Master Thesis

February 7, 2017

Algorithmic Extraction of Microservices from Monolithic Code Bases

Genc Mazlami

of Glarus Nord GL, Switzerland (09-923-061)

supervised by

Prof. Dr. Harald C. Gall

Jürgen Cito

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Master Thesis

Algorithmic Extraction of Microservices from Monolithic Code Bases

Genc Mazlami

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Master Thesis

Author: Genc Mazlami, genc.mazlami@uzh.ch

Project period: 8.8.2016 - 8.2.2017

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

This thesis constitutes the final steps towards the completion of my master studies in computer science. This thesis was as rewarding as it was challenging. Without the support of some persons, this thesis would not have been possible in this form. Here, I would like to take the chance to thank them.

First and foremost, I would like to thank Jürgen Cito, PhD student at the Software Evolution and Architecture Lab at UZH. Without his great coaching, his creative suggestions and his critical examination of my work, this thesis would not have been possible. Thank you for your support and for the great collaboration, Jürgen.

I would also like to thank Dr. Philipp Leitner, Senior Research Associate at the s.e.a.l. lab at UZH. His experience and comments in scientific aspects of this thesis contributed important assistance. Furthermore, I thank Prof. Dr. Harald Gall for giving me the opportunity to work on this thesis at the Software Evolution and Architecture Lab of the University of Zurich.

Last but not least, my gratitude goes to friends and family. Special thanks to my sister Marigona, who never hesitated to read through the script and suggest linguistic improvements. Also, I want to thank my dear friends Joel Scheuner and Dominik Schöni – master students at the s.e.a.l. lab – for their ongoing motivation, fruitful discussions during my thesis and the great friendship. Finally, I would like to thank my parents for their neverending support, motivation and inspiration during the years of my studies.

Abstract

Driven by developments such as mobile computing, cloud computing infrastructure, DevOps and elastic computing, the microservice architectural style has emerged as a new alternative to the monolithic style for designing large software systems. Monolithic legacy applications in industry undergo a migration to microservice-oriented architectures. A key challenge in this context is the extraction of microservices from existing monolithic code bases. While informal migration patterns and techniques exist, there is a lack of formal models and automated support tools in that area. This thesis tackles that challenge by presenting a formal microservice extraction model to allow algorithmic recommendation of microservice candidates in a refactoring and migration scenario. A set of three coupling strategies is defined to mine structure information from monolithic code bases. A graph-based clustering algorithm is designed to extract microservice candidates from the model. The formal model is implemented in a web-based prototype. A performance evaluation demonstrates that the presented approach provides adequate performance. The recommendation quality is evaluated quantitatively by custom microservice-specific metrics. The evaluation results exhibit satisfactory scores in all of the considered metrics. The results show that the produced microservice candidates lower the average development team size down to half of the original size or lower. Furthermore, the size of recommended microservice conforms with microservice sizing reported by empirical surveys and the domain-specific redundancy among different microservices is kept at a low rate.

Zusammenfassung

Angetrieben durch Entwicklungen wie Mobile Computing, Cloud Computing Infrastrukturen, DevOps und Elastic Computing hat sich die Microservice-Architektur als neue Alternative für den Entwurf grosser Softwaresysteme etabliert. Existierende monolithische Systeme werden hin zu Microservice-Architekturen migriert. Eine entscheidende Herausforderung in diesem Kontext ist die Extrahierung von Microservices aus bestehendem monolithischem Quellcode. Hierfür existieren zwar informelle Migrations-Schemata und Techniken, jedoch gibt es einen Mangel an formalen Modellen und automatischen Werkzeugen zur Unterstützung in diesem Bereich. Diese Arbeit präsentiert ein formales Extrahierungsmodell, das es erlaubt Empfehlungen für Microservices in einem Refactoring-Szenario algorithmisch zu generieren. Drei Strategien zur Aggregierung struktureller Information von monolithischem Quellcode werden vorgestellt. Des Weiteren wird ein Graphen-basierter Extrahierungsalgorithmus hergeleitet, der aus dem formalen Modell Empfehlungen für Microservices extrahiert. Das vorgestellte formale Model wird in einem webbasierten Prototyp implementiert. Eine Performance-Evaluation des Prototyps demonstriert zufriedenstellende Leistungscharakteristiken. Die Qualität der extrahierten Empfehlungen wird durch spezifisch definierte Metriken evaluiert. Die Experimente liefern in allen betrachteten Metriken adäquate Ergebnisse. Die durchschnittliche Grösse der Entwicklungsteams wird durch die empfohlenen Microservices mehr als halbiert. Des Weiteren entspricht die resultierende Grösse der Microservices derjenigen, die in empirischen Studien erhoben wurde und die Redundanz zwischen den Domänen einzelner Microservices wird tief gehalten.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In recent years, the software engineering community has seen a tendency towards cloud computing [BYV+09]. The changing infrastructural circumstances pose a demand for architectural styles that leverage the opportunities given by cloud infrastructure and tackle the challenges of building cloud-native applications.
An architectural style that has drawn a substantial amount of attention in the industry in this context — as for instance in [Ła15, Ric14, Fow14, Thö15, New15] — is the *microservices* architecture. This thesis aims to explore and formalize some of the most important aspects of the challenge of migrating from a monolithic (legacy) application to a microservice oriented architecture. In the following, the motivation for the topic and contribution of the thesis is presented.

1.1 Motivation

As elaborated further in Section 2.2.2, a microservice architecture has several benefits and advantages over the traditional monolithic style. An example of such a benefit is the fact that services are independently developed and independently deployable [Ric14], which enables more flexible horizontal scaling in IaaS environments and technology heterogeneity.
It is therefore no surprise that big internet industry players like Google and eBay [Hof15], Netflix [Mau15] and many others [Ric14] have undertaken serious efforts for moving from initially monolithic architectures to microservice-oriented application landscapes. What these efforts all have in common is the fact that identifying components of monolithic applications that can be turned into cohesive, standalone services is a tedious manual effort that encompasses the analysis of many dimension of software architecture views [Ric14] and often heavily relies on the experience and know-how of the expert performing the decomposition.
Identifying components to be extracted as services is not only important when moving from existing monolithic applications. It can also be used as a design methodology when building new systems from scratch:

> "You shouldn't start with a microservices architecture. Instead begin with a monolith, keep it modular, and split it into microservices once the monolith becomes a problem." [Fow14]

Newman explains the rationale behind the *monolith-first* approach with the fact that determining the correct service boundaries is very difficult and costly [New15]. Therefore it often makes sense to start with a monolithic approach until the domain model of the application has fully stabilized [New15].
In both of the aforementioned cases, the question of how to break up a system into components, which in turn are extracted as services, plays a key role [Fow14]. While refactoring and architecture processes in other software engineering disciplines such as object-oriented design are supported by numerous automated and tool-supported techniques, architects of microservices have no such option at hand when refactoring monolithic applications. Because of the typically very high complexity and maintenance costs of legacy monoliths, these refactoring efforts are very demanding and challenging [SRK+09]. Extracting microservices from such monoliths not only demands technical experience by the architect, but also requires domain knowledge of the business area. This indicates a great need for tool-based and automated support in these refactoring scenarios. Hence, this thesis aims to collect available microservice extraction know-how and heuristics from research and industry, and formalize it with the goal of building a tool prototype that enables engineers to identify extractable microservices in monoliths in a semi-automatic, tool-supported manner.

1.2 Contribution and Research Questions

As of today, there exists no formal collection or model that captures strategies that can be used to algorithmically analyze monoliths with the goal of microservice extraction. Therefore, the goals of this thesis are manifold.
The first goal is to find a formal extraction model that acts as a generic approach for the extraction of microservices from static information of monolithic code bases. It is a goal of this thesis to provide an approach that is easily automatable so that refactoring effort and interaction of the architect that performs the extraction is minimized. Therefore, dynamic methods relying on runtime analysis of the monoliths are not considered. Hence, the first research question is as follows:

> **RQ1: What is the design of a formal extraction model that uses static information to extract microservices from monolithic code bases?**

The first research question can further be divided into more specific sub-goals. One of those goals is to find and compile a collection of formal strategies or heuristics that can be used to mine static information of monolithic code bases:

> **RQ1.1: What formal strategies can be constructed to mine monolithic code bases for information that helps in the extraction of microservices?**

The other partial goal is then concerned with the question of how to make use of the information structures generated by the strategies in order to create recommendations for microservices to be extracted:

RQ1.2: What algorithm can use the information aggregated by the extraction strategies to extract microservice candidates?

The formal extraction model presented in this thesis is the contribution resulting from RQ1. The second contribution of the thesis builds on the formalisms resulting from RQ1. The formal model is validated by designing and building a research prototype that takes a monolithic code base as an input and automatically detects the parts that are suitable for extraction as microservice candidates. Thus, the second research question is:

RQ2: How can we build a research prototype that implements the formal extraction model and automatically detects candidates for microservice extraction?

The second research question is further divided into two sub-questions so that the prototype can be evaluated properly:

RQ2.1: What is the performance of the implemented prototype with respect to execution time?

To that end, an evaluation is designed with a specific set of sample code bases from open-source projects and a series of performance measurement experiments is conducted to answer RQ2.1.

RQ2.2: What is the quality of the microservice recommendations generated by the prototype?

Research question RQ2.2 is tackled by defining a set of microservice-specific metrics that can be automatically measured in a large scale manner. The sample projects are evaluated with respect to those metrics in order to give an answer to RQ2.2.

1.3 Thesis Outline

The thesis is structured as follows:

- Chapter 2 lays out the background information on the topics of monolithic and microservice architectures and elaborates on their characteristics, advantages and drawbacks. Special attention is given to the definition of the term *microservice*. Furthermore, the relations and implications between concepts like *DevOps*, *cloud computing*, *continous delivery* and *scalability* are explained.
- Chapter 3 provides a brief review of related research work that ties to the topic of this thesis. The related work is found mostly in the areas of system decomposition, program comprehension, software maintenance and reverse engineering. Prior attempts at solving similar problems in related areas are presented and potentially promising techniques and approaches from other software engineering research fields are listed. Also, the recent research on microservices in general and decomposition and extraction in particular is reviewed.

- Chapter 4 presents the basic formal approach that is used to tackle the research questions introduced above. The formal notion of a monolith and microservices as interpreted in this thesis are introduced. The graph representation used to capture structured information about the monolith to be decomposed is introduced.
- Chapter 5 lists the extraction strategies used to aggregate information from monoliths in a structured way so that the extraction model presented in Chapter 4 can utilize the information to recommend microservice candidates. Rationale and related work behind the strategies is outlined. Furthermore the formal definition of the strategies is given, together with a short running example for each of the strategies.
- Chapter 6 elaborates on the technique that is used to cut the graph representing the original monolith down into cohesive connected components that correspond to microservices. The most important parts of the algorithm are outlined in pseudo code, and an asymptotic complexity analysis is given.
- Chapter 7 explains the architectural design of the web application prototype that implements the presented model. Furthermore, important and specific details of the implementation of the strategies are described and a complexity analysis for the implementations is given.
- Chapter 8 aims to validate the presented extraction approach and the implemented prototype by running a series of performance and quality experiments on a list of sample repositories. Specifically defined metrics are used as proxies for different aspects of the achieved recommendation quality.
- The thesis is concluded by Chapter 9, where a summarization of the main outcomes is given and a brief look at limitations of the work and possible future improvements is given.

Chapter 2

Background

Unlike most other terms and concepts in the technical disciplines, monoliths and microservices have no clear, distinct definition in the formal sense. Therefore, this chapter introduces the terms and lays out background information on the respective areas.

2.1 Monoliths

A traditional (enterprise) web application back-end component is a prime example for a monolithic application [Fow14]. Monoliths usually consist of a single logical module or program that runs in a single process or executable. A monolith mostly implements a complex domain model with multiple separate domain entities and relationships among them. It is built as a single unit and therefore a change to one subpart of the monolith requires the entire application to be rebuilt and redeployed [Fow14, New15]. Also, the entire monolith is usually built on the same technology stack and programming language. An exception to this characteristic are monoliths written in one of the JVM-based[1] languages and frameworks, which enables the monolith to use different languages, as long as they run on the underlying JVM.
As a consequence of the singular nature of monoliths, scaling a monolithic application (*e.g.*, in an Infrastructure-as-a-Service (IaaS) environment) is achieved by replication of the entire unit on multiple instances behind a load balancer [VRMB11]. This approach is often referred to as *horizontal scaling*. This scaling method is fairly simple to employ for monoliths, but it comes with the drawback of being very coarse-grained: If a single subpart of the monolith experiences overloading, the entire monolith needs to be replicated, thus potentially wasting resources.
Although monoliths are singular, contained units, it is important to note that monoliths exhibit internal componentization [Fow14]. Monoliths are designed - if done properly - with low coupling and high cohesion as a design strategy [RV04] and hence have internal modules or components that are decoupled from each other using language-provided techniques such as methods, classes, packages or namespaces [Fow14]. Internal communication between the components in a monolith occurs through method invocation or function calls. Still, fault isolation in monoliths is difficult; if a submodule of a monolith fails, the entire application fails.

[1] JVM: Java Virtual Machine [SSB12]

Monoliths are in principle fairly simple to build and develop, since there are numerous frameworks available that support fast prototyping and productivity, especially in a traditional three-tier web-environment. Also, most integrated development environments (IDEs) are designed with the development of monoliths in mind.
In general, monoliths are the right approach for many standard problems. But they come with a lot of drawbacks when it comes to scalability and fault tolerance. Trying out new technologies that differ from the stack in use within a monolith is nearly impossible. Over their life cycle, monoliths tend to grow in an organic manner. It is non-trivial to fight this entropy while adding functionality. Hence, very often, monoliths end up in a situation where engineers are hesitant to introduce major refactorings or structural changes that would be beneficial to the design in fear of breaking critical parts.

2.2 Microservices

> " Gather together those things that change for the same reason, and separate those things that change for different reasons." (Single Responsibility Principle) [Mar02]

2.2.1 Definition

Literature review shows that the term *microservice* is not formally defined [Fow14, Thö15]. Newman describes microservices as small, autonomous services that work together and are focused on doing one thing well [New15]. This implies that a microservice focuses on a concise, specific, and clearly defined section of the problem domain. Evans introduced the concept of *bounded context* [Eva04]. A bounded context contains the domain entities that are relevant only to that context, and shares only the entities that are needed for communication with other bounded contexts in the domain. Literature on microservices [Fow14, New15] suggests building services according to the identified bounded contexts in the domain, allowing for cohesive and decoupled services that concentrate delivering the resources or functions related to their respective bounded context only.

The microservice architecture composes one single large application from a suite of microservices which communicate with each other through lightweight mechanisms such as HTTP or remote procedure calls [Fow14]. In essence, it is a new form of componentization: A component is not a class, package or library but an independently deployable service that runs in its own processes.
Microservices take the traditional software architecture principle of *loose coupling and high cohesion* [LJK+01] to the extreme. As a consequence, the communication mechanisms are as lightweight as possible and carry no business logic – this principle is sometimes also referred to as *smart endpoints and dumb pipes* [Fow14].

2.2.2 Benefits

Microservices have attracted high attention in industry because of the several advantages they provide over monoliths. In the following, some of those benefits are outlined.

Independently Scalable and Deployable

The fact that a microservice is independently deployable and hence independently scalable enables a much more fine-granular and efficient horizontal scaling of applications where only certain services are scaled up or down depending on their respective load instead of scaling up the entire monolith. Propagating changes into a production environment is much easier and less prone to risks and failure, since only the concerned microservice goes through the deployment pipeline. Ultimately, this leads to a much faster reaction time upon changes in the business environment [New15] and a higher velocity when correcting errors.

Heterogeneous Programming Languages

Since the services in a microservice architecture are decoupled from each other as much as possible, it is feasible to write different microservices in different programming languages and technology stacks and still compose a harmonic application. This allows the teams to use the right tool for the job depending on the domain of the service they are currently working on – *e.g.*, using a low level language such as C for performance-critical components and Java for enterprise business logic. Also, it enables a much more effortless replacement of a service by a service written in an entirely different programming language, thus eliminating any long-term commitment to one technology. Replacing or rewriting an entire microservice is much less involved than replacing a component of a monolith.

