provide information using standard data formats [26]. The design of microservices mainly follows the *Single Responsibility Principle (SRP)*: Each service should not have more than one reason to change [9]. The SRP mainly corresponds to the idea of not implementing more than one business capability. The following covers the benefits and challenges of the microservice architecture.

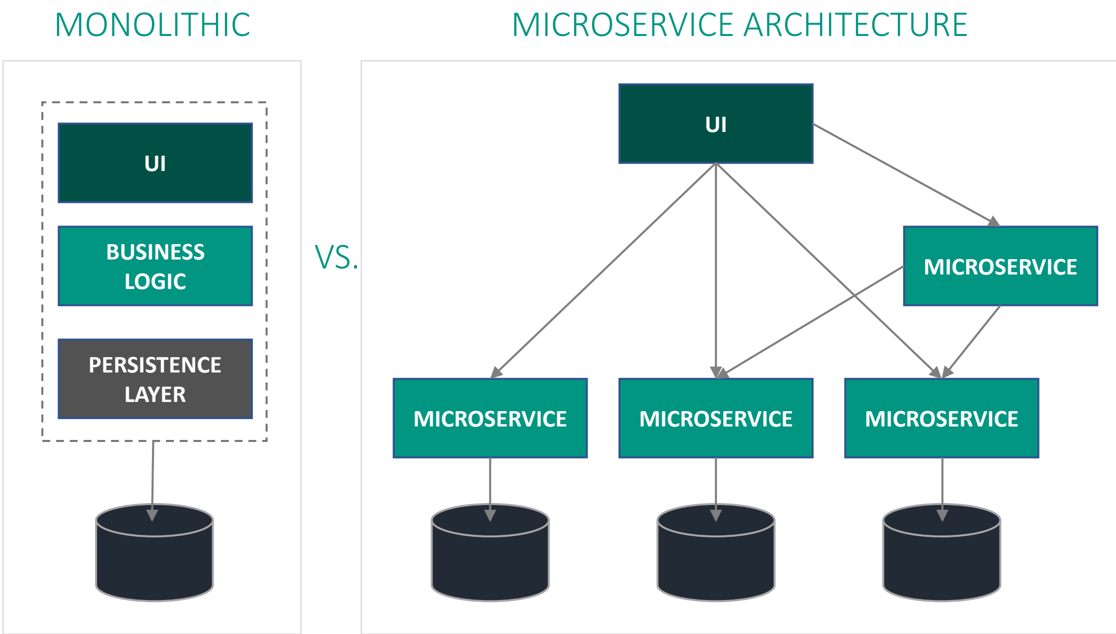

Figure 2.1: Monolithic vs. Microservice Architecture

2.2.2 Benefits

Fast and Independent Deployment
As a matter of fact, each microservice is deployed independently [3]. Changes to the code do not result in a full redeployment of the entire application [26]. Consequently, software developers are able to react much quicker to changes in business requirements. This includes an acceleration in error correction. Per contra, any changes in a monolithic code base requires a time consuming build of a new version and the redeployment of the entire application [10].

Availability, Resilience and Fault Isolation
Microservices are designed to operate independently of each other and to tolerate failure of services [10]. Large parts of the application remain unaffected of partly failures and the availability of the system is, at least partly guaranteed. Monolithic application do not provide this type of fault isolation. If a failure occurs, the whole application remains unavailable as it is usually running in a single process [21].

Scalability and Resource Utilisation
Small and independent microservices allow more fine-granular horizontal scaling [18]. Single services can be duplicated to cope with changing workload during runtime [7]. Thus, dynamic (de-)allocation of resources on demand prevent infrastructure from being idle [8]. Scaling monoliths can only be attained by duplicating the entire application , leaving resources unused [12]. Further, each microservice is deployed on the best suitable infrastructure for its needs, allowing a more efficient system organization [24].

Improved Productivity
In traditional software development, teams are divided based on their expertise: Database architects, UI-developers and server-side engineers, resulting in a three-tiered application (cf. Sec.2.1). Additionally, software engineers are responsible for the development only. Deployment is part of the operations team. This team structure results in high communication overhead and slows down the productivity [22].

In contrast, microservices are organized around business capabilities and require cross-functional, independent teams [2]. Each team has the full range of skills required for the end-to-end realization of a microservice, including UI-development, database architecture, back-end engineers and project management. This minimizes the communication and interaction between the teams and thus, speeds up the productivity. Ultimately, microservices enable a more agile flow of development and operation [12], also referred as *DevOps*.

Neutral development technology
Microservices are highly decoupled from each other, as they use standardized and lightweight communication mechanisms such as REST [26]. Microservices can be realized using different programming languages, technologies and even deployment environments [7]. Developers are consequently not longer limited to use a single technology for the whole application. They can choose the most appropriate technology for each particular business problem or try out some new technology without rewriting the whole application [13] [19].

2.2.3 Challenges

The previous section provides a vast amount of benefits that come with microservices. However, it is not the panacea of software engineering and has to face some challenges before being able to fully benefit from them. The challenges are further described in the following.

Expensive Communication
Microservice use network protocols such as *HTTP* to communicate with each other. Compared to standard, inter process communication (*IPC*) as used in monoliths, remote procedure calls a more expensive [1]. As a consequence, applications experience a decrease in performance as network communication is generally slower than IPC.