Module Boundaries and Interfaces

Microservices have very firm module boundaries. The modularity of an application composed of microservices is therefore much more stable or stringent since it is harder to move around code across service boundaries [Fow14]. A change to a microservice should usually have no impact whatsoever on the other services and components in the application.
Since services act as components, the public interfaces of such components are much more explicit [Fow14]. The public interface is defined by the set of communication endpoints that the service provides to the outside world. In contrast, most programming languages don't have good mechanisms for explicitly defining public interfaces [Fow14].

Resilience and Fault Isolation

A microservice architecture can drastically improve the resilience of an application compared to a monolithic structure. Resilience can be described as the ability to successfully accommodate unforeseen environmental perturbations or disturbances [Lap08]. An example of such an unforeseen event can be the failure of a subcomponent due to erros in the code or invalid state. In a

monolithic environment, the entire application often runs in the same process on the same machine or instance. If one of the parts of the monolith experience a failure, the whole application fails and is hence unavailable to respective users. In resilience engineering, a well-known pattern to protect systems against failures of subparts that cascade throughout the entire application is the *bulkhead* pattern [Nyg07]. As its name suggests, it follows the principle of watertight bulkheads in a ship's hull that prevent water from flooding additional segments of the ship in case of water ingress. Microservices are a prime example for an implementation of the bulkhead pattern [New15]. They divide the system into separated segments across which occuring failures can not propagate. Hence, failures in a microservice remain local to the affected microservice. This aspect makes it much easier to isolate errors in a system.

Microservices and Organization

According to Conway's law [Con68], organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations. In software engineering, evidence of Conway's law can be observed when teams are divided into database architects, server-side engineers and front-end developers and consequently those teams end up building three-tiered applications consisting of a persistence layer, business layer and presentation layer. Moreover, in a traditional development organization, software engineers are responsible for the development, while operations teams are responsible for the deployment and operation of the application in production. All these separations – database architects vs. back-end engineers, development vs. operations – cause friction and communication overhead, increasing feedback time and reducing velocity when fixing errors. In this context, the microservices practice and the DevOps paradigm [BWZ15] propose small cross functional teams that are organized around business or domain capabilities, and hence are responsible for one service [BHJ16]. By consequence, a team combines people with all different skill sets needed for the end-to-end devlopment of a service. The team includes front-end developers, back-end engineers, database architects and operation or DevOps engineers. This approach has the advantage that it minimizes the interaction with other teams and reduces the communication overhead [Fow14].

2.2.3 Challenges

Despite all the benefits mentioned in the previous section, microservices are not the holy grail of software engineering. All of the mentioned advantages only take full effect if microservices are designed properly. Also, they come with some difficulties and challenges, which shall be outlined in the following.

Expensive Communication

In contrast to monoliths, all inter-component communication in a microservice architecture is achieved by network protocols. Those remote calls are more expensive and tend to have higher latency than built-in function calls or method invocation as used in monoliths. Furthermore, designing communication APIs for each microservice can be a challenging task. Simply replacing

method invocations or function calls by remote network equivalents is not an option when transforming a monolith into a suite of microservices, since it may lead to chatty protocols that cause unnecessary overhead [Fow14].

Complexity

A microservice architecture carries some additional complexity. First, the application landscape has much more moving parts, and hence much more operational complexity [Ric14], thus requiring a high degree of automation in the areas of continuous deployment, delivery, and monitoring. Another known challenge when building microservices is *service discovery* [New15]. Horizontal scaling is one of the main advantages of microservices, and hence, engineers designing and employing microservices to build scalable systems will face the challenge of finding (network-) locations of specific services in a dynamic environment where different services scale up and down independently and may change their host frequently. Developers have to deal with the complex task of providing service discovery mechanisms for their systems, either via using one of the several publicly available solutions (such as ETCD [2], Consul [3] or Netflix's Eureka [4]), or by developing a custom service discovery mechanism.
The task of writing higher level tests, such as integration tests or end-to-end tests of features, is further complicated by the distributed nature of microservices. Testing a service requires stubbing of all the downstream services used by the service-under-test [New15] and ensuring the system-under-test uses the correct stubs.

The CAP Theorem

In addition to the challenges that were outlined in the previous section, the distributed nature of microservices causes issues that can be explained by the CAP Theorem [GL02]. The CAP Theorem [GL02] states that, in a distributed system (such as a microservice architecture), one can only achieve two of the three requirements *consistency*, *availability* and *partition tolerance* simultaneously. This implies practical problems when dealing with microservices: In an environment such as IaaS, where microservices can be horizontally scaled – potentially in different data centers – there is a need for synchronization mechanisms between the data storages of the replicas of said horizontally scaled microservices. If the data storages lose connection, and thus cannot synchronize, the system is in a so-called AP-state [New15], meaning it is still available despite network partitioning. But we have sacrificed consistency, since the data in the data storages are not congruent anymore. Analogous examples can be found for a system in CP-state (*e.g.*, sacrificing availability). Microservices are inherently affected by this theorem, and engineers need to consider it before employing microservices.

[2] ETCD: `https://github.com/coreos/etcd`
[3] Consul: `https://www.consul.io/`
[4] Eureka: `https://github.com/Netflix/eureka`

Decomposition

As suggested in [Fow14, New15], applications should not be built from scratch following the microservice approach. Instead, existing monolithic code bases should be transformed into microservices by careful decomposition. Decomposing a monolith requires profound knowledge of the respective application domain in order to align the resulting services with the bounded contexts of the domain entities the that the services represent. It also involves analyzing and resolving dependencies between components and determining where to draw the service boundaries between the components of the monolith and determining the size of each service. All these tasks are inherently manual and require great amounts of experience in terms of software design and architecture. Finding the right boundaries is non-trivial even for experienced engineers. Hence, deriving formal strategies for decomposition and implementing them in a support tool – as is the goal of this thesis – may advance the state of the art in the area of microservice architecture and design.

Chapter 3

Related Work

Due to the fact that the microservice architectural style is still a recent development in the software engineering and cloud computing community, no extensive body of academic research has been formed yet. This chapter outlines the relevant existing microservice literature but also takes a glance at prior research in traditional software engineering disciplines that is relevant in terms of methodology and techniques. These disciplines include fields such as reverse engineering, system decomposition and maintenance.

3.1 System Decomposition

Decomposing software systems has been an important branch in the software engineering research discipline since the early days. Extracting microservices from monoliths is the next evolutionary form of the original challenge of decomposing systems. Parnas *et al.* were among the first to systematically investigate the criteria that should be used when decomposing systems into modules [Par72]. Parnas presents two decompositions for the same computational problem based on different criteria and performs a qualitative comparison of the resulting modularization structure with respect to properties such as changeability, comprehensibility, and development effort. The results indicate that a decomposition based on the processing flow of the algorithm is much less favorable with respect to the mentioned properties than the alternative decomposition based on the principle of *information hiding*. According to Parnas, every module "is characterized by its knowledge of a design decision which it hides from all others." [Par72]
In [KNS+], Komondoor *et al.* present an approach to extract independent online services from legacy enterprise business applications. Because of changed customer demand and business environment, many enterprises attempt to enable online processing of business events, in contrast to the batch-oriented transaction processing that dominated the previous decades of business computing [KNS+]. The authors note that the resulting challenge of migrating monolithic legacy applications is a labor-intensive and manual task, and that the tool support and automation in that area is not satisfactory. The paper presents a static analysis-based approach that is tailored towards the imperative nature of batch processing programs in legacy languages such as COBOL. The approach uses backwards data flow slicing to extract highly cohesive services. A qualitative evaluation of the approval is performed, finding that the resulting decomposition is precise

enough in four out of five cases.
Decomposing and modularizing systems has also been a research topic in the context of *re-engineering* of large monoliths, as for instance in [TVB12]. The authors argue that re-architecting monoliths is usually an ad-hoc process that does not follow principles or methodology. To tackle this, an structured manual technique is presented which allows to extract code fragments in a monolith that belong to a specific concern and extract them into cohesive modules. The extraction is achieved by a series of refactoring templates that are applied depending on source code context of the fragments where the concern is present [TVB12]. The resulting architecture gathers code fragments belonging to the same concern into the same class file. The effectiveness of the approach is evaluated by applying it to a monolith and measuring coupling and cohesion metrics such as CBO [1] and LCOM [2] [CK94].

3.2 Software Maintenance and Evolution

The research area of software maintenance and evolution is an insightful source for techniques and approaches for challenges such as program comprehension, change prediction and architecture recovery. Some of the techniques lend themselves to an application in the context of decomposing monoliths to microservices as well.
Coupling metrics are generally regarded as an effective basis for reengineering and remodularization of software systems [LJK+01,BDLMO10,BDLMO13]. Gall *et al.* [GHJ98,GJK03] introduced the notion of *logical coupling* between components or source code artifacts. The authors argue that identifying dependencies among code artifacts using traditional coupling and cohesion metrics has several inherent problems. One issue is the fact that dependencies between code artifacts may be hidden and implicit, *i.e.*, they may not be contained in code or documentation. Software engineers often have implicit knowledge about those dependencies [GJK03]. There is a need to make this knowledge explicit. Thus, the logical coupling metric is introduced. Logical coupling is computed based on the number of co-changes of source code entities in the repository history [RPL08]. The final logical coupling between two code artifacts is defined as the number of times the artifacts were changed at the same time in the revision history of the source code. According to [RPL08], the rationale behind logical coupling is that "entities which have changed together in the past are bound to change together in the future". This can be an indicator that said entities change for the same reasons. Following the single responsibility principle [Mar02], such entities should be gathered together. As outlined in 2.2.2, the single responsibility principle is an important design guideline when defining a microservice architecture. Hence, the logical coupling metric may provide a useful basis for a microservice decomposition strategy.
Coupling measures usually indicate structural properties of systems. But especially in the context of microservices, semantics of modules are of high importance. Decomposing a monolithic application involves knowing the semantics and domain of the application (see 2.2.3). Therefore, the notion of *conceptual coupling*, introduced by [PM06], provides a promising basis for de-

[1] CBO: Coupling between Object Classes
[2] LCOM: Lack of Cohesion of Methods

composition strategies. Poshyvanyk *et al.* [PM06] present a set of metrics based on the semantic information shared between source code artifacts. The term was coined "conceptual coupling" since it expresses the strength of conceptual similarities among artifacts [PM06]. Measuring the conceptual coupling between two code artifacts involves several techniques from information retrieval (IR). Source code elements (such as classes or methods) are seen as documents in a text corpus. As a first step, a term-by-document matrix of said corpus is generated using latent semantic indexing (LSI) [Dum04]. Next, the matrix is transformed into what the authors call semantic space, where each document is represented as a vector. This is achieved using the singular-value-decomposition (SVD). The *conceptual coupling* between two documents or code artifacts is then denoted by the cosine of the two vectors [PM06]. The measure is evaluated both theoretically and through a case study with an implemented prototype, which indicates that the conceptual coupling metrics are able to reveal effectively existing (domain) coupling between artifacts that would be overlooked using traditional coupling metrics.

3.3 Reverse Engineering

Reverse engineering is defined as the process of analyzing a subject system to create representations of the system in another form or at a higher level of abstraction [CC90]. The process for decomposing monoliths to microservices proposed in this work is very similar to reverse engineering as it includes creating higher-level representations of the system to be decomposed as a first step (see Section 4.1). Hence, reverse engineering techniques are taken into account in this related work section.

As the authors of [BH98] argue, many existing systems lack proper architectural documentation. Reverse engineering research proposes using automated tools and techniques to recover the architecture of a system. In [BH98], Bowman *et al.* indicate that recovering the architecture of a system usually means extracting the relationships between artifacts from the implementation of the system. The authors propose the use of the organization structure of development teams as a source for discovering such relationships. The approach is founded on Conway's Law [Con68], which postulates that the structure of a system will reflect the structure of the team that built it. So by analyzing the organization structure of developers, the interconnections between the subsystems can be discovered – the authors denominate this with the term *ownership architecture*. However, the proposed method has the drawback that the subsystems have to be clustered first, using any other reverse engineering technique. The authors run a small evaluation in [BH98] and come to the conclusion that studying the organization structure – *e.g.*, by mining revision control histories – performs at least as well as recovering subsystem relationships from documentation artifacts.

Stroulia [SS02] notes that the context of reverse engineering and software maintenance has shifted away from the traditional purpose of understanding the structure of a program. As more and more applications move to distributed environments such as the World Wide Web, the main objectives of reverse engineering and maintenance also move towards system migration of legacy application into distributed environments [SS02]. As a consequence, new reverse engineering

techniques and models are focused on run-time measurements of applications through dynamic analysis [SS02]. Prior work [Bal99] illustrates how analyzing the frequencies of execution paths through a program can help understanding and decomposing a program. The extraction of microservice from monoliths can be regarded as a special case of migration of a legacy system into a distributed environment that also includes decomposing an existing program. Dynamic analysis is therefore a viable method for gathering information to extract microservices from a monolith.

3.4 Microservices

General Microservices Research

As a young research discipline, the microservices field has not yet seen a large body of research work. Pahl and Jamshidi conducted a systematic secondary study and reviewed and classified the existing research body on microservices [PJ16]. In their secondary study, the authors investigated 21 studies published in the years 2014 and 2015. It is the first study of this kind for microservices. Their findings indicate that microservices research is still immature and in a formative stage [PJ16]. Furthermore, the results of the review indicate that the microservice research body needs more experimental evaluation of the presented solutions and benefits. The authors detected a specific lack of tool-support for microservices in the current state of the art. The secondary study discovered that the microservices community suggests a number of patterns for microservice migration, refactoring and architecture [PJ16]. A good part of these efforts and patterns is only conceptual and the review reveals a higher number of use case studies than technology solutions and tool support studies, a fact which the authors consider as a sign of immaturity for the research field [PJ16]. The secondary study is concluded with an outlook of coming research trends in the field. The authors name microservice migration, architectural refactoring and intelligent technologies for microservice discovery in repositories as important aspects regarding methodology and tool support [PJ16]. This shows that there is a clear research need for research efforts such as the one presented in this thesis.
One of the first studies to investigate industrial practices in services computing is [SCL15], where service computing practices in 42 companies of different sizes were empirically studied. The work has been conducted with specific attention to the recent microservice trend in service computing [SCL15]. Among the questions discussed in their survey, there were several topics with relevance to the goal of this thesis. Service size and complexity was collected. The results imply that services vary in size, but extreme values such as very small services with under 100 LOC or very large services with more than 10000 LOC are rarely encountered in practice. Another result that is of interest in the context of this thesis is the observation that services are dedicated. In other words, the respondents reported that the services they're involved in typically are dedicated to relatively narrow and concise tasks [SCL15]. This principle is revisited in the following sections on microservice background in Chapter 2.2 and in the motivation of the proposed microservice decomposition strategies in Section 5.

Migration to Microservice Architectures

Migration of legacy code bases to microservice-style architecture has been identified as an important trend in the microservices field, as described in the first paragraph of this section. A comprehensive collection of microservice migration patterns is presented in [BHJ15]. The migration patterns listed in [BHJ15] are informal conceptual refactoring suggestions that can be applied to a monolithic code base depending on the specific initial situation. These patterns include specific migration procedures for decomposing a monolith into microservices. The authors use mainly two rationales in those procedures. One relies on the concept of bounded context adapted from domain-driven design. The suggested migration procedure involves identifying concluded bounded contexts in the domain model of the monolith and create separate microservices for the identified bounded contexts. The suggested approach is also present in standard microservice architecture literature such as [New15]. Bounded contexts and the domain information on a monolith are therefore promising starting points and are further discussed in Chapter 2, 5 and 8 of this thesis.