Technical Challenges
Microservices require a high degree of infrastructure automation [11]. The benefits of fast and independent deployment cannot be utilized, if it has to be done manually. Dynamic (de-)allocation of resources when scaling individual microservices need a well defined and structured cloud environment [20]. Besides, the distributed microservice landscape complicates the logging mechanisms and performance monitoring [1]. Traditional centralized logging, as it is used in monolithic applications, is not longer applicable. Instead, a careful aggregation system to gather logging and monitoring data from each service is required.

Organizational Challenges
The microservice approach needs the establishment of cross-functional teams [10]. Adopting *Continuous Practices*, such as *Continuous Deployment*, are essential for the success of a profitable microservice architecture. Therefore, closer collaboration between development teams, operational staff and management has to be established. In summary, a costly and time consuming restructuring process of the entire organization is required [5].

Data Consistency
Distributed systems need to share data. Heinrich et al. propose two concepts for the database

architecture [26]: The first concept applies the basic idea of the microservice approach, as it splits the database into several parts. Each microservice has its own database which manages the entities that belong to the corresponding bounded context. Higher speed and horizontal scaling are facing data consistency issues. Data needs to be synchronized which leads to inconsistency, if services are unavailable. The second concept is about sharing a single database. This approach overcomes the issue of consistency, as data is stored centrally. But sharing results in a loss of independence. Scaling can only be achieved through replicating the whole database. Research revealed, that the first concept is preferred [26].

Decomposition
Decomposing a system into microservice is a very complex task that requires experienced system architects and domain experts [10]. Identifying the right granularity of microservice is one of the key issues. Too fine grained services cause inefficiency due to a high amount of expensive inter-service calls [23]. Developing the basic communication infrastructure adds additional complexity and slows down the initial developing process [24].

3 Case Study

The *Common Component Modelling Example (CoCoME)* is a case study on software architecture modelling [16][15]. In this thesis, it is used to demonstrate and validate the presented approach. Sec.3.1 provides a short introduction of the demonstrator, followed by a presentation of its system specifications.

3.1 Introduction to CoCoME

CoCoME represents a trading system as it can be found in a supermarket chain. The main task is handling and processing sales at a single store of the chain. Therefore, customers can pick goods and place them on the *Cash Desk* whose main component is a *Cash Desk PC*. Several other components like *Bar Code Scanner*, *Light Display*, *Printer*, *Card Reader* and *Cash Box* are wired by the *Cash Desk PC*.
Multiple *Cash Desks* of a single store form a *Cash Desk Line*, which is connected to the *Store Server*. A set of stores in the CoCoME chain is organized as an enterprise where each store is connected to a single enterprise server.
More detailed description of the CoCoME system can be found in [16][15]. The next section provides information about the system requirements specifications in form of use cases.

3.2 System Specifications

The system specification is informal and given in the form of detailed use cases. Fig.3.1 provides an overview of the use case of CoCoME. A full detailed description can be found in [16].

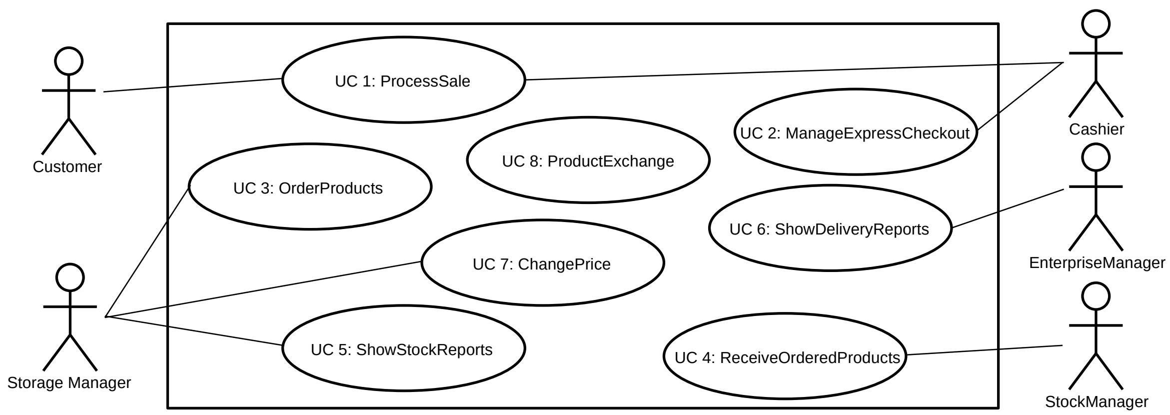

Figure 3.1: Use Case Diagram of CoCoME

Use case description

- *Process Sale:* Handles the products a customer wants to purchase and the payment (either cash or card).
- *Manage Express Checkout:* The cash desk switches automatically in the express mode (under certain conditions). The cashier is able to switch back in normal mode.
- *Order Products:* A store manager can order products from suppliers.
- *Receiver Ordered Products:* Ordered products which arrive at the store need to be checked for correctness and inventoried by the stock manager.
- *Show Stock Reports:* A store manager can request a stock-related report for his/her store.
- *Show Delivery Reports:* Calculation of the mean time for a delivery.
- *Change Price:* The sale price of a product is changed.
- *Product Exchange:* Automatic stock exchange if a store is running out of stock and other stores still have the required product

4 State of the Art

This chapter outlines the current state of the art regarding microservice identification. Sec. 4.1 presents the search strategy and several existing approaches (Table 4.1) that deal with the identification of microservices. Thereupon, the approaches are further explained and finally compared on the basis of several criteria.