Microservice Extraction from Monoliths

The uprise of the microservices paradigm also spawned efforts in the software engineering research community that set the extraction of microservices from existing monoliths as a goal. An example of such efforts is [LTV16]. The paper presents a systematic, manual approach to identify microservices in large monolithic enterprise systems. The authors propose the assumption that monolithic (enterprise) systems have three main parts: client side (denoted as F for *facades*), server side (denoted as B for *business functions*) and database (denoted by D). The organization O that employs the monolith has several business areas a_i. As a first step, the database tables $d_j \in D$ are each assigned to a business area a_i according to their respective domain. Next, a dependency graph between all $f_k \in F$, $b_l \in B$ and $d_m \in D$ is constructed based on static relationships such as function calls. In the last steps, the dependency graph is traversed and candidate paths from $f_k \in F$ to $d_m \in D$ are found for each database table d_k in a specific business area a_i. These paths serve as the final microservice candidates. The source code of said candidates is inspected and hence microservices are identified [LTV16]. An additional step introduces an API gateway to make the deocomposition transparent to any kind of client that uses the system. The authors perform a validation of their approach by applying it to a large banking system and qualitatively inspecting the resulting microservices. The approach provides a valuable first step in the area of monolith decomposition since it provides a somewhat formal definition of the terms *monolith*, *microservice* and precisely and semi-formally describe decomposition technique. The limitation of the work lies in the fact that some of the steps – like the mapping of database tables to business areas – is dependent on deep domain knowledge of the system under investigation and by consequence, requires manual interaction.

Another notable attempt at providing a structured way of identifying microservices in monolithic code bases is done by the authors of *ServiceCutter* [GKGZ16]. The authors present a tool that supports structured service decomposition. The internal representation of the system to be decomposed is based on a catalog of 16 different coupling criteria that were abstracted from lit-

erature and industry know-how [GKGZ16]. Software engineering artifacts and documents such as domain models and use cases act as an input source to generate the coupling values in the 16 different coupling variants. A consequence of the decision to use only documentation artifacts as an information source and not the actual application source code is the fact that the amount of interaction and domain knowledge required for the user of the *ServiceCutter* tool is higher. The approach followed in [GKGZ16] builds a graph from the coupling criteria and the artifacts that were fed to the tool. Using traditional community-detection algorithms from social network analysis, the tool then supports the architect by suggesting cuts between nodes where internal cohesion is high and external coupling is low. A small scale validation of the tool is conducted by applying it to two sample projects, one of which is a fictional application for which artifacts and documents were created specifically for the evaluation. Then, ground-truth is constructed by suggesting an ideal decomposition of the sample application based on the author's own experience. The resulting decomposition suggested by the tool is then compared to the constructed ground truth in order to rate the resulting decomposition on a three level scale. *ServiceCutter* presents an interesting attempt to formally capture aspects that play a role when identifying services in monolithic environments. The way source information is supplied to the tool is a drawback because the heavy-lifting in terms of information aggregation about the monolith is done by the user. A very promising aspect of this work is the use of social network clustering algorithms to identify densely connected components.

3.5 Research Gap

The research gap that this thesis attempts to close is mainly twofold. On one side, there is a large body of prior work on traditional software decomposition and modularization approaches, as indicated by the short summaries in the previous sections. These approaches include a great variety of techniques from the use of traditional software metrics, utilization of repository mining and information retrieval or static analysis techniques such as slicing. Among all those previous solutions, there are none that are specifically designed for an application in a microservice extraction or monolith refactoring scenario.
On the other hand, there exists a small and recent body of work on the topics of microservice migration, extraction of microservices from monoliths or decomposition of monolithic applications. Among these attempts at tackling the microservice extraction problem, there are almost none that provide a fully automatable or formal approach that allows to be implemented in support tooling. *ServiceCutter* [GKGZ16] presents a promising formal graph framework to capture monoliths and compute cuts to decompose the monolith into service candidates, but it lacks the means of recovering the necessary graph construction information from the actual monolith. Instead, it relies on the software architect to manually define use cases and domain models in a specific expected model in order to parse information about the structure. Therefore, the quality and precision of the coupling graph will vary depending on the user's input. Other microservice extraction methods revisited in the previous sections of this chapters are limited to conceptual migration and refactoring patterns that are mostly non-formal and therefore difficult to implement as au-

tomated tool support. Furthermore, the works mentioned in the previous section mostly lack a quantitative evaluation. The validation of the approaches is limited to case studies on specific monolithic applications and qualitative manual inspection of the resulting microservice decompositions.
This thesis aims to build on the best of both worlds – traditional reverse engineering and decomposition techniques on one side and microservice extraction approaches and design principles on the other side – to close the illustrated research gap. To that end, as a first contribution, an extraction model that is able to properly abstract the problem of decomposing monoliths to microservices is presented. The second contribution of this thesis is the derivation and definition of extraction strategies motivated by microservice-specific properties and guidelines. Finally, quantitative metrics specific to the microservice recommendation use case are defined to evaluate the presented work through a series of experiments on a sample set of monoliths.

Chapter 4

Extraction Model

This chapter proposes a formal graph-based extraction model as a solution to **RQ1**. Graph-based representations of software, source code and artifacts has been used as a technique in prior research [GJK03, GKGZ16]. The model presented in the following sections utilizes the insights of previous employments of graph-based models such as in [GKGZ16] to extend them to an extraction process that involves notions for the information aggregation from source code and a subsequent microservice extraction based on this information.

4.1 Basic Extraction Process

All of the extraction strategies outlined in more detail in Chapter 5 are embedded into a generic extraction process. This extraction process comprises of three extraction stages or representations: the *monolith* stage, the *graph* stage and the *microservices* stage. There are two transformations between the stages: The *construction* step transforms the monolith into the graph representation, and the *clustering* step decomposes the graph representation of the monolith into microservices. The transformations performed during the construction step differ according to the extraction strategy in use (cf. Chapter 5). The algorithm performing the clustering step is explained in detail in Section 6.

4.2 Construction

The starting point is always an actual code base or repository of an implemented application from some form of version control system such as Git [1]. The aspects of a code base that are important to the extraction process are captured by the *monolith* representation. A monolith M is a triple

$$M := (C_M, H_M, D_M)$$

where C_M is the set of class files, H_M is the change history and D_M is the set of developers that

[1] `https://git-scm.com/`

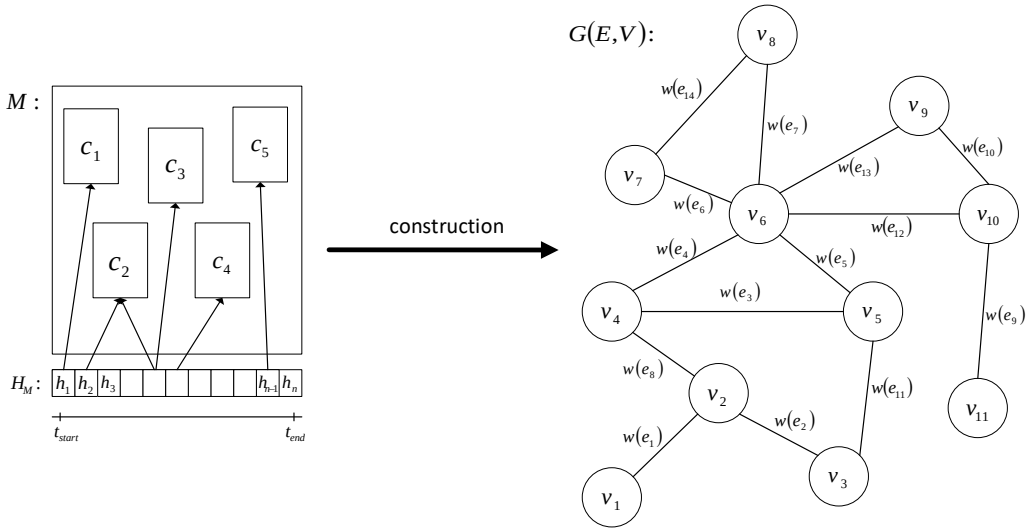


Figure 4.1: Construction step from the monolith M to the graph representation G. The construction step utilizes coupling strategies to transform the monolith M into a undirected, weighted graph G.

contributed code to the monolith M. Each class file $c_k \in C_M$ has a file *name*, a file *path* which is assumed to be unique in the monolith M and file *contents* in the form of text.

The change history H_M of a monolith M is defined as an ordered sequence of *change events*:

$$H_M := (h_1, h_2, \ldots, h_n)$$

Each change event h_i is defined as a triple $h_i := (E_i, t_i, d_i)$, where E_i is a set of classes from the set C_M that were added or modified by the change event h_i, t_i is the *timestamp* at which the change event occurred, and $d_i \in D_M$ is the developer that committed the corresponding change.
The *monolith* is transformed into the *graph representation* by the *construction* step, as illustrated in Figure 4.1. The construction step employs one of the *coupling strategies* presented in Chapter 5 to mine the information in the monolith and create an undirected, edge-weighted graph G. In the graph $G = (E, V)$, the vertices $v_i \in V$ each correspond to a class file $c_i \in C$ from the monolith. Each graph edge $e_k \in E$ has a weight defined by the *weight function*. The weight function is defined as:

$$w : E \mapsto \mathbb{R}$$

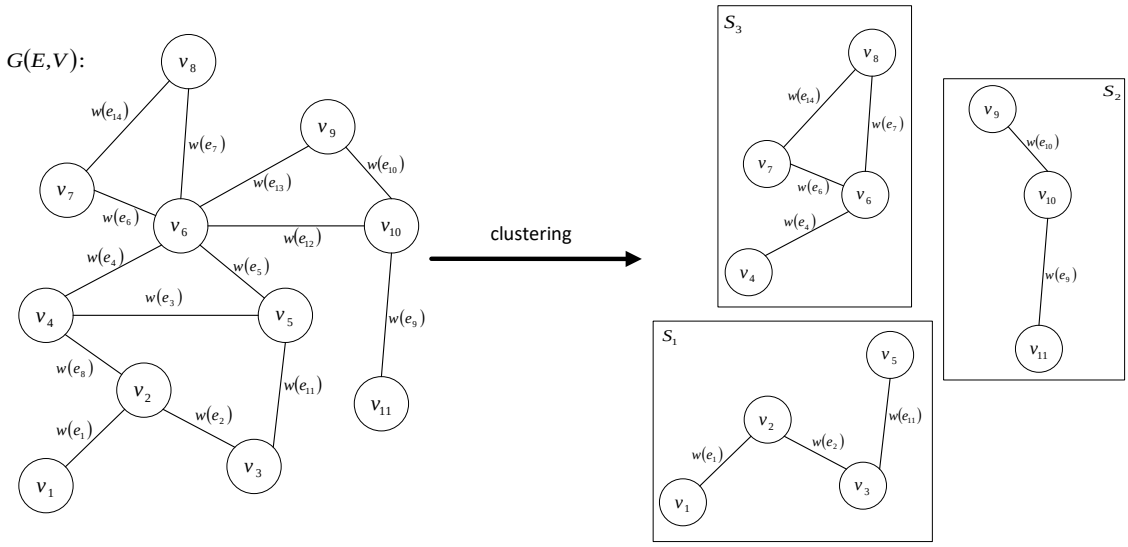


Figure 4.2: The clustering step decides which edges of the graph to delete in order to compute microservice extraction of the original monolith M.

The weight function determines how strong the coupling between the neighboring edges is according to the coupling strategy in use. A higher weight value indicates a stronger coupling. Note that not all classes $c_k \in C$ from the original monolith will result in a corresponding vertex $v_i \in V$. There may be classes that do not exhibit any coupling to any other class when using the coupling strategies, which would lead to the class being discarded from further processing in the graph.

4.3 Clustering

The graph representation is then cut into pieces to obtain the candidates for microservices. This is achieved by the *clustering* transformation using the graph clustering algorithm outlined in Chapter 6.

The clustering results in the *microservice* recommendation D_M for the monolith M. Formally, a microservice recommendation is a *forest* D_M consisting of at least 1 and at most N *connected components* S_j. A connected component S_j in the forest is referred to as a **microservice**, while N denotes the number of microservices in the recommendation D_M.

Each microservice S_j is a connected, directed graph $S_j = (C_{S_j}, R_{S_j})$, where the vertices of the graph consists of the set C_{S_j} of classes in the microservice S_j. The set C_{S_j} is a proper subset of the

set of classes in the monolith: $C_{S_j} \subset C$. The set R_{S_j} denotes the set of relations or edges between the class nodes $c_k \in C_{S_j}$.

Chapter 5

Extraction Strategies

As outlined in Chapter 3, the fields of software maintenance, reverse engineering, program comprehension and decomposition provide numerous options and techniques that lend themselves to be applied in the context of microservice extraction. Generally, the techniques can be divided into static and dynamic approaches.
Dynamic analysis measures program properties during execution time. As noted in the related work section 3.3, dynamic analysis is already established as a tool for the migration of legacy systems to distributed environments and as an aid in software decomposition. Both of these tasks also play a role when extracting microservices from a monolith. Dynamic analysis techniques are particularly well suited in this case because some of the main driving factors behind the concept of microservices are inherently dynamic in nature: *independent scalability* and *fault isolation* (see Section 2.2.2). By taking those two design goals as guidelines, it is possible to devise strategies that employ dynamic analysis techniques to extract microservice candidates from a monolith. An example strategy would be concerned with the varying resource consumptions and execution times of different parts in the monolith, thus extracting services based on CPU, I/O or network utilization. Another strategy can be constructed based on dynamic program flow paths from endpoints towards the back-ends of the monolith. In that case, one might combine parts of the monolith into a microservice if the parts are frequently on the same dynamic program flow paths.
Despite the apparent eligibility of dynamic analysis techniques for the use case at hand, there are serious caveats. Taking dynamic measurements of programs during run time requires that the application under investigation is at hand in an executable form. This implies that the program builds successfully, has all the third party dependencies that are needed and has sufficient documentation or scripting to run the program and any necessary container or server software. But having an executable program alone is not enough. The program must be targeted with load (e.g. requests to a web application). The load data should represent the conventional load that the program usually experiences in a real-world environment. Measuring properties during run time requires prior injection of measurement infrastructure into the software, a task commonly referred to as *program instrumentation*. All these issues highlight the fact that dynamic analysis is highly involved in practice and that the space of potential programs one can run dynamic analysis on is greatly reduced by the above-mentioned requirements. The goal of **RQ1.1** is to formalize extraction strategies that are applicable to a wide range of candidate projects and support as much

automation as possible. These goals contradict with the limitations posed by the requirements of dynamic analysis and hence, this thesis does not conduct further investigation of dynamic analysis strategies.
Static analysis usually involves building the abstract syntax tree (AST) of the program under inspection and then running the desired form of analysis on the AST. From a software maintenance and evolution perspective, software metrics [CK94] play an important role in static program analysis, especially when dealing with object-oriented programs. Software metrics and static analysis techniques exhibit several advantages. Software metrics are well studied in the research field and are established tools in the software maintenance portfolio. For most of the well-known metrics and techniques, implementations exist in multiple programming languages. Research indicates that metrics are a good predictor for software defects [NBZ06] and help in identifying poorly designed software artifacts during maintenance [KR87]. But in the context of microservice extraction, the task is different from defect prediction or design issue detection. It is essentially a *recommendation* problem: The approach should recommend a set of microservice candidates to extract given the original monolith. If heavily simplified, this recommendation problem can be broken down the question: Which classes belong together in the same microservice? Traditional software metrics [CK94] and static analysis fail to provide assistance or valuable information in such a case. Assume that class c_1 and class c_2 from a monolith M are tightly coupled to each other – by *coupled* we mean coupling as in CBO (coupling between object classes) [CK94] in this case. What does this tight coupling between c_1 and c_2 imply if we want to recommend microservice candidates? Is this an indicator that c_1 and c_2 belong to the same microservice? The high coupling value between the two classes can be an indicator of poor design but it also might be intentional. This simplified example illustrates that metrics which investigate only the static *structure* of a program are not well suited for a microservice extraction use case. Static approaches employed in our use case should therefore also include information sources external to the actual code structure of the monolith. Therefore, the strategies that were designed in this thesis rely on information such as the revision history of the monolith (*logical coupling strategy*), the team structure (*contributor coupling strategy*) and semantic information (*semantic coupling strategy*).

5.1 Logical Coupling

The *Single Responsibility Principle* [Mar02] states that a unit of software componentization should have only one reason to change. By consequence, software design that follows the principle should gather together the software elements that change for the same reason [Mar02].
Furthermore, one of the main benefits and design goals behind the concepts of microservices is to enforce strong module boundaries [Fow14]. Strong module boundaries and adherence to the Single Responsibility Principle provide benefits in case of a change: If a developer has to make a change to a system, they only need to locate the module to be changed and only need to understand that confined module. In contrast, in a software design where responsibilities are spread across the entire system, identifying and understanding parts that are involved in changes is much more involved.

What both of the above-mentioned arguments have in common is the fact that they are founded on the *change* of software elements such as class files. Hence, in order to obtain microservices that provide the mentioned benefits, it is essential to analyze the changing behavior of the original monolith. Class files that change together should consequently also belong to the same microservice. This constitutes the rationale behind the *logical coupling strategy*, which is defined in the following sections.

5.1.1 Definition

Above, it was noted that files and classes that *change together* should be considered as an information source for microservice extraction from monoliths. But what does *changing together* mean formally in the concept of software engineering and maintenance?
Gall *et al.* coined the term *logical coupling* as a retrospective measure of implicit coupling based on the revision history of an application source code [GJK03]. In the original formulation, the history of an application is defined as an interval of fixed-length sessions [GJK03, RPL08]. Recall the definition of the *change history* H_M of a monolith M from Section 4.2. Each predefined fixed length session from [GJK03] maps to a *change event* $h_i \in H_M$ as defined in Section 4.2. That means each for each change event $h_i := (E_i, t_i, d_i)$ the set of changed classes E_i contains all class files that were changed *during* the predefined session starting at t_i.
In this thesis, only changes that *add* a new file or changes that *modify* files without deleting them are considered during the computation of the logical couplings.
Let the classes $c_1, \dots c_n \in E_i$ be the pairwise distinct classes that were changed in that respective change event.

$$l \neq k \implies c_l \neq c_k$$

Let δ be a function that takes two classes as arguments. Function δ indicates whether the two distinct class files c_1, c_2 have changed together in a certain commit $h_i \in H_M$:

$$\delta_{h_i}(c_1, c_2) = \begin{cases} 1 & \text{if } c_1 \text{ and } c_2 \text{ were modified in commit } h_i \\ 0 & \text{else} \end{cases}$$

Now let Δ be the *aggregated logical coupling* for the class files $c_1, \dots, c_n$:

$$\Delta(c_1, \dots, c_n) = \sum_{h \in H_M} \delta_h(c_1, \dots, c_n)$$

In the construction step, the strategy employs the aggregated logical coupling Δ to compute the weights on the edges of the graph G. Let $e_l = (c_i, c_k)$ be an edge on the graph G representing the original monolith. Then the weight $w(e_l)$ is computed by:

$$w(e_l) = \Delta(c_i, c_k) \tag{5.1}$$

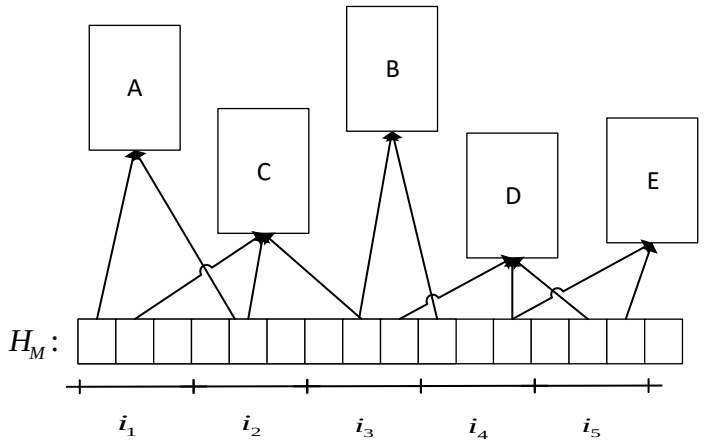


Figure 5.1: Example Monolith for Logical Coupling Computation. The rectangles A - E denote class files, H_M represents the change history, and i_1 - i_5 represent the history intervals. An arrow from a history interval to a class file indicates a modification to that file during that interval.

The graph G is constructed by computing edge weights as in Section 5.1 for all pairwise distinct class files $c_k \in C$ of the monolith $M := (C_M, H_M, D_M)$.

5.1.2 Example

To illustrate the computation of the logical coupling strategy on a hands-on example, let's imagine a monolith with a change history as depicted in Figure 5.1.

The small segments on H_M represent commits, while an arrow from a commit box towards a file indicates that the corresponding file was changed in that commit. The computation iterates through the intervals i_1 to i_5 and creates coupling weight pairs for all pairs of files that were changed at each interval. Let $\Delta(A, B)$ denote the current coupling value between two files A and B. For the current example the iteration would look as follows:

i_1: $\Delta(A, C) = 1$

i_2: $\Delta(A, C) = 2$

i_3: $\Delta(B, C) = 1$ and $\Delta(B, D) = 1$ and $\Delta(C, D) = 1$

i_4: $\Delta(B, D) = 2$ and $\Delta(D, E) = 1$ and $\Delta(B, E) = 1$

i_5: $\Delta(D, E) = 2$

For all of the encountered class pairs, the latest coupling values are taken over into the graph representation as weights on the edges between the corresponding files. In the case of the example, the resulting graph is depicted in Figure 5.2.

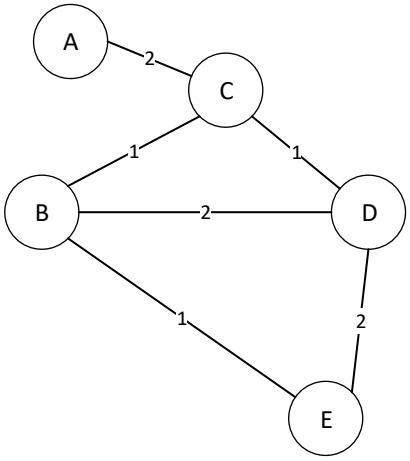


Figure 5.2: Resulting coupling graph for the example presented in Figure 5.1

5.2 Semantic Coupling

In Section 2.2.1, the notion of bounded context from domain-driven design was introduced as a possible design rationale for microservices and their boundaries. According to that rationale, each microservice should correspond to one single defined bounded context from the problem domain. This results in scalable and maintainable microservices that focus on one responsibility. So it is desirable to infer a extraction strategy from said concept. The issue with this design principle is that it is inherently informal, requires manual expert know-how and thus is hard to capture in a formal strategy. One possibility from prior work that enables to formally identify entities from the problem domain is by examining the contents and semantics of source code files through information retrieval techniques [MV99, MM01, KDG07, PM06].
Basically, the strategy should couple together classes that contain code about the same "things" – things as in domain model entities. As prior work [MM01, KDG07] has shown, identifiers and expressions in code (variable names, method names and so on) can be used to identify high level topics or domain concepts in source code. The semantic coupling strategy was constructed to use these expressions and identifiers as input with the term-frequency inverse-document-frequency method [Ram03]. It computes a scalar vector for a document given a predefined set of words. By computing these vectors for class files in the monolith and then computing the cosine similarity between vectors of pairwise distinct classes, the semantic coupling strategy can compute a score that indicates how related two files are in terms of domain concepts or "things" expressed in code and identifiers. Note that despite relying on the same rationale and motivation as the conceptual coupling [PM06] mentioned in Chapter 3, it uses different computation techniques.

5.2.1 Definition

The semantic coupling strategy looks at all pairwise distinct classes $c_i, c_j \in C_M$ in a monolith $M := (C_M, H_M, D_M)$ with $i \neq j$. For each pair of classes c_i, c_j the procedure described below is executed.

Tokenization
First, each of the class files tokenized, which results in a set of words $W_j = \{w_1, w_2, \ldots, w_n\}$ for each class c_j. During the tokenization process, all special characters and symbols specific to the programming languages used in the classes will be filtered out. The set W will now contain all identifiers from the code of the respective class.

Stop Word Removal
There are words or identifiers in class files that are not related to any actual domain concept or entity of the application, but are identifiers that belong to the programming language or framework in use. Examples of such words are `public`, `class` or `package` in the Java programming language. Examples for a framework-specific word are `controller` or `view` in Rails, or `model` in Django. In the next step, these words are filtered out of the set W_j, W_i for both of the class files c_j, c_i.

Term Extraction
In the next step, the *term list* $T = \{t_1, t_2, \ldots, t_k\}$ is constructed by combining all the words $w_l \in W_j$ from the first class with all the words $w_l \in W_i$ in the second class. This term list serves as the basis for the computation of the tf-idf vectors for each class.

Vector Computation
The procedure continues to compute a vector $V \in \mathbb{R}^n$ for the word list W_j for the class c_j and analogously a vector $X \in \mathbb{R}^n$ for the class c_i and its word list W_i. The dimension n of the vectors V and X is equal to the size of the term list T computed in the previous step. The k-th element of the vectors is formed by computing the tf-idf (term-frequency inverse-document-frequency) value [Ram03] of the k-th term in the term list T with respect to the corresponding word list.
X is therefore computed by

$$\forall\, t_k \in T:\ x_k = tf(t_k, W_i) * idf(t_k, W_{all})$$

and analogously for V

$$\forall\, t_k \in T:\ v_k = tf(t_k, W_j) * idf(t_k, W_{all})$$

The variables v_k and x_k in the above equation denote the k-th element of vectors V and X respectively. W_{all} is the set of all word lists of all class files in the current computation – in this example $W_{all} := \{W_i, W_j\}$. In the following, the details of the functions $tf(\ldots)$ and $idf(\ldots)$ are outlined.

Term Frequency
The *raw term frequency* $f(t_k, W_i)$ of a term $t_k \in T$ with respect to a document or list of words W_i is defined as the number of times that the term t_k is found in W_i [LTSL09]. To avoid corruption of the result caused by unusually high raw frequency of a term in a specific document, the *logarithmic term frequency* [LTSL09] is used as a term frequency measure for the semantic coupling strategy. It is defined as:

$$tf(t_k, W_i) = \begin{cases} 0 & \text{if } f(t_k, W_i) = 0 \\ 1 + log_e(f(t_k, W_i)) & \text{else} \end{cases} \tag{5.2}$$

Inverse Document Frequency
The *inverse document frequency* measures how rare or common a given term t_k is across a set of documents or word lists $W_{all} := \{W_1, W_2, \dots\}$. Assume that n is defined as the number of documents / word lists a term t occurs in:

$$n(t) = \{W_i \in W_{all} : t \in W_i\}$$

The inverse document frequency is then defined as:

$$idf(t_k, W_{all}) = log\left(\frac{|W_{all}|}{n(t_k)}\right) \tag{5.3}$$

Similarity Computation
After the vector computation using the tf and idf measures, there are two vectors X and V, one for each word list of the respective class. By using the cosine between the two vectors, the similarity of the two original classes with respect to the semantics of their identifiers and contents can be computed. Let $e_k \in E$ be the edge between the class files c_i and c_j in the graph G representing the original monolith M. The weight $w(e_k)$ of the edge is defined by:

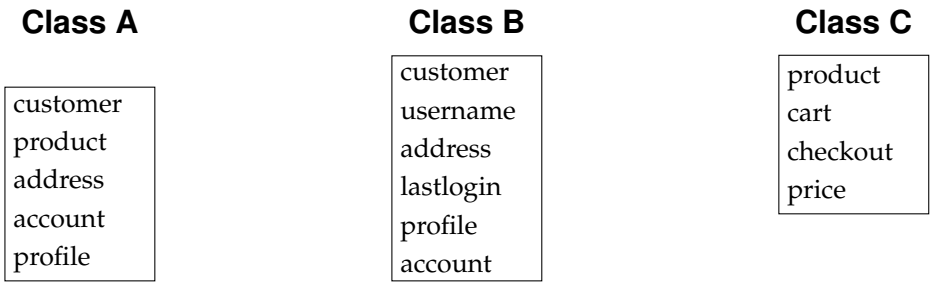
$$w(e_k) = cos(\theta) = \frac{V \cdot X}{||V||\ ||X||} = \frac{\sum_{i=1}^{n} v_i \cdot x_i}{\sqrt{\sum_{i=1}^{n} v_i^2}\sqrt{\sum_{i=1}^{n} x_i^2}} \tag{5.4}$$

where v_i and x_i are the elements of the vectors of V and X and n is the size of the vectors and the size of the term list T.

5.2.2 Example

To construct an example for the semantic coupling strategy computation, three fictional class files A, B and C are presented. The class files only contain already filtered and cleaned tokens. For the

sake of simplicity and brevity, programming language constructs such as control flow or reserved keywords are not displayed in the classes. The contents of the three classes are depicted below:



The computation is carried out by permuting all pairwise combinations of all classes in the monolith. In the case of this example, this means the pairs (A, B), (A, C), and (B, C). For each pair (A, B), the term list $T(A, B)$ is constructed and then used to compute the tf-idf vectors for the two classes of the pair.
For the pair (A, B), the term list is $T(A, B) =$ [customer, product, address, account, profile, username, lastlogin]. Using the formulas for the term frequency in 5.2 and inverse document frequency in 5.3, the tf-idf vectors for the class files A and B are computed.
Vector X contains the tf-idf values for the class A with respect to term list $T(A, B)$:

$$X = [1, 1, 1, 1.41, 1, 1, 1, 1, 0, 1, 0]$$

For the file B, the vector Y contains the tf-idf values with respect to $T(A, B)$:

$$Y = [1, 1, 1, 0, 1, 1, 1, 1, 1.41, 1, 1.41]$$

The semantic similarity coupling that corresponds to the weight on the graph edge e_{AB} between the class files A and B is then computed as:

$$w(e_{AB}) = cos(X, Y) = 0.73$$

The coupling for the pair (A, C) is computed analogously. The term list for (A, C) will obviously contain slightly different tokens: $T(A, C) =$ [customer, product, address, account, profile, cart, checkout, price]. For the sake of simplicity, the explicit values of the vectors X and Y for the class files A and C are omitted. The resulting coupling for the edge e_{AC} is:

$$w(e_{BC}) = cos(X, Y) = 0.23$$

The last pair (B, C) performs the same procedure as the other pairs, but further details of the vectors and term list are omitted. The semantic coupling on the edge e_{BC} is:

$$w(e_{BC}) = cos(X, Y) = 0$$

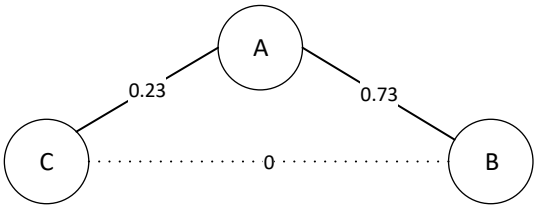


Figure 5.3: Resulting coupling graph for the classes A,B and C generated by the semantic coupling strategy. The dotted connection between nodes C and B indicates a theoretical edge with edge weight 0, so that the edge is not considered in the coupling graph.

The coupling value of 0 can also be intuitively explained, since classes B and C contain no common tokens at all. In practice, this case will be extremely rare.
The computed edge weights are then used to construct the coupling graph as in the logical coupling example before:

5.3 Contributor Coupling

As indicated in Section 3.3, prior work on reverse engineering has successfully employed team and organization information to recover relationships among software artifcats that stay undiscovered otherwise [BH98]. Other research has shown that analysis of code authorship allows to identify experts for certain software artifacts and components among the developers of a project. Related research [BH98] shows that it is possible to recover an ownership architecture from version control systems. Such ownership architectures reveal team structures and communication patterns between teams of different components of the software.
Well-organized teams are a main concern when migrating a project to microservices. Section 2.2.2 describes how the microservice paradigm proposes cross-functional teams of developers organized around domain and business capabilities. One of the main objectives of the team and organization philosophy in the microservice paradigm is to reduce communication overhead to external teams and maximize internal communication and cohesion inside developer teams of the same service.
The contributor coupling strategy aims to incorporate these team-based factors into a formal procedure that can be used to cluster class files according to the above-mentioned viewpoints. It does so by analyzing the authors of changes on the class files in the monolith's version control history.