4.1 Literature Review

The approaches mentioned in table 4.1 are the result of an extensive literature research which was conducted using the digital libraries IEEE [1], ACM [2] and SpringerLink [3]. The web search engine Google Scholar [4] provided further approaches and general information.
"Identifying Microservices using Functional Decomposition" [26] was provided by the supervisor of this thesis. Besides, *"Service Cutter - A Systematic Approach for Service Decomposition"* [13] was cited by various approaches, including [3] while the remaining papers were found using following search string:

["identify" OR "identification" OR "migrating" OR "monolith" OR "decomposition" OR "decompose monolith" OR "decompose"] AND "microservice"
OR
"microservice" AND ["identification" OR "transformation" OR "refactor"]

Table 4.1 presents the 8 most promising approaches in the area of microservice identification. Other papers like [27] only presented a conceptual train of thought, whereas [20], for instance, focuses on migrating strategies on infrastructural level. This thesis mainly focus on the identification part and disregards the actual implementation and deployment process afterwards. To compare the available approaches, criteria have to be defined. Sec.4.3 introduces 8 criteria and explains why they take part in the comparison. The comparison itself is done by applying the criteria to each approach using Table 4.2 and 4.3. Incidentally, the comparison including some criteria are inspired by the work of [12]. Further information is given in textual form in the same section.

[1]http://ieeexplore.ieee.org
[2]http://portal.acm.org
[3]http://www.springerlink.com
[4]http://scholar.google.com

Link	Titel	Author (Year)	Origin	Search String
[21]	Extraction of Microservices from Monolithic Software Architectures	G. Matzlami et. al. (2017)	Google Scholar	*microservice identification*
[2]	Object-Aware Identification of Microservice	M. J. Amiri (2018)	IEEE	*identification microservices*
[3]	Microservices Identification Through Interface Analysis	L. Baresi et. al. (2017)	SpringerLink	*microservice identification*
[26]	Identifying Microservices Using Functional Decomposition	S. Tyszberowicz et. al. (2018)	*provided*	*n/a*
[23]	Partitioning Microservices: A Domain Engineering Approach	I. J. Munezero et. al. (2018)	ACM	*partition microservices*
[7]	From Monolith to Microservices: A Dataflow-Driven Approach	R.Chen et. al	IEEE	monolith microservice
[8]	Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems	A. De Alwis et. al. (2018)	Google Scholar	identify microservices
[13]	Service Cutter: A Systematic Approach to Service Decomposition	M. Gysel et. al. (2016)	[3]	*n/a*

Table 4.1: List of authors and approaches

4.2 Approaches for Identifying Microservices

The following section provides a short introduction in the approaches mentioned in table 4.1.

Extraction of Microservices from Monolithic Software Architectures
The approach presented in [21] is a class based extraction model, that uses (meta-)information of a version control system *(VCS)* such as Git[5] to identify microservices. The approach is divided in two phases: The *Construction Phase* and the *Clustering Phase*. Starting with a given code base, the approach uses three different coupling strategies and the information provided by the *VCS* to transform the monolith into a weighted graph. Here, the nodes represent classes, and the edges have weights according to the chosen coupling strategy. In the second phase, a clustering algorithm determines possible microservices (each cluster is a microservice candidate).

Object-Aware Identification of Microservice
[2] identifies microservices from business processes, using the widely known *Business Process and Model Notation (BPMN)*. The approach uses clustering based on structural dependency and data object dependency. The first aspect is extracted from related activities within the business process model. A relation exists, if an edge directly connects a pair of activities or if only gateways are in between.
The latter aspect is based on the data object read and writes of each activity. Activities that are directly or indirectly connected and perform write or read operations are more likely to partition into the same microservice.

Microservices Identification Through Interface Analysis
In [3], the author proposes an approach that is based on semantic similarity of functionality specified through OpenApi[6] specifications (OpenApi defines a language-agnostic, standardized and machine-readable interface for RESTful APIs). The similarity depends on a reference vocabulary: each operation of the specification is analysed along with its resources (parameters, return values, complex types) and mapped to a concept of the chosen reference vocabulary. Each mapping has a score, based on a fitness function that uses the collocation of words (called terms) found in the operation and in the concepts. A co-occurrence matrix contains all mappings of possible pairs of terms and concepts. It is maximized to obtain the best mappings. Finally, this approach identifies potential candidate microservices, as fine-grained groups of operations, that are mapped to the same reference concept.

Identifying Microservices Using Functional Decomposition
The approach presented in [26] identifies microservices by functional decomposition of the software requirements, provided as use case specifications. In order to achieve the decomposition, the system is modelled as a finite set of *system operations* and the system's *state space*. Use cases provide the necessary input data: Verbs found in the use cases serve as *system operations* and nouns correspond to the *state variables* that the operations read or write. The state

[5] https://github.com/
[6] https://www.openapis.org/

variables constitute the state space. Relationships between the operations and the variables are stored in a relation table, that is visualized as a weighted graph. Finally, the approach uses graph analyse tools to determine clusters, where each cluster is a potential candidate of a microservice that fulfils the criteria of low coupling and high cohesion.

Partitioning Microservices: A Domain Engineering Approach
Munezaro et al. [23] propose an approach to identify appropriate microservices using *Domain-driven Design (DDD)* patterns. As a prerequisite, developers define a domain by using ubiquitous language. The domain indicates what the system does, precisely the system responsibilities, and what functionality it must implement. Domain experts define the boundaries of each responsibility and make it as a *business capability*, where a business capability is something that a system does in order to generate value. Each business capability is a microservice. When defining the boundaries, the focus is on the relationships among the services to minimize cross-cutting transactions.

From Monolith to Microservices: A Dataflow-Driven Approach
Chen et al. [7] uses a top-down data flow driven decomposition approach to determine high cohesive and loosely coupled microservices. Before the actual identification process starts, a *Data Flow Diagram (DFD)* needs to be constructed on the users' natural language description of the system to illustrate the detailed data flow. The first step of the approach consist of manually constructing a purified DFD, which focuses on data's semanteme and operations only. Afterwards, the purified DFD is algorithmically transformed into a decomposable DFD which is finally used to extract potential microservice candidates.

Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems
This is an approach that utilizes heuristics to specify two fundamental areas of microservice discovery: Function splitting based on common object subtypes and functional splitting based on common execution fragments across software [8].
The discovery process consists of two steps: First, the code, database tables and the SQL queries are evaluated to identify business objects and their relationships. Along with a set of given execution call graphs (different sequences of operations; generated though e.g. analysing log data), the information found is passed to the second part of the process. Algorithms processes the call graphs of the legacy system to derive a set of subgraphs and analyse which fragments are related to the same business objects in order to recommend possible microservices.

Service Cutter: A Systematic Approach to Service Decomposition
Gysel et al. [13] introduce a service decomposition tool based on 16 coupling criteria coming from industry and literature. A coupling criterion is a decision driver to decide whether data, operations or artifacts (generalized under the term *nanoentity*) should or should not be owned and exposed by the same service. Additionally, each criterion has a different score according to its priority. The input is in form of various *System Specification Artifacts (SSAs)*, such as domain models and use cases. The tool *Service Cutter* extracts coupling criteria information out of it, that must be prioritised by a user. To analyse and process the coupling criteria, Service Cutter creates a weighted graph. The nodes represent the nanoentities and the weights on and edge is the sum of all scores per criterion, multiplied by a user defined priority. In the end,

an exchangeable clustering algorithm identifies potential microservice candidates where each cluster correspond to a high cohesive and loosely coupled service.

4.3 Comparison

Table 4.2 and 4.3 provide a short description of the identified approaches mentioned above regarding some comparison criteria. The following criteria were used: **Basis Concept** recaptures the underlying approach of the microservice identification for classification purposes. **Prerequisites** presents the necessary preconditions for the success of the approach. For example, the approach mentioned in [21] cannot be used without meaningful VCS[7] data. The **Input** row describes the type and amount of input that is used realize the approach, i.e. Data Flow Diagrams in [7]. The row **Tool Support** indicates, whether the approach has been implemented or if other supporting tools are available to simplify the identification of high cohesive and loosely coupled microservices. The **Degree of human involvement** is part of the comparison, as this thesis aims to reduce the complexity of the service identification while keeping the required amount of expertise and manual tasks on a minimum. As evaluated in Sec.2.2, defining fine-grained microservices is a key challenge. Therefore, the approaches need to be compared in regard to the **Granularity of the recommended Microservices**. Some approaches allow an adjustable level of the granularity (e.g. [21]), whereas others generate a predefined granularity (e.g. always the most fine-grained microservice candidates [8]). **Validation** compares how the approaches are validated to strengthen the credibility of the individual results. Each approach might have some drawbacks regarding it's applicability to universal systems, required amount and type of input, user interaction and further expertise. **Limitations** is meant to point out the identified drawbacks. The following paragraph replenishes and explains the results given in Table 4.2 and Table 4.3 shortly:

The approach mentioned in [21] is the result of a master thesis [22]. Therefore, the degree of available information is larger compared to other approaches. The algorithmic recommendation of microservices candidates is implemented in a web-based, open source prototype and permits to choose three different coupling criteria, which can be combined for better results. Nevertheless, the main limitation relies in its type of input data: meaningful VCS data. For instance, developers must not commit changes on two independent functionalities together, but split it up. If this is not the case, the outcome may be wrong.

Amiri's approach [2] uses the open-source clustering software *Bunch*[8] . Further, information about the validation process (i.e. tested systems) are not given, but he claims that the process was successful. The weighting of the relationships lack formal explanation and need further analysis. Moreover, the author of the paper does no clearly differentiate data- and control flow. Nonetheless, the approach is straightforward and does not require any user involvement once the input data is available.

[7]Version Control System

[8]https://www.cs.drexel.edu/ spiros/bunch/

Baresi et al. [3] developed an experimental prototype to validate their results. They used a multitude of specifications and compared the outcome with results of software engineers and the tool *Service Cutter* [13]. Nevertheless, the outcome highly depends on the chosen reference vocabulary and well-defined APIs in the legacy system. Operations and resources (variables, return values) have to be expressive and represent what they do. Variable names like *temp* or *var1* would result in a useless service decomposition.

[26] uses external tools to realize the approach. Once the operations and state variable are identified, the identification method is universally applicable to all sort of legacy systems and also greenfield applications[9]. Nonetheless, identifying relevant nouns and verbs that represent the operations and state variables is only partly supported by tools. It still requires human expertise to eliminate duplicates and identify ambiguities.