5.3.1 Definition

Recall that in Section 4.2 a monolith was defined as a triple $M := (C_M, H_M, D_M)$ of the set of classes C_M, the set of historic change events H_M and the set of contributing developers D_M. The following procedure is applied to all class files $c_i \in C_M$ in the monolith M.

The first step for computing the contributor coupling involves finding all history change events $h_k \in H_M$ that have modified the current class c_i. Let $\gamma(h_k)$ denote a function that returns the set of changed class files E_k from the change event $h_k = (E_k, t_k, d_k)$. Then the set of change events where class c_i was involved is denoted by $H(c_i)$ and defined as:

$$H(c_i) = \{h_k \in H_M | c_i \in \gamma(h_k)\}$$

Let $\sigma(h_k)$ denote a function that returns the uniquely identifiable author $d_k \in D_M$ that contributed the change event h_k to the monolith M. Then, the set of all developers that contributed to the current class file c_i are denoted by $D(c_i)$:

$$D(c_i) = \{d_x \in D_M \mid \forall\ h_k \in H(c_i) : \sigma(h_k)\}$$

After computing the set $D(c_i)$ for all classes $c_i \in C_M$ in the monolith M, the coupling can be computed. In the graph G representing the original monolith M, the weight on any edge e_l is equal to the contributor coupling between two classes c_i and c_j which said edge connects in the graph. The weight is defined as the cardinality of the intersection of the sets of developers that contributed to class c_i and c_j:

$$w(e_l) = |D(c_i) \cap D(c_j)| \tag{5.5}$$

5.3.2 Example

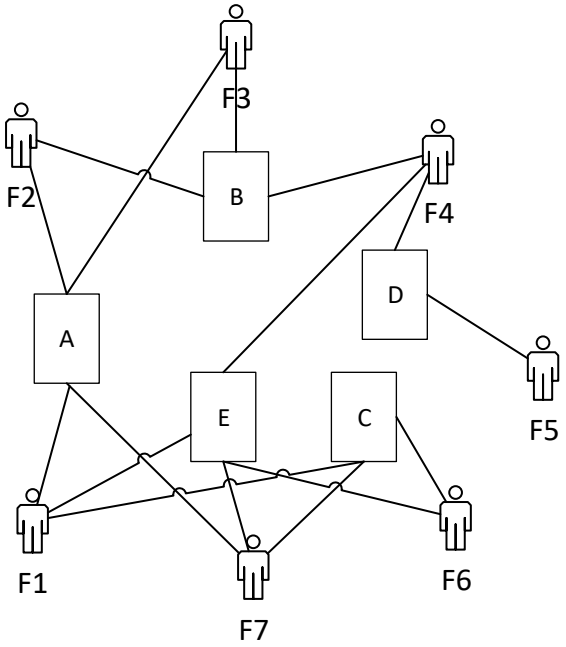


Figure 5.4: Example Contributor Situation of a Monolith Before the Contributor Coupling Computation. $F1$ to $F7$ represent the contributors of the monolith. The rectangles A to E represent the class files, and the connections between contributors and classes imply a change by that contributor on that class.

Figure 5.4 represents the initial contributor participation situation in a monolith to illustrate an example of how the contributor coupling is computed. Note that the step of recovering the developers for each file from the history H_M is omitted for simplicity. First, a mapping from the class to the set of contributors that modified it is constructed:

$S(A)$: { F2, F3, F1, F7 }

$S(B)$: { F3, F4, F2 }

$S(C)$: { F6, F1, F7 }

$S(D)$: { F4, F5 }

$S(E)$: { F1, F6, F7, F4 }

Then, all the contributor sets of the class files are compared in a pair-wise permutation to find the contributor couplings between the classes. Let $w(A, B)$ denote the resulting contributor coupling weight on the edge between classes A and B:

$$w(A, B) = |S(A) \cap S(B)| = 2$$

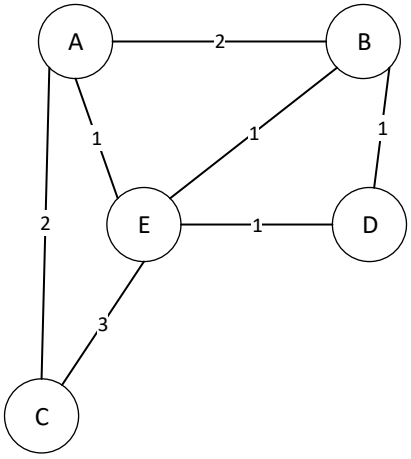


Figure 5.5: Resulting contributor coupling graph representation of the monolith from Figure 5.4

$$w(A, C) = |S(A) \cap S(C)| = 2$$

$$w(A, E) = |S(A) \cap S(E)| = 2$$

$$w(B, D) = |S(B) \cap S(D)| = 1$$

$$w(B, E) = |S(B) \cap S(E)| = 1$$

$$w(D, E) = |S(D) \cap S(E)| = 1$$

$$w(E, C) = |S(E) \cap S(C)| = 3$$

The corresponding graph that is constructed from the contributor coupling weights $w(\cdot, \cdot)$ for the example from Figure 5.4 is depicted in Figure 5.5.

5.4 Combination of Strategies

In order to exploit the potential information gain of the presented strategies to the full extent, the coupling weights from the three strategies can be arbitrarily combined to form the weights on the edges of the graph.
The strategies produce coupling values with greatly varying magnitudes. This is due to the fact that some of the strategies rely on absolute aggregation or summation of the values, while other strategies such as the semantic coupling compute normalized values in the range $[0, 1]$.
Therefore, the combination of the values to form a single weight on the graph edge needs to include weight factors to control the impact of the different strategies. Let $w(e_{i,j})$ denote the weight of the edge $e_{i,j}$ connecting the classes c_i and c_j. Furthermore let w_{LC}, w_{CC} and w_{SC} denote

the resulting coupling weight for the mentioned classes generated by the logical coupling (LC), contributor coupling (CC) and semantic coupling strategy (SC) respectively. Then the weights are combined by:

$$w(e_{i,j}) = c_{LC} \cdot w_{LC}(e_{i,j}) + c_{CC} \cdot w_{CC}(e_{i,j}) + c_{SC} \cdot w_{SC}(e_{i,j}) \tag{5.6}$$

The values of the weighting factors c_{LC}, c_{CC} and c_{SC} are given as user input.

Chapter 6

Clustering Algorithm

The construction step utilizing the strategies defined in Section 5 results in a graph representation of the original monolith M. The resulting graph $G = (E, V)$ is a undirected weighted graph. The weights are defined by the coupling strategies in use. The next step in the extraction process is to cut the graph G into (connected) components that will represent the recommended microservice candidates. Formally, this means deleting edges $e_k \in E$ from the graph in such a manner that the remaining connected components converge to a satisfactory result. The partitioning of the graph must take the weights $w(e)$ on the edges into account when deciding on which edges to delete. In the presented extraction model, a high weight $w(e_k)$ on an edge e_k between two class vertices c_i and c_j implies that the classes c_i and c_j belong to the same service candidate according to the utilized strategies. Consequently, the algorithm should primarily favor edges with low weights for deletion.

In the course of this work, the first attempt on decomposing the graph G involved a naive approach that defined a *cutoff threshold* τ and deleted all edges e_k where $w(e_k) < \tau$. The cutoff threshold t was chosen by sorting all edges in ascending order according to weight and selecting a certain percentile level of the sorted values as the cutoff τ. Despite tweaking and varying the percentile level, this approach produced less than satisfactory results. The resulting partitioning of the graph tended to form a low number of very large clusters, while often no other clusters remained besides these large ones. Independent of the candidate project that was tested and independent of the chosen percentile values, the results did not improve significantly. Upon further investigation of the approach, it became apparent that deleting all edges below a certain threshold has two major shortcomings. First, deleting all edges below the threshold leads to destruction of all microservice candidates that have a higher average internal coupling or weight than adjacent edges, but whose weights are still under the threshold. This causes the number of resulting service candidates to be too small. The second shortcoming is that the approach suffers from the formation of extraordinarily large clusters. This is mainly caused by situations where some of the connecting edges have weights lower than the intra-component average, but above the cutoff threshold τ. Intuitively these edges should be deleted so that the large cluster is partitioned into highly cohesive components. Because of the nature of the cutoff, this does not happen.

The revision of the partitioning algorithm resulted in a crucial insight on how to select the best edges for deletion. To increase the partitioning effect caused by the deletion of a single edge,

not the entire graph is considered during extraction, but only the minimum spanning tree (MST). By definition, every time an edge in a MST is deleted, it causes the MST to partition into two connected components. Similar techniques are used in social network analysis [GN02] to detect cohesive communities. Using the MST has the effect of increasing the tendency to create multiple components and create less large clusters. Another realization was the fact that it is very difficult to determine when to stop partitioning the graph. As a consequence, the algorithm must take the number of targeted partitions as an input parameter.
The above-mentioned observations were formalized and implemented in the MST-based graph clustering described below.

6.1 MST-Based Graph Clustering

Algorithm 1 presents a pseudocode notation of the algorithm implemented to divide the graph into microservices. The first step of the procedure is the weight inversion.

Algorithm 1 MST Clustering Algorithm

```
function CLUSTER(edges, n_part, s)
    for e ∈ edges do
        e.weight ← 1 / e.weight                          ▷ Invert edge weights
    end for
    edges_MST ← KRUSKAL(edges)                           ▷ Compute minimum spanning tree
    edges_MST ← SORT(edges_MST)        ▷ Sort the MST edges ascending according to weights
    edges_MST ← REVERSE(edges_MST)
    n ← 1
    while n <= n_part do
        edges_MST[0].delete()
        n ← COMPUTECOMPONENTS(edges_MST)
    end while
    components ← REDUCECLUSTERS(edges_MST, s)    ▷ Handle components that exceed size s
end function
```

Weight Inversion

In the previous section, it was noted that the extraction should be executed on the minimum spanning tree of the graph representing the monolith. But it is also desired that class nodes that have a high weight on the edges between them remain in the same microservice candidates or connected components. By directly computing the minimum spanning tree, this reasoning would be defeated. Hence, the weights $w(e_k)$ on the edges $e_k \in E$ in the graph $G = (V, E)$ have to be inverted:

$$w(e_k) = \frac{1}{w(e_k)}$$

With this conversion, the minimum spanning tree $MST(G)$ of the graph G will preserve the most important edges according to the computed couplings or weights.

MST Computation

The next step is the computation of the minimum spanning tree $MST(G)$. In this work, the KRUSKAL algorithm [Cor09] is used to compute the minimum spanning tree. Since the KRUSKAL algorithm is a well-known standard algorithm for minimum spanning tree problems, it will not be discussed in more detail here.

Edge Sorting

The goal is to partition $MST(G)$ by deleting edges with low coupling. Let $edges_{MST}$ be the set of edges that make up the minimum spanning tree $MST(G)$. The set $edges_{MST}$ is sorted according to their inverted weight, and then reversed so that edges with the highest inverted weight – and thus the lowest coupling – occur first in the list.

Iterative Edge Deletion

As noted above, the number of partitions to be achieved is assumed to be available as an input parameter to this algorithm. Let this number be denoted by n_{part}. The initial number of current connected components is always $n = 1$, since the procedure starts with an entire minimum spanning tree. The algorithm iteratively repeats the following steps until $n >= n_{part}$:

- Remove the edge with the highest inverted weight from the list of edges in the minimum spanning tree. Since the edges were sorted in the previous steps, this means removing the first edge in the list.

- Traverse the remaining edges and compute the number of connected components that the edges span up, the variable n is updated with this computed value. The *depth first search* (DFS) strategy is used to detect the number of connected components, which is illustrated in algorithms 2 and 3.

The iteration ends when n grows to be equal to or larger than n_{part}.

Algorithm 2 Identification of Connected Components

```
function COMPONENTS(nodes)
    components ← ∅
    for n ∈ nodes do
        if n.visited == false then
            component ← ∅
            n.visited = true
            component.add(n)
            DFS(n,component)
            components.add(component)
        end if
    end for
    return components
end function
```

Algorithm 3 Depth First Search

```
function DFS(node, component)
    for neighbor ∈ nodes.neighbors do
        if neighbor.visited == false then
            neighbor.visited = true
            component.add(neighbor)
            DFS(neighbor,component)
        end if
    end for
end function
```

Handling Exceptionally Large Components

To tackle the above-mentioned problem of having a few extraordinarily large components in the extractions, the resulting connected components after the edge deletion are checked against a predefined size threshold s. All connected components whose number of contained class nodes is larger than this threshold s are divided further by deleting the class node with the highest degree. This reduces the size of the clusters while keeping internal coupling inside the remaining connected components high. This procedure is outlined in algorithm 4. The splitting function that divides large clusters into smaller ones is presented in algorithm 5.

Algorithm 4 Reduce Large Clusters

```
function REDUCECLUSTERS(components, s)
    while SIZE(components) > 0 do
        components ← SORT(components)
        components ← REVERSE(components)
        largest = components[0]
        if SIZE(largest) > s then
            components[0].delete()
            splitted = SPLIT(largest)
            components.add(splitted)
        else
            return components
        end if
    end while
    return components
end function
```

Algorithm 5 Split Component

```
function SPLIT(component)
    nodes ← NODES(component)
    nodes ← SORT(component)
    nodes[0].delete()
    return COMPONENTS(nodes)
end function
```

6.2 Analysis

The complexity of the presented algorithm in algorithm Listing 1 is determined by the complexity terms of the subprocedures performed step-by-step. Let's denote the time complexity with respect to the number of edges $|E|$ and the number of vertices $|V|$ of the graph.
The first step where the edge weights are inverted has a complexity of $O(|E|)$ by iterating through the entire set of edges once. KRUSKAL's algorithm for finding the minimum spanning tree of a simple, connected and weighted graph has a worst case complexity of $O(|E|\ log(|V|))$ [GT08]. The sorting algorithm used to sort the edge list is not specified further, but for this analysis the theoretical lower bound for comparison-based sorting is taken into consideration, which is $O(|E|\ log(|E|))$. This complexity is achieved by very common sorting algorithms such as merge-sort and quick-sort [GT08]. The reversion step after sorting the edges has to iterate once through the entire list, thus requiring $O(|E|)$ time. The loop that iteratively deletes edges and computes the remaining connected components iterates n_{part} times, since n is always initiated with the

value of 1. As noted, the computation of the connected components is achieved by a depth-first search, which has a worst case complexity of $O(|E| + |V|)$ [GT08]. For the graphs at hand, the number of edges $|E|$ is usually comparable or higher than the number of vertices $|V|$. Therefore, the complexity can be simplified $O(|E|)$. Taking the loop repetitions into account, the entire while-loop results in a worst case execution time of $O(n_{part} \cdot |E|)$. As a last part of the complexity analysis, the cluster reduction function in algorithm 4 is investigated: The while loop executes at most $O(|E|)$ steps, with $O(|E| \cdot log(|E|))$ for the sorting routine and $O(|E|)$ for the reverse function. The splitting function defined in algorithm 5 is also called inside the loop. It has combined complexity of $O(|E| \cdot log(|E|))$ since the sorting inside the split function dominates the other operations asymptotically. For the entire reduce function, the combined complexity equals to $O(2 \cdot |E| \cdot log(|E|)) + O(|E|)$ and hence can be simplified to $O(|E| \cdot log(|E|))$ for the entire function.
The overall complexity is computed by adding the complexities of the several parts listed above:

$$\underbrace{O(|E|\ log(|V|))}_{\text{KRUSKAL}} + \underbrace{O(|E|\ log(|E|))}_{\text{sorting}} + \underbrace{O(|E|)}_{\text{reverse}} + \underbrace{O(n_{part} \cdot |E|)}_{\text{edge deletion loop}} + \underbrace{O(|E| \cdot log(|E|))}_{\text{cluster reduction}}$$

As noted above, the number of edges $|E|$ is usually comparable or higher than the number of vertices $|V|$. Therefore, the complexity terms for the KRUSKAL algorithm and for the sorting procedure can be ignored since they are asymptotically dominated or equal to the complexity of the cluster reduction. Also, the term, $O(|E|)$ for the reverse function can be ignored since it has a slower growth than the remaining terms. These steps give a resulting complexity of:

$$O(|E| \cdot log(|E|)) + O(n_{part} \cdot |E|) \tag{6.1}$$

By applying the sum rule for the big-o notation [Pre08] as follows:

$$O(f(n)) + O(g(n)) = O(max(f(n), g(n)))$$

the simplified complexity of the MST-based clustering algorithm is:

$$O(max(log(|E|), n_{part}) \cdot |E|) \tag{6.2}$$

Chapter 7

Implementation

The derived strategies, extraction model and clustering algorithm have been implemented in a research prototype. The settings in which the prototype is intended for use are refactoring scenarios where engineers and developers have the goal of extracting microservices from a monolithic code base as a part of a migration towards a complete microservice architecture. The prototype presented here is to be understood as a support tool which points the developer towards the most promising candidates for microservice extraction rather than an out-of-the box recommendation tool that comes up with complete and cohesive microservices that can be directly adapted and deployed.