Munezero et al. present a conceptional approach based on DDD patterns.[10]. Although the domain-driven design approach is currently the most common technique for identifying microservices ([26][10][11] and more), it does not tackle RQ2 (precisely *RQ2*) mentioned in Sec.1.2, as it requires the expertise and experience of domain experts.

Chen's semi-automate approach [7] is based on Data-flow Diagrams (DFD). Transforming the traditional DFDs to purified DFDs is not trivial an therefore requires a vast amount of additional manual work. Nevertheless, the purified DFD represents the real information flow of the corresponding business logic in the legacy system and therefore provides valuable information regarding potential inter- and intra service communication.

The approach presented by Alwis et al. [8] is a more complex method for microservice identification: Many steps and prerequisites are necessary to prepare the input data. For example, expressive *Log Files* are required to generate call graphs. Further, source code and the system's database is required to identify so-called business objects and their relationships. The latter one lacks a formal description in the paper. Additionally, the algorithms to identify potential microservice candidates are solely provided conceptually without further tool support.

Service Cutter [13] is a mature open-source software with available wiki. It was the first attempt to automatize service extraction and is therefore a reference project for other approaches like [21] and [7]. It uses 16 coupling criteria extracted from industrial experience and knowledge to decompose a system into services. However, the input requires special formats and consequently extensive and time consuming preparation.

[9] Project which lacks any constraints imposed by legacy systems
[10] Domain-driven Design

Approach/Criterion	Mazlami et al. [21]	Amiri [2]	Baresi et al. [3]	Tyszberowicz et al. [26]
Basic Concept	meta-data aided graph clustering	business process oriented graph clustering	semantic similarity of OpenApi specification	functional decomposition of sw requirements
Prerequisites	applications with meaningful VCS data	business processes and entities available	well-defined Api with proper naming	specification of software requirements
Input	Source Code and VCS meta data	BPMN business processes with data object reads and writes	reference vocabulary (fitness function), OpenApi specifications	use cases
Tool support	prototype available (https://github.com/gmaz frontend)	Clustering tool "Bunch"	experimental prototype (https://github.com/mgar riga/decomposer)	use external graph visualize and analyse tools
Degree of human involvement	choose amount of clusters that will represent the microservices	no interaction needed	user defines level of hierachy	manual elimination of synonyms, irrelevant nouns and verbs
Granularity	depends on choosen amount of clusters	depends on iteration of genetic algorithm for convergence of fitness function	depends on choosen hierachy lebel, varies from one to many	depends on size of business capability
Validation	experiements using open-source projects with VCS data (200 to 25000 commits, 1000 to 500000 LOC, 5 to 200 authors)	multiple experiments, results compared with domain experts knowledge	452 OpenApi specification, 5 samples compared with results of sw-engineers and [13]	case study, compared to three manual implementations
Limitation	need meaningful VCS data and ORM model for its data entities	given weight definitions lack formal explanation	depends on reference vocabulary and well-defined interfaces	manual revision of operations (nouns) and state variable (verbs)

Table 4.2: Comparison of Approaches, Part I

Approach/Criterion	Munezero et al. [23]	Chen et al. [7]	Alwis et al. [8]	Gysel et al. [13]
Basic Concept	define business capabilities by using domain-driven design patterns	algorithmic identification of microservices using data flows	graph-based identification process using heuristics to describe call graph similarities	service decomposition based on 16 coupling criteria
Prerequisites	domain defined by ubiquitous language	systems's data flows constructen on users' natural langugae description	Log files of legacy system	various System Specification Artifacts (SSAs) in specified format
Input	well defined domain model	Data Flow Diagrams (DFD)	Call Graphs, Source Code, System Database	instances of SSAs (e.g. ERM models, use cases)
Tool support	n/a	n/a	External tool for generating call graphs	implementation and wiki available
Degree of human involvement	domain experts define boundaries for business responsibilities	manual construction of purified DFD	no interaction needed	priorization of coupling criteria
Granularity	depends on the size of the defined business capability	most fine-grained ms candidates in terms of data operations	lowest granularity of sw based on structural and behavioural properties	n/a
Validation	demonstrated on sample domain	two case studies verified against relevant microservice principles and results of [13]	two experiemtns with complex enterprise systems (legacy vs. ms implementation)	validation via implementation and two case studies
Limitation	only conceptional approach, requires vast amount of expertise	transforming purified DFD not trivial (identifying same data operations requires expertise)	requires expressive log files to generate call graphs and identify business object relationships	generating SSAs in specified format is work intense

Table 4.3: Comparison of Approaches, Part II

5 Solution

//Erst mal nur Stichpunkte

5.1 Distance Metric

- Determine distance between pair of Objects in BPMN (Count activities in between)
- Basic Idea: Data accessed more successively is more connected and belongs to same service
- Pro: Easy
- : Contra: Not really Data Flow; pair occurs several times –> On time close, other times far away, what about average?

5.2 Count Activities Metric

- Count activities that access a pair of objects in BPMN
- Basic Idea: Data frequently accessed together is more connected and belongs to the same service
- Pro: represents idea of data cohesiveness
- Contra: Again activities (no clean data flow) but nothing else!

6 Evaluation

In the introduction chapter, the *Research Questions* are presented. The third question is about the elaborated approach and its evaluation.

RQ3: What is the accuracy of the approach?

To tackle this question, a *Goal Quality Metrics Plan* (GQM) is introduced to specify the key aspects of the evaluation. In a word, the elaborated approach is used to identify a set of microservice candidates which is compared to two reference sets of microservices.
In the following, a *GQM Plan* is introduced to specify what exactly needs to be evaluated. Also, metrics to measure the results of the comparison are introduced. Second to last, the two reference sets are illustrated before the actual results of the approach are finally depicted.

6.1 GQM Plan

Basili et al. originally proposed the *GQM Plan* (Goal Quality and Metrics) as a paradigm in software engineering but it is further extended to other engineering disciplines [4]. Based on a precise and structured procedure, the paradigm enables a high traceability and assessment of the engineering process. The main purpose is to specify the goals for a project, illustrate the data to define these goals and provide an environment to interpret the collected data. In the case of this thesis, the *GQM Plan* is used to clarify the desired intention of the evaluation in order to prevent unnecessary metrics and measurements and consequently reduce the expenditure of work.
The *GQM Plan* is a Top-Down approach and divided in three fundamental steps that precede the measurement and evaluation of results. First, the goal of the evaluation is defined on a conceptual level. Seconds, questions are delineated to achieve the specific goal. Finally, to answer the questions in a measurable way, metrics have to be defined that are associated with the questions.
In the following, the *GQM Plan* for this the consecutive evaluation is illustrated:

- **G1:** Determine the accuracy of the approach
- **G1.Q1:** What is the *Precision and Recall* of the identified microservices compared to the reference amount?
- **G1.Q1.M1:** Precision and Recall
- WAS WAR HIER NOHCMAL MIT DEN ZYKLISCHEN ABHÄNGIGKEITEN

6.2 Metrics

Using metrics is mandatory to measure the quality of the elaborated approach. In this case, it is required to choose a metric to classify a set of instances, namely microservices, regarding their relevance. Two reference sets are available as further depicted in Sec.6.3.
A metric that is capable to measure the relevance of a set of instances compared to a reference set is *Precision and Recall*. In subsequent, the proposed metric is briefly presented.

6.2.1 Precision and Recall

Precision and Recall is a classification metric that measures the relevance of retrievable items with respect to a reference set [6]. Commonly, two distinctions for items in the reference set are made: First, Retrieved or not Retrieved. More precisely, an item is retrieved if it is part of the selected items and vice versa. Secondly, Relevant or Not Relevant. As a result, all retrievable items belong to one and only one of four cells in the following matrix:

	Relevant	Not Relevant	Sum
Retrieved	$N_{ret \cap rel}$	$N_{ret \cap \overline{rel}}$	N_{ret}
Not Retrieved	$N_{\overline{ret} \cap rel}$	$N_{\overline{ret} \cap \overline{rel}}$	$N_{\overline{ret}}$
Sum	N_{rel}	$N_{\overline{rel}}$	N_{total}

Table 6.1: Retrieval Matrix, Source: [6]

Recall describes the completeness of the retrieval. In other words, how many relevant items are selected in regard to all possible relevant items.

$$Recall = \frac{N_{ret \cap rel}}{N_{rel}}$$

Precision illustrates the purity of the retrieval because it puts into proportion the number of retrieved relevant items and the number of all retrieved items.

$$Precision = \frac{N_{ret \cap rel}}{N_{ret}}$$

It is important to notice that $N_{\overline{ret}}$ and $N_{\overline{rel}}$ are not part of the formulas. With that in mind, it is possible to apply *Precision and Recall* to the prevalent evaluation scenario. With respect to Table 6.1, the reference set that is used forms the relevant items, or N_{rel}. Accordingly, the set of microservices identified by the proposed approach constitutes the retrieved items, or N_{ret}. The remaining part which are the non-relevant and non-retrieved items ($N_{\overline{ret} \cap \overline{rel}}$) are to be unimportant. The following list draws the analogy between Table 6.1 and the predominant evaluation scenario:

- **True Positives:** $N_{ret \cap rel}$, identified microservices that have a similar partner in the reference set
- **False Positives:** $N_{ret \cap \overline{rel}}$, identified microservices that do not have a similar partner in the reference set
- **False Negatives**: $N_{\overline{ret} \cap rel}$, microservices in the reference set that have not been discovered by the proposed approach
- **True Negatives**: $N_{\overline{ret} \cap \overline{rel}}$, microservices that are neither discovered by the approach, nor part of the reference set [1]

Figure 6.1: Precision and Recall for Microservices

[1]Note, that this amount consists of all imaginable microservices and is therefore an infinite set. As it is not used to calculate either of the metrics, it is negligible.

6.2.2 Some more Metrics if necessary

//Hier noch erklären

6.3 Reference Sets

To evaluate the approach, the identified set of microservices (cf. Sec.6.4) is compared to two alternative decompositions of the case study: First, a decomposition proposed in the paper *Identifying Microservices Using Functional Decomposition* [26] and second, a set of microservices which we manually identified.