7.1 Use Cases

Although the presented extraction model (cf. Section 4) and the corresponding strategies and clustering algorithms are designed to work on repositories of any class-based object oriented programming language, for the prototype, the space of possible input repository types was restricted for the sake of simplicity. The prototype supports repositories with projects written in *Java*, *Ruby* or *Python*. Those three technology stacks were chosen due to their wide usage in web applications and microservice-oriented applications throughout industry and the open-source community. In the following, a brief summary of the actual use cases the prototype supports is given.

U1: Clone
The prototype allows the user to clone a publicly accessible Git repository via its HTTPS uniform resource identifier (URI). The given repository is automatically copied by the prototype to the local storage of the machine. The cloning process always checks out the `master` branch of the corresponding Git repository.

U2: Extract
The application provides a graphical user interface that enables the user to select a previously cloned repository for microservice extraction. The user interface redirects the user to a view where the decomposition parameters (cf. data model description in 7.2) can be configured and the actual extraction process can be triggered. A successful extraction process returns the results and

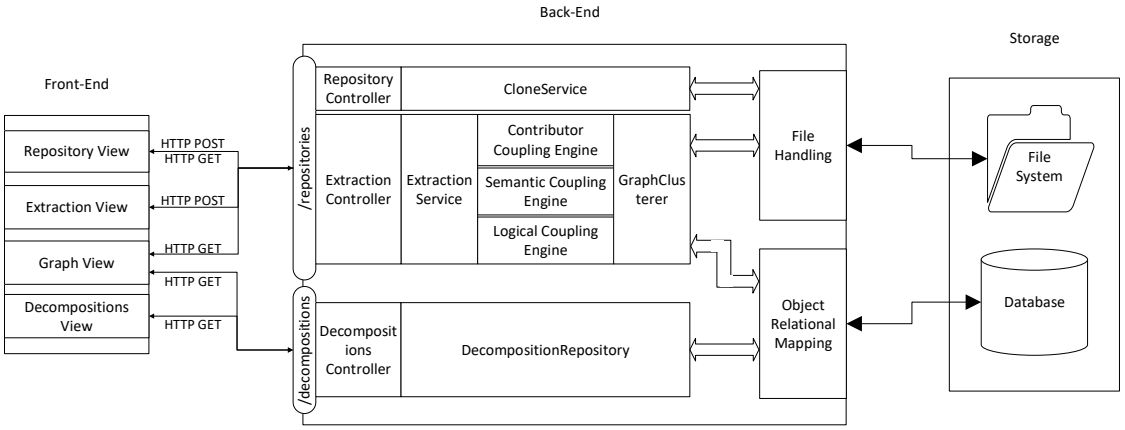


Figure 7.1: Conceptual architecture diagram of the prototype

redirects the user to a view where the extracted classes are visualized in a graph. Nodes of the graph represent the extracted classes, while edges represent the connections between the classes.

U3: View
The user has the option to view a list of all previously performed extractions. The list displays a unique identifier for each extraction and displays information about the specific decomposition parameters and repository used in that exact extraction. By selecting a specific extraction of the archived list, the graph representation of the results is again displayed to the user.

7.2 Architecture

The prototype should provide a simple user interface for the user to enter the decomposition parameters defined in Section 5 and select a certain code base or monolith that shall be analyzed for extraction. Due to its many advantages such as ubiquitous accessibility or lack of need for installation and updates from the user perspective, the prototype was designed as a web application.
Architecturally, the prototype takes the form of a traditional three-tiered web application consisting of *front-end*, *back-end* and a *data storage*. Since the nature of the problem domain and the approach is more computation-intensive than it is data-intensive, a simple relational database fulfills all the data-storage needs for the prototype. Hence, PostgreSQL [1] was chosen as a main data storage.

[1]PostgreSQL: `https://www.postgresql.org/`

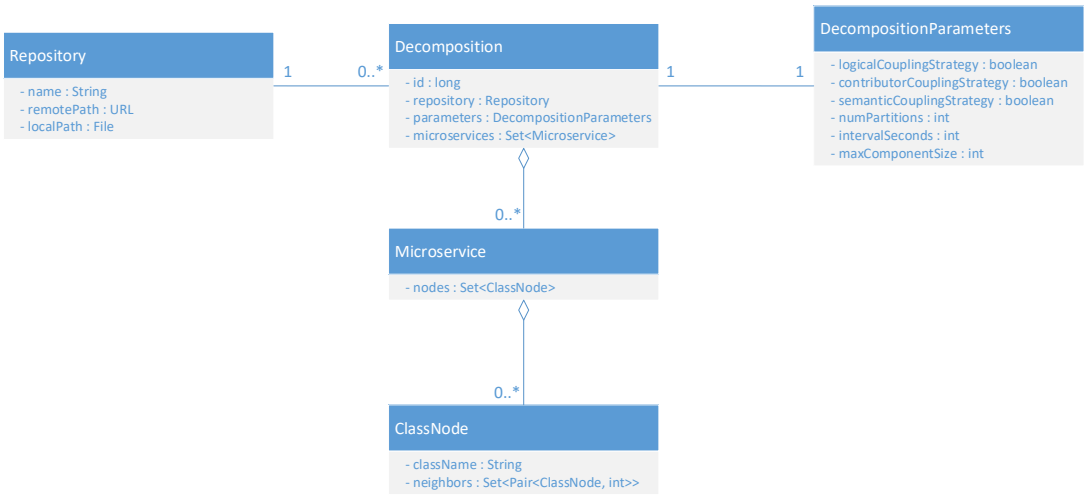


Figure 7.2: UML class diagram of the data model for the back-end resources

Communication between the front-end tier and the back-end is performed through HTTP requests. The front-end sends HTTP requests to the back-end which in turn responds with the requested resources. The HTTP request interface of the back-end is designed according to the RESTful principles [Fie00]. The back-end is responsible for the computation of the presented strategies and algorithms and also handles data management and queries towards the database. The front-end uses the resources provided by the back-end to display the results of the microservice extraction to the user in an intuitive manner.

7.3 Back-End

The back-end component is based on the well-known and widely used Java web application Framework *Spring* [2]. Using *Spring*'s notion of controllers, two RESTful resources are exposed by the back-end to outside consumers via HTTP: The *repository* resource and the *decomposition* resource.

Repository Resource

The repository resource represents a git repository, which is always the starting point for the extraction use case of the prototype. As explained in Section 4.1, the extraction model on which

[2] `https://spring.io/`

the prototype is built on operates on monolithic code bases from version control systems. The strategies are partially dependent on information from the version control systems such as file changes, authors and timestamps (cf. Section 5.1 and section 5.3). Since this constitutes a research prototype, the input source remains limited to one of the various version control systems. Due to its near ubiquitous use throughout industry and research, Git [3] was chosen as a source format for the code bases that are analyzed by the prototype. This allows to use some of the largest open source software communities like GitHub and Bitbucket as sources for test projects to be used in the evaluation of the prototype. By performing an HTTP POST request on the `/repositories` route of the back-end, an external client – such as the front-end component – can store new Git repository meta data in the database. At the same time, this request triggers the cloning process of the given repository from its remote location to the local file system of the machine where the prototype is deployed. The repository meta data – such as the remote path to the repository location and the repository name – are passed as a HTTP request body on the POST request and read by the back-end on the receiving side. For the Git-specific functionalities, JGit [4] was used as a client library to aid with git specific API commands such as cloning, computing diffs between revisions and retrieving files and contributors of a specific commit. Besides the possibility to create and clone new repositories via a HTTP POST request, the repository resource allows to request already existing repositories from the back-end either in bulk or by specifying the unique repository id generated at creating time as a URL path variable.

Extraction Parameters

The repository resource also provides the decomposition action – not to be confused with the *decomposition resource*. The decomposition action is a RESTful action that can be triggered for any existing repository under the `/repositories` endpoint by simply calling the specific repository path with the action name as a subroute: `/repositories/{id}/decomposition`. An HTTP POST request on that path triggers an analysis and microservice recommendation for the repository denoted by the `id` path variable. The configuration parameters for the recommendation to be performed are passed as a `DecompositionParameters` object in the body of the HTTP POST request. The `DecompositionParameters` object (cf. Figure 7.2) is then passed to the `DecompositionService` by the receiving controller. The `DecompositionService` determines which of the implemented strategies need to be combined based on the supplied `DecompositionParameters` object and calls the corresponding coupling engine to compute the couplings that represent the weights on the graph. There are three implemented coupling engines: The `LogicalCouplingEngine`, the `SemanticCouplingEngine` and the `ContributorCouplingEngine`. Implementation details about the engine internals are discussed further below. The couplings resulting from the coupling engines are then passed on to the graph clusterer component which handles the extraction from the graph representation (cf. *clustering* phase in Section 4).

[3] Git: `https://git-scm.com/`
[4] `https://eclipse.org/jgit/`

Decomposition Resource

Furthermore, the back-end REST API provides the `decomposition` resource which represents the result of a successful microservice extraction from a given repository. The `decomposition` resource is exposed on the `/decomposition` path through HTTP GET requests either by getting all existing decompositions in bulk or returning a specific `decomposition` instance if the consumer has specified a `id` variable on the request path. Also, the corresponding `decomposition` object is returned after the extraction procedure triggered by the decomposition action on the `repository` resource has finished its computation.

7.3.1 History Computation

The back-end contains a dedicated component that is responsible to generate an iterable list of changes between commits of the repository. The JGit library is used to retrieve the Git log of the corresponding repository. The returned log does not contain the actual changes of the commits, but only meta information. Hence, the log has to be traversed. At each iteration step of the traversal, the difference between the entire file tree at the current commit versus the file tree at the previous commit is computed. In the prototype, the content of the changes is ignored, only the change author, change type, timestamp and the involved file names are recovered. For each commit, these data are stored into a `ChangeEvent` object, which is in turn appended to a list. This resulting list of `ChangeEvent` instances is called the change history of the repository.
The history component needs to give special attention to the change types that it encounters. JGit provides information about whether a change is an *ADD*, *MODIFY*, *DELETE* or *RENAME* operation. Multiple problems arise in practice when running the strategies on the change history.
One type of problem is caused by files that were deleted in the history, but still manifest *ADD* or *MODIFY* changes in the history prior to being deleted. This leads to these files being processed in the strategy and clustering, and hence resulting in the final microservice recommendations despite not being present in the latest version of the monolith. Therefore, the history component detects those cases and excludes them from any further processing in the strategies that follow.
The second problem arises from renamed files. If a file named `A.java` is modified or added in the history, and in a later commit is then renamed to `B.java` and there are further modifications of that file, the resulting graph representation will interpret both `A.java` and `B.java` as separate files and nodes in the graph, despite the fact that `A.java` does not exist anymore. This is handled by carrying and updating a lookup table that keeps track of all renames encountered during the history traversal. At the end, the last valid name of each file is retrospectively applied to all changes involving that file. This way, no coupling information gets lost, but still the mentioned issue is prevented. As a sidenote, the Git version control system only interprets changes as *RENAME* if the name is changed and not more than 50 % of the files contents are changed. If the more than 50 % of the file content changes, Git considers this as a *DELETE* of the original file and an *ADD* of a new file. This case is not covered by the history component in the prototype, since in the semantics of the strategies, it makes sense to view those files as new files instead of renames.

7.3.2 Logical Coupling Engine

The `LogicalCouplingEngine` performs the computation of the logical couplings between class files in the given repository. Its computation is dependent on the `historyInterval` parameter from the `DecompositionParameters` object. The computation iterates over the entire git change history of the given repository and divides it into intervals of the size given by `historyInterval`. For each of the intervals, all the class files that have been mutated during that interval are collected into a set. Each of those sets of files are continuously appended to a list data structure denoted by L.
For each of the sets of files in the list L produced above, the logical coupling engine computes all possible pairwise permutations of files from that set. Formally, if S_i denotes the set of files modified at the i-th interval, this means computing the power set $P(S_i)$. The power set computation algorithm has a very high worst case complexity of $O(2^N)$ where N denotes the number of elements or files in the set. During development and testing of the prototype, it became apparent that the computation of the power set for sets with a size of $N > 12$ takes unacceptable amounts of time for the computation. To guarantee tolerable execution times for the user, the power set computation is therefore limited to sets with a maximum of 12 files.
The procedure maintains a hash table h for continuously storing and updating discovered logical coupling values. The two file names of each pair act as a compound unique key for the pair from the power set. Each time that pair is encountered in any power set during the iteration of the list L, the compound key of the pair is looked up in the hash table h. If this pair was already encountered, the value returned by the hash table on the given compound key is incremented by one and stored in the hash table h. If the hash table has no record of the compound key for the current file pair, a new hash table entry is created for the key and the value it points to is initialized to 1.
After a complete iteration of the list L, the hash table h will contain all pairs of files that have changed together and the number of times the conjoined change has occurred. The last step of the procedure exports a list of edges, where each edge has the two class files as adjacent nodes and the logical coupling from the hash table as a weight. The graph denoted by those edges is then processed further by the clustering algorithm from Section 6.