6.3.1 Reference Set 1: Functional Decomposition Approach

Identifying Microservices Using Functional Decomposition [26] is a systematic approach to find a appropriate partition of a system into microservices. This paper emerged as a result of the collaboration of the Academic College Tel-Aviv Yafo, the Karlsruhe Institute of Technology and the Southwest University China and uses CoCoME as demonstrator as well.
As depicted in Sec.4.2, Tyszberowicz et al. utilize the Use Case specifications of CoCoME [16] as input for their decomposition approach. Several external tools are used to extract verbs an nouns from the use cases that serve as *system operations* and *state variables*. Irrelevant noun, verbs and synonyms are eliminated via brainstorming. The relationships between the aforementioned concepts are stored in a relation table. A relation exists, if a *system operation* reads or updates a *state variable*. Thereupon, the relation table is visualized as a weighted graph, which enables to identify clusters of dense relationships. Each cluster serves as a microservice candidate.
As mentioned in Sec.4.3, the compulsory and non-trivial revision of nouns and verbs to eliminate synonyms etc. is a substantial disadvantage. However, the evaluation results in Tyszberowicz's approach demonstrate, that the identified microservices are good candidates for a microservice-based system decomposition of CoCoME. The aforementioned evaluation includes a comparison to three independent software projects that implemented CoCoME. Two groups identified, apart from the naming, the same set of microservices. The third group identified a more detailed decomposition of the case study, but a revision reveals that the additional microservices are only a refinement of the proposed microservices.
The following microservices are identified:

- List of Services

6.3.2 Reference Set 2: Manual Decomposition

//Gefahr vs Aufwand reinbringen

6.4 Results

6.4.1 Identified Microservices

//Hier veranschaulichen was unser Ansatz gefunden hat

7 Conclusion

8 Organization

The schedule of the thesis is presented as a *Gantt Chart* (Fig.8.1). Each row represents either an activity (grey bar) or a milestone (black diamond). Milestones result upon the completion of an activity and mark an important step towards the completion of the thesis. The last two milestones are *Thesis Submission* and *Presentation* finish the thesis. All in all, eleven weeks are scheduled to complete the thesis, starting with the proposal presentation. The proposal date is set to 11.02.2019. Afterwards, an approach to identify microservices will be elaborated, based on the work of M.J. Amiri [2]. It will further be applied to CoCoME and the results are compared to the outcome presented in [26].

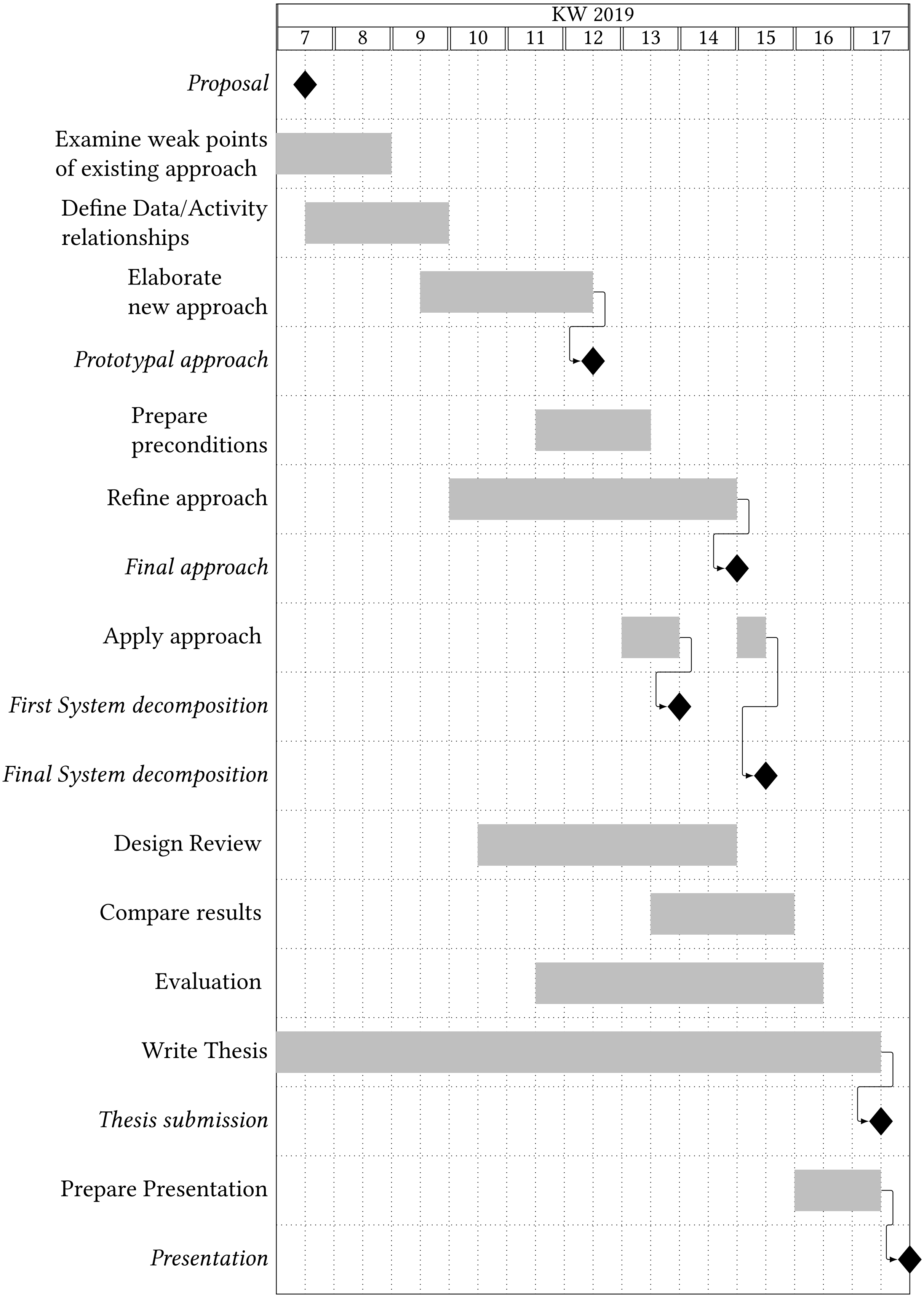

Figure 8.1: Schedule

Bibliography

[1] N. Alshuqayran, N. Ali, and R. Evans. “A Systematic Mapping Study in Microservice Architecture”. In: (Nov. 2016), pp. 44–51.

[2] M. J. Amiri. “Object-Aware Identification of Microservices”. In: (July 2018), pp. 253–256. ISSN: 2474-2473. DOI: 10.1109/SCC.2018.00042.

[3] Luciano Baresi, Martin Garriga, and Alan De Renzis. “Microservices Identification Through Interface Analysis”. In: (2017). Ed. by Flavio De Paoli, Stefan Schulte, and Einar Broch Johnsen, pp. 19–33.

[4] Victor R. Basili. *Software Modeling and Measurement: The Goal/Question/Metric Paradigm.* Tech. rep. College Park, MD, USA, 1992.

[5] Niko Benkler. *From Traditional Development to Continuous Deployment: Strategies and Practices in CI/CD Pipelines.* Accessed on 20.01.2019. URL: https://github.com/Benkler/Proseminar/blob/master/Niko_Benkler_Proseminar.pdf.

[6] Michael Buckland and Fredric Gey. “The relationship between recall and precision”. In: *Journal of the American society for information science* 45.1 (1994), pp. 12–19.

[7] R. Chen, S. Li, and Z. Li. “From Monolith to Microservices: A Dataflow-Driven Approach”. In: (Dec. 2017), pp. 466–475. DOI: 10.1109/APSEC.2017.53.

[8] Adambarage Anuruddha Chathuranga De Alwis et al. “Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems”. In: (2018). Ed. by Claus Pahl et al., pp. 37–53.

[9] D. Escobar et al. “Towards the understanding and evolution of monolithic applications as microservices”. In: (Oct. 2016), pp. 1–11.

[10] Lewis Fowler. *Microservices.* Accessed on 17.01.2019. URL: https://martinfowler.com/articles/microservices.html.

[11] P. Di Francesco, P. Lago, and I. Malavolta. “Migrating Towards Microservice Architectures: An Industrial Survey”. In: (Apr. 2018), pp. 29–2909.

[12] Jonas Fritzsch et al. “From Monolith to Microservices: A Classification of Refactoring Approaches”. In: *CoRR* abs/1807.10059 (2018). arXiv: 1807.10059. URL: http://arxiv.org/abs/1807.10059.

[13] Michael Gysel et al. “Service Cutter: A Systematic Approach to Service Decomposition”. In: (2016). Ed. by Marco Aiello et al., pp. 185–200.

[14] S. Hassan, N. Ali, and R. Bahsoon. “Microservice Ambients: An Architectural Meta-Modelling Approach for Microservice Granularity”. In: (Apr. 2017), pp. 1–10.

[15] Robert Heinrich, Kiana Rostami, and Ralf Reussner. “The CoCoME Platform for Collaborative Empirical Research on Information System Evolution”. In: (Jan. 2016). DOI: 10.5445/IR/1000052688.

[16] Sebastian Herold et al. “CoCoME - The Common Component Modeling Example”. In: *The Common Component Modeling Example: Comparing Software Component Models*. Ed. by Andreas Rausch et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 16–53. ISBN: 978-3-540-85289-6. DOI: 10.1007/978-3-540-85289-6_3. URL: https://doi.org/10.1007/978-3-540-85289-6_3.

[17] G. Kecskemeti, A. C. Marosi, and A. Kertesz. “The ENTICE approach to decompose monolithic services into microservices”. In: (July 2016), pp. 591–596.

[18] S. Klock et al. “Workload-Based Clustering of Coherent Feature Sets in Microservice Architectures”. In: (Apr. 2017), pp. 11–20.

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

[20] J. Lin, L. C. Lin, and S. Huang. “Migrating web applications to clouds with microservice architectures”. In: (May 2016), pp. 1–4.

[21] G. Mazlami, J. Cito, and P. Leitner. “Extraction of Microservices from Monolithic Software Architectures”. In: (June 2017), pp. 524–531.

[22] Genc Mazlami. *Algorithmic Extraction of Microservices from Monolithic Code Bases*. Accessed on 20.01.2019. URL: https://www.merlin.uzh.ch/contributionDocument/download/10978.

[23] I. J. Munezero et al. “Partitioning Microservices: A Domain Engineering Approach”. In: (May 2018), pp. 43–49.

[24] Chris Richardson. *Microservices: Decomposing Applications for Deployability and Scalability*. Accessed on 08.01.2019. May 2014. URL: https://www.infoq.com/articles/microservices-intro.

[25] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. “Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices”. In: *IEEE Access* 5 (2017), pp. 3909–3943.

[26] Shmuel Tyszberowicz et al. “Identifying Microservices Using Functional Decomposition”. In: (2018). Ed. by Xinyu Feng, Markus Müller-Olm, and Zijiang Yang, pp. 50–65.

[27] Zhiping Luo UU, Michel Korpershoek, and AnaMaria Oprescu VU. “Towards a MicroServices Architecture for Clouds”. In: ().