Complexity

As mentioned above, the logical coupling strategy starts by dividing the history into intervals and collecting the change events for each of the time intervals. For the following analysis, the operation that collects all change events that lie in a certain time interval is assumed to have a complexity of $O(1)$. Let H_M denote the history of the monolith M, then $L(H_M)$ gives the length of the history in seconds. Furthermore, the `DecompositionParameters` object comes with a parameter named `intervalSeconds` that determines the length of the intervals that the history H_M is divided into. The number of intervals is then defined as:

$$N = \frac{L(H_M)}{intervalSeconds}$$

All the following steps are consequently executed N times. The repeated steps involve the computation of the power set and the storage of the coupling pairs in the hash table. Let S be the set of class files in an arbitrary interval. As described above, the recursive power set algorithm requires $O(2^{|S|})$ time. If c denotes the maximum number of couplings, the hash table used to keep track of already discovered class file couplings has a worst case complexity of $O(c)$ for insert, read and update operations. An upper bound for the maximum number of couplings c can be estimated easily. Let C_M be the set of all class files of the monolith M If all the class files of the repository are fully connected to each other – *i.e.*, the representing graph is a full mesh or complete graph – the number of edges, and hence the number of couplings to be kept in the hash table is bounded by:

$$c \leq \frac{|C_M| \cdot (|C_M| - 1)}{2}$$

where $|C_M|$ denotes the number of classes in the monolith. Asymptotically the upper bound for c can be reduced to $O(|C_M|^2)$.
Hence, at each interval, the computational complexity is $O(2^{|S|} \cdot |C_M|^2)$. For N intervals, the total complexity of the logical coupling computation procedure results in

$$O(N \cdot 2^{|S|} \cdot |C_M|^2) \tag{7.1}$$

7.3.3 Semantic Coupling Engine

The `SemanticCouplingEngine` receives the git repository meta data as an input. The meta data point to the location of the cloned repository on the local file system of the machine. The semantic coupling procedure starts at the root of the repository file tree and traverses each class file it encounters. It uses the *visitor* design pattern [Gam95] to perform the visiting operation on each file it encounters. The visiting operation is responsible for reading, filtering and preprocessing the contents of all the files.
The visiting procedure filters out files that are written in a different programming language than Java, Ruby or Python, since the prototype and its use cases are optimized for those languages. After reading the raw class file content, the content is tokenized into a list of keywords. The tokenization splits the content into keywords by division characters. Examples for such characters are the *space* character, hyphens, underscores and all forms of brackets and parentheses. Also, compound identifiers in camel case format are split into the atomic words they are made up of.
The tokenized list is then passed through a blacklist filter. This is necessary because the semantic coupling strategy is very sensitive to the contents of a file. A great amount of the content in class files is very generic and not related to any domain or business entity or problem. Tokens like the reserved keywords for the different programming languages, but also commonly used keywords in web frameworks such as `controller`, `model` or `service` will occur in a large part of the class files without contributing any semantic information about the problem domain. Therefore, all instances of these keywords are removed from the list of tokens before any further processing

steps are conducted. A similar problem arises – especially in open source projects – with license headers in class files. A self-evident option would be to ignore comments, since license headers are always on commented-out lines. But comments can contain valuable domain information, therefore the comments are kept, but the typical keywords that occur in the most used licenses are filtered out analogous to reserved keywords of programming languages.
The tokenized and filtered list of words is stored for each file that is encountered during the tree traversal. The procedure continues to the computation of the cosine similarities. For this, all pairwise distinct class files are matched in a nested for loop, and their respective word lists – denoted W_i and W_j in the definition Section 5.2 – are passed to the tf-idf computing component which returns a similarity value of type `double`, with 0 meaning no similarity and 1 meaning complete congruence between the two files.
Internally, the tf-idf component performs the exact steps described in Section 5.2: Term extraction to form the term list of both files, vector computation for each of the files, and the cosine between the vectors which returns the similarity. The computed similarity value equals the weight between the two class nodes on the edge list that is exported at the end of the strategy computation.

Complexity

The traversal of the file tree of a monolith M with a set of classes C executes $O(|C|)$ read and filter steps. Each of those steps performs the read and filter operations for each line of code in the class file that is being visited. Hence, the read/filter sub-operation has a complexity of $O(LOC_{avg})$ where LOC_{avg} denotes the average number of lines of code per class file in the monolith M. By combining the $O(|C|)$ for the traversal and $O(LOC_{avg})$ for each sub-operation during the traversal, the total complexity of the traversal procedure can be expressed by $O(LOC_{tot})$, where LOC_{tot} denotes the total source lines of code for the entire monolith M.
For each of the class files read in the previous steps, the procedure computes the tf-idf vectors. Since this operation's complexity is directly dependent on the size of the token content of each class, it is not considered any further in the worst case complexity analysis. This is because the token content size and hence the vector computation is mainly determined by the LOC_{avg}, which is already factored into the complexity by now. The semantic coupling procedure then proceeds to permute all pairs of distinct class files to compute their tf-idf similarities. For this permutation, $O(|C|^2)$ steps have to be performed.
In total, this leaves us with a worst case complexity for the semantic coupling strategy of:

$$O(LOC_{tot}) + O(|C|^2) \tag{7.2}$$

7.3.4 Contributor Coupling Engine

The contributor coupling procedure performed by the `ContributorCouplingEngine` expects a list of change events as input. Each of the change events in the list keeps track of the files that were modified at that certain change event and contains the unique identification of the author who committed the change. In the concrete case of the prototype, the e-mail address from the

version control system is used as a means of uniquely identifying authors.
In a first step, the input list of change events H_M is traversed in order to create a hash table h that links the names of the files (hash keys) with the sets of authors that have contributed to them (hash values). To construct this hash table, the procedure iterates through the list H_M change event by change event. For each change event, all the files that were modified during that change event are processed at once. Each of the files is put into the hash table h with its file name as a hash key and the author that contributed the change event is added to the set of authors associated with that key. After having processed all change events in this manner, the hash table h will finally contain all modified class files with their associated contributor sets.
The computed contributor hash table h is then passed to the next stage, where the contributor couplings between the class files are computed. This is achieved by combining all distinct pairs of class files present in the hash table h and counting the number of contributors they share with each other. In other words, for each pair, the cardinality of the intersection of their respective contributor sets is stored as a coupling value between those files. The procedure is concluded by exporting the couplings between the files as a list of edges between class nodes with the coupling value as weight.

Complexity

As described above, the construction of the author lookup hash table h involves iterating through the history H_M of the monolith. The construction hence gives a complexity of $O(|H_M|)$. To compute the number of shared contributors between all pairwise distinct class files, the procedure has to perform a nested loop which examines all possible pairs of class files. Knowing that there are $\frac{|C|\cdot(|C|-1)}{2}$ possible pairs, one can deduce that the procedure needs to perform $O(|C|^2)$ processing steps for computing the contributor coupling value for all pairs.
Therefore the total complexity of the contributor coupling strategy is defined as:

$$O(|H_M|)) + O(|C|^2) \tag{7.3}$$

7.4 Front-End

The front-end of the prototype is based on the Angular2 [5] web application framework. The used framework allows clean separation of view code in HTML5 from the behaviour and front-end logic, which can be expressed in TypeScript. TypeScript brings advantages of object-oriented languages such as classes, interfaces, inheritance and static typing to JavaScript environments and thus helps in creating more readable and robust front-end code.
The front-end implements four views, as depicted in architecture diagram 7.1. The repository view implements the starting point for the clone use case **U1**. It allows the user to clone Git repositories by entering the remote HTTPS URI. Also, previously cloned and stored Git repositories are

[5] `https://angular.io/`

listed in the view. By choosing one of the listed repositories, the user can initiate the extraction use case **U2** and is redirected to the extraction view.

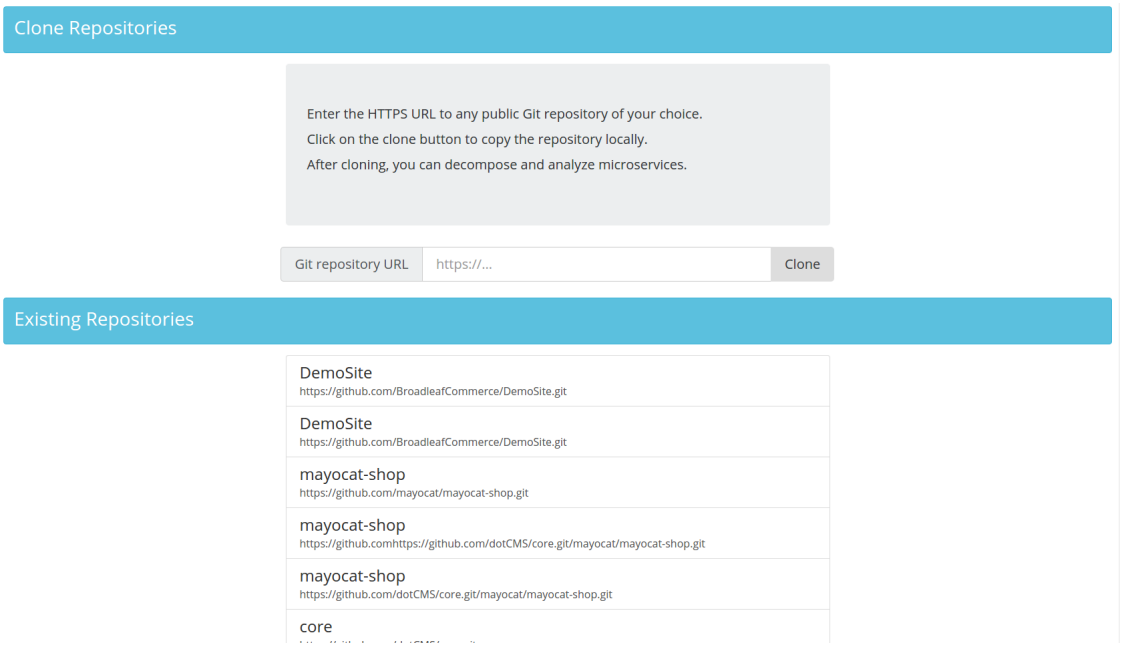


Figure 7.3: Repository view in the front-end component that implements use case **U1**.

The extraction view implements **U2**. It provides controls for configuring decomposition parameters such as the types and combination of coupling strategies to be used or the tweaking of parameter values such as the history interval. Upon clicking the decompose button to trigger the extraction of microservice candidates, the front-end component creates a `DecompositionParameters` instance with the configured values. The object is sent in the body of a HTTP POST request to the `/repositories/{id}/decompose` endpoint on the REST API of the back-end. The back-end computes the microservice extractions and returns them as a response to the HTTP POST request. The response carries a representation of the microservices in the body. The `GraphRepresentation` model was defined so the internal model of microservices used by the back-end can be hidden from external consumers and only view-specific information and structure of the resulting microservices are returned to the front-end.

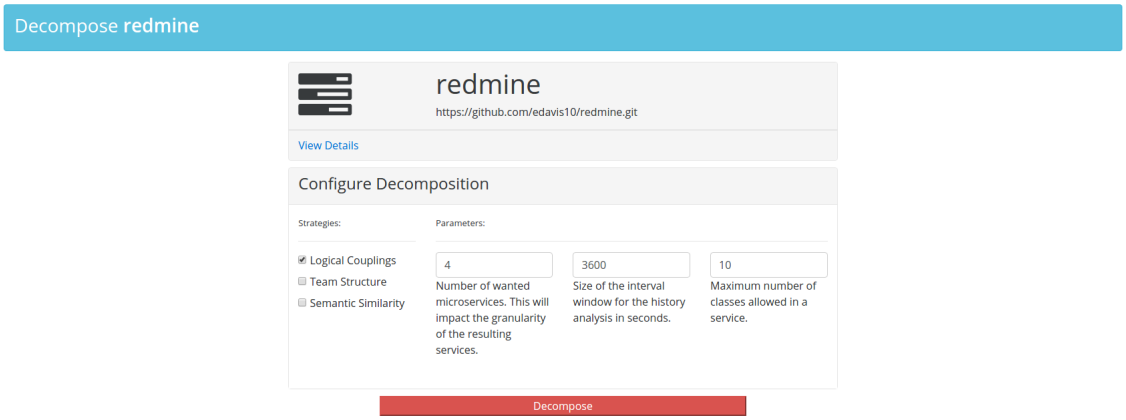


Figure 7.4: Screenshot of the extraction view which allows the user to configure the decomposition parameters and trigger the analysis.

As depicted in Figure 7.5, a `GraphRepresentation` instance represents one single microservice and aggregates all the nodes and edges between the nodes that belong to that microservice. The nodes are of the type `NodeRepresentation` and have a unique id in form of a long integer, a label to be displayed, which will be filled with the class name of the node, and the nodes store a color value for their graphical visualization on the front-end. The colors are randomly generated based on the structure of the services, *i.e.*, all the nodes in a microservice have the same color, while all microservices have distinct colors from each other. The edges are of the type `EdgeRepresentation`, and basically just link two `NodeRepresentation` instances together by their unique ids.

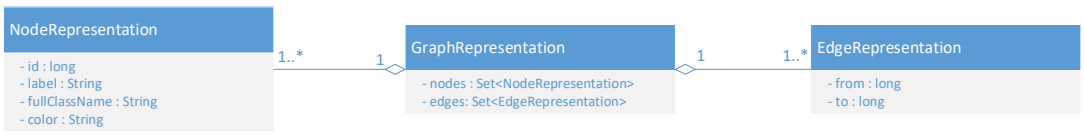


Figure 7.5: UML Class Diagram of the Representation Types between Front-End and Back-End

A successful extraction process returns an array of `GraphRepresentation` objects to the front-end. The graph view is then responsible for displaying the extracted microservices in a graph, as described in use case **U3**. It allows the user to zoom and pinch the graph and to open or close microservices in order to see their details. For the rendering of the nodes and edges and the layout of the graph, the VisJS [6] library was used. It helps in presenting a clean view of the graph by using techniques such as gravitational force layout.

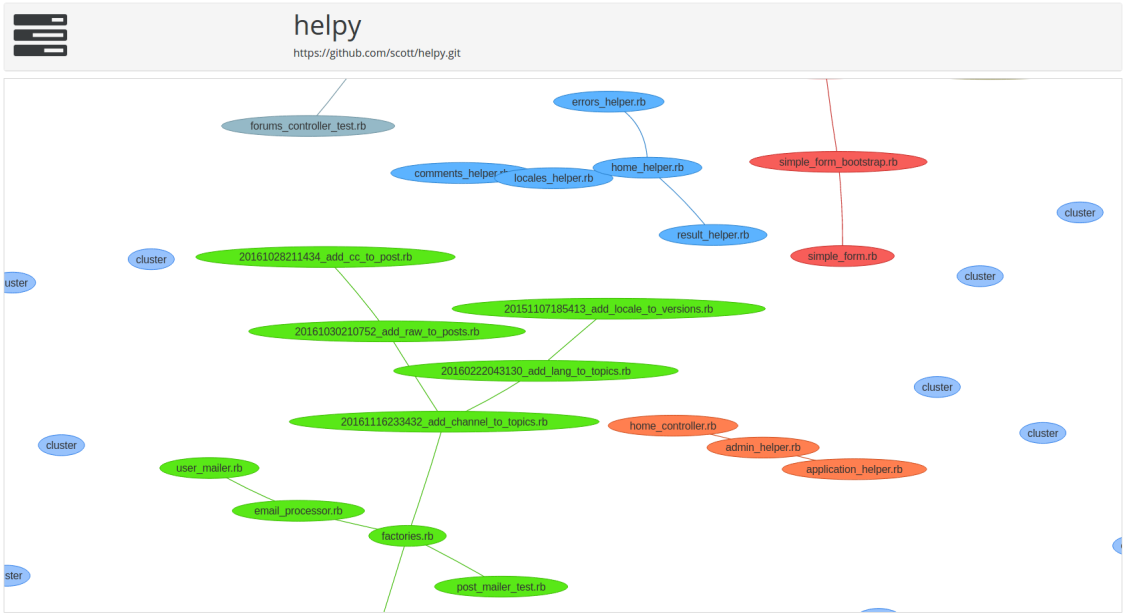


Figure 7.6: Screenshot of the graph view which enables the user to browse through computed microservice recommendations.

Already performed extractions can always be looked up and reviewed again. This is enabled by the decompositions view, which lists all successful previous extraction attempts together with their repository and decomposition parameters. Upon clicking one of the decompositions in the list, the graph view displays the microservices extracted during that extraction.

[6] `http://visjs.org/`

Chapter 8

Evaluation

The introduction chapter stated two research questions for the implemented prototype and its evaluation:

> **RQ2.1: What is the performance of the implemented prototype with respect to execution time?**
>
> **RQ2.2: What is the quality of the microservice recommendations generated by the prototype?**

Therefore, the evaluation for the implemented prototype and the presented approach consists of two aspects, the performance aspect evaluation that aims to answer **RQ2.1** and the recommendation quality evaluation, which is the topic of interest in **RQ2.2**.
While the performance of the prototype can be evaluated in a straight-forward manner by recording execution times with different input settings, the evaluation of the recommendation quality is much more involved. As noted in the introductory section of this thesis, the microservice architecture has no formal definition or notion that can be used as a ground truth for studies. The standard literature on microservices [New15, Fow14] struggles to give a quantifiable or at least formal characterization of the term *microservice*. Similar works and studies conduct qualitative evaluation by manual inspection of the results as in [LTV16]. Surveys and inspections by experts on the subject as a tool of qualitative evaluation can potentially deliver more meaningful insights. In the specific context of microservices a qualitative inspection by experts comes with several drawbacks and difficulties. The quality of the resulting microservice candidates is not a one-dimensional aspect. The rating of the microservice recommendation depends highly on the context and the motivation behind the refactoring effort that has lead to it. A microservice extraction might have scalability as its main motivation, resulting in different quality requirements by the expert than for a microservice extraction done to improve team structure and code maintenance. It is therefore very difficult to come up with an expressive and precise set of dimensions on which a survey could be conducted.
In related research fields, a favoured alternative approach to evaluate contributions such as the one in this thesis has been the use of quantitative metrics as a proxy for the quality of the produced results. While there are clearly established and well-known quality metrics in fields such as object-oriented design [CK94], the field of services computing – and especially microservices

– exhibits a very scarce amount of such efforts. Propositions of metrics for service environments such as in [PRFT07] are often based on traditional software metrics and slightly extended in some aspects to accommodate the different requirements in a service environment. Other proposed metrics suites for service environments [KAR+11, RKS+11] involve information about business properties and business entities in their computation and hence are not easily obtainable directly from the code base. The authors of [KAR+11] present a method to compute the conceptual coupling between services using information retrieval techniques. The *average domain redundancy* metric introduced in the following sections is built on a similar rationale, but takes a different measurement approach that is more appropriate for the formal service model of this thesis. The other metrics used in the quality evaluation are designed for expressiveness with respect to aspects that were deemed important in the very specific context of microservices. These aspects include microservice sizing, team structure and team communication overhead.

All the results in the performance and quality evaluation sections in this chapter were obtained through experiments conducted with the following algorithm parameters:

- numPartitions: 4
- maxComponentSize: 10
- historyInterval: 3600
- logical coupling weight factor c_{LC}: 1
- contributor coupling weight factor c_{CC}: 1
- semantic coupling weight factor c_{SC}: 1

During testing and development of the prototype, the combination of the $numPartitions$ and $maxComponentSize$ listed above showed to deliver the most promising results when it comes to the sizing of the microservice recommendations. The $historyInterval$ was set to the size of an hour (3600 seconds) in order to balance the tread-off between a stringent interpretation of a logical coupling – implied by a shorter $historyInterval$ of a few hundred seconds as in [ZW04] – and a larger interval size that generates stronger couplings while still preserving the co-change semantics. The weights for the strategy factors are set to 1 for this initial evaluation since there is no empirical evidence on the importance of the factors.

8.1 Sample Selection

As described in Section 7.2, the implemented prototype utilizes Git repositories as an input for monoliths. Open-source projects and communities are a main source of input for evaluations such as in this thesis. Benefits include open accessibility not only to code but also to meta-information such as contributor info and change history. Also, open-source projects include a great variety of project sizes and team sizes. These characteristics make open-source projects a prime candidate

for the evaluation in this thesis. The largest and most prominent source for open-source projects is GitHub [1] [LRM14].
To select viable candidate repositories of monoliths, several criteria were formulated. Technically, the approach devised in this thesis and the prototype are able to handle any repository of any class-based, object-oriented programming language. Nevertheless, the application context and the research questions demand a more specifically confined set of criteria for test repositories.

8.1.1 Criteria

Application Type
The main drivers behind microservices have been the new demands posed by the inherently distributed nature of the world wide web and novel developments on the infrastructure side such as cloud computing. Consequently it makes sense to limit the evaluation sample projects to web applications.

Technology Stack
Since the entire extraction model presented in Chapter 4 is based on the premise that the monolith under investigation is composed of classes, it is obvious that the set of sample projects will be limited to projects written in a technology stack that adheres to this class-based nature.
A hard challenge in the decomposition of monoliths to microservices – or distributed systems in general – is state. Having state – *e.g.*, in the form of a database – in your application tremendously increases the difficulty of scaling, distributing and decomposing said application. The approach presented here overcomes this issue by assuming that all the candidate projects employ an object-relational-mapping (ORM) system to handle domain model and data. In monoliths where an ORM is used to handle data, the domain models and database tables are represented by plain classes in the corresponding programming language used for the monolith. Thus, in the extraction model, the data entities defined by the ORM classes are treated like any other class in the monolith and hence result in a corresponding graph node that is coupled to other classes and potentially ends up in one of the recommended microservice candidates.
In combination with the above mentioned requirement that the applications should be web applications, the class-based ORM criterion further limits the space of potential candidate repositories for evaluation. Applications that typically fulfill these requirements are based on traditional web application technology stacks such as Spring [2], Java EE [3] in Java environments or Rails [4] and Django [5] for Ruby and Python environments. Furthermore, empirical studies on practices in (micro-) service computing revealed that Java is still the most prominent language to implement microservices, followed by scripting languages such as Ruby or Python [SCL15]. It is therefore clear that the projects considered for the evaluation are all web application projects written in

[1] https://github.com/
[2] https://spring.io/
[3] http://www.oracle.com/technetwork/java/javaee/overview/index.html
[4] http://rubyonrails.org/
[5] https://www.djangoproject.com/

Java, Python or Ruby and employ an ORM solution such as the one provided by Hibernate [6] or the database schema migration solutions in Rails and Django.

Repository Size

The size of a repository can be expressed along multiple dimensions. With regard to the performance evaluation, the important size dimensions are the ones that impact the performance of the strategies. The logical coupling strategy mainly depends on the size of the history of the corresponding repository. The history size can be expressed either by the number of commits in the history or by the length of the time window between the first and last contribution to the repository. For the contributor coupling, the total number of contributors is the size metric that we're interested in. But this in turn also depends on the history size, since the history has to be traversed to detect contributions by contributors. For the semantic coupling strategy, the source code class files have to be read from disk and processed. Therefore, the size of the code in SLOC (source lines of code) will be the main influencing factor of performance.
The history size of the projects should include an equal distribution of projects with large histories (*e.g.*, a large number of commits) and small histories. Young productive open source web applications can be found with a commit count of a few hundreds, while larger and more active projects can exhibit tens of thousands of commits. In this evaluation, the candidate repositories must have at least 200 and at most 25000 commits in their history to be considered in the sample.
The team size of any of the repositories shall represent typical web application team scale found in the industry. This means very large open-source projects with thousands of contributors will not be taken into account for the evaluation. Candidate repositories must have at least 5 and at most 200 contributors. This range of contributors mirrors the typical sizes of development teams from small development agencies up to large internet leaders like Amazon or Facebook.
The criteria regarding the size of repositories in source lines of code (SLOC) are devised similar to the history size. The evaluation includes repositories with varying SLOC counts to account for the diverse monolith sizes that are present in practice and industry. A lower bound for the code size is at 1000 SLOC while projects larger than 500'000 are also not taken into account.

Summary

In summary, the definitive criteria that must be fulfilled by a repository to be included in te evaluation boils down to the following: A repository must

- be open source
- use Git as a version control system
- contain a web application
- use a class-based ORM technology
- be written in Java, Ruby or Python
- have at least 200 commits in its history

[6] `http://hibernate.org/`

- have at most 25000 commits in its history
- have at least 5 contributors
- have at most 200 contributors
- contain at least 1000 SLOC
- contain at at most 500000 SLOC

8.1.2 Sample Projects

Table 8.1 summarizes all the monolithic repositories used in the evaluation that follows. The SLOC of each project was measured using the unix tool *sloccount* [7], that has been previously [Whe01] used to determine the size of very large open source projects such as the Linux kernel. The commit and contributor counts for the projects are taken from the corresponding Git version control information of the *master* branch of each repository.

8.2 Performance

In research question **RQ2**, the goal is to implement a prototype implementation of the devised extraction model, strategies and clustering algorithm. The prototype is targeted at developers in refactoring scenarios when moving from a monolithic software architecture to a microservice oriented architecture. Besides the actual nature and quality of the microservice recommendations, performance is a critical aspect when it comes to refactoring support tools for developers. The prototype must show an acceptable scaling behaviour when confronted with large input data sets.
In this part of the evaluation, the performance of the implemented algorithms shall be investigated depending on different input projects of different sizes. The execution time of each of the processing stages is used as a proxy for the performance of the prototype. Not every strategy depends on the same input characteristics when it comes to execution time. Consequently, the performance recordings of the different strategies are each contrasted with a different input variable, such as history size, code size or team size of the input project – depending on the strategy that is being evaluated.

[7] `http://www.dwheeler.com/sloccount/`

Table 8.1: Monolith Projects used in Evaluation

Project Name	Technology	Commits	Contributors	SLOC
BroadLeaf DemoSite	Java	1477	22	1301
Mayocat Shop	Java	1670	4	33216
OpenCMS	Java	21087	35	512294
TNTConcept	Java	216	15	118356
PetClinic	Java	545	32	1564
Sunrise Shop	Java	1859	11	18820
Helpy	Ruby	1650	42	9230
Spina	Ruby	489	38	2468
Sharetribe	Ruby	14940	46	53845
Hours	Ruby	796	21	4268
Rstat.Us	Ruby	2260	64	6807
Kandan	Ruby	868	52	1553
Fulcrum	Ruby	697	44	3085
Redmine	Ruby	12990	6	87529
Chiliproject	Ruby	5532	39	65171
DjangoCMS	Python	15451	355	41631
Django Fiber	Python	848	20	4686
Mezzanine	Python	4964	238	11780
Wagtail	Python	6808	205	48971
Mayan	Python	6000	25	39225
Django-Shop	Python	3451	62	8281
Django Oscar	Python	6881	170	32272
Taiga Black	Python	3226	38	58295
Django-Wiki	Python	1589	64	8113

8.2.1 Logical Coupling Strategy

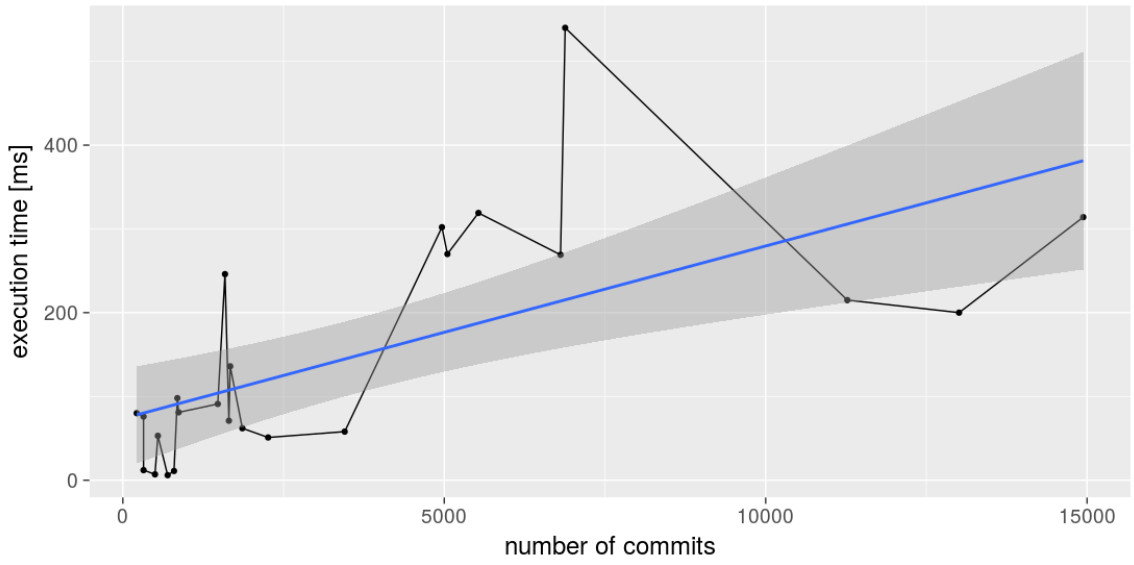


Figure 8.1: Plot of the execution time as a function of the commit count for the logical coupling strategy

As described in Section 5.1 and Section 7.3.2, the performance of the logical coupling strategy is mainly determined by the size of the history of the test project to be analyzed. The performance is evaluated against the *commit count* of each evaluation project and against the *history length* in days. The history length corresponds to the number of days that have passed since the first contribution to the project in the version control and the last contribution that was made.
The performance measurement is done by recording a first UNIX epoch timestamp t_{start} at the point in the program immediately before the logical coupling strategy implementation starts processing the ordered change history H_M of the monolith M. The second timestamp t_{end} is recorded immediately after the logical coupling routine has finished computing the edge list of the class nodes and the weights on the edges. The logical coupling execution time t_{LC} is then given by $t_{LC} = t_{end} - t_{start}$. Figure 8.1 shows a plot of the execution time in milliseconds against the number of commits of the corresponding repository. The bullet points correspond to the measured values while the black solid line is just used as a connector for better visibility. The blue trend line indicates clearly that the execution time rises with the number of commits. Nevertheless, there are some large variations of the execution times with spikes towards higher execution time values. Examples are the execution time measurements around 5000 commits and arount 7500 commits. Points like these lie outside the grey-marked 95 % point-wise confidence interval.

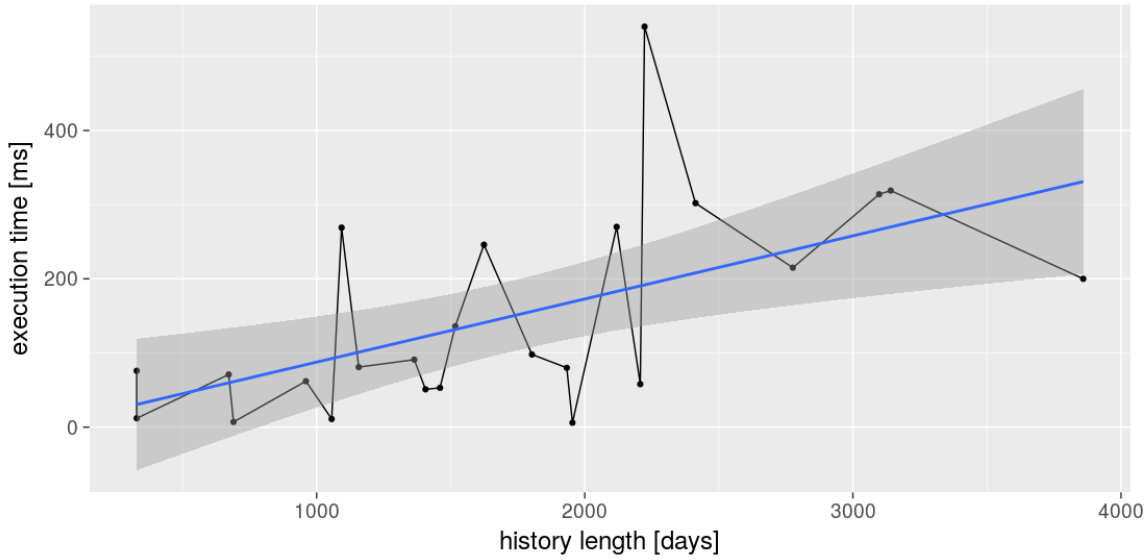


Figure 8.2: Plot of the execution time as a function of the history length for the logical coupling strategy

A similar constellation can be observed in Figure 8.2. Again, the trend line shows a flat increase of the execution times when the history length rises, but there are some anomalies that lie outside the grey interval.
These observations mostly confirm the theoretical understanding of the complexity and performance behaviour that was derived in the asymptotic analysis of the logical coupling strategy implementation in Section 7.3.2. The theoretical complexity of $O(N \cdot 2^{|S|} \cdot |C_M|^2)$ indicates that the execution time will only partially depend on the history size, denoted by N. The spikes in the plots above are attributed to sample repositories that exhibit certain change events in their history where the number of files that were modified is significantly higher than the usual number of changed files. A few of those outliers per repository are enough to cause a significantly higher execution time for the computation of the logical couplings because of the exponential behaviour of the complexity depending on the number of changed files, as indicated by the $2^{|S|}$ in the total complexity term, where $|S|$ denotes the size of the sets which are involved in the power set computation at each step (cf. Section 7.3.2).

8.2.2 Contributor Coupling Strategy

As described in Section 7.3.4, to compute the common contributors between class files, the contributor coupling strategy routine has to traverse the history H_M of the monolith M and maintain a hash table data structure that collects the new contributors that made a change to a file at a certain change event $h_i \in H_M$. Therefore, the history size is the major influencing factor when it comes to execution time. The execution time of the contributor coupling strategy t_{CC} is measured analogously to the logical coupling execution time.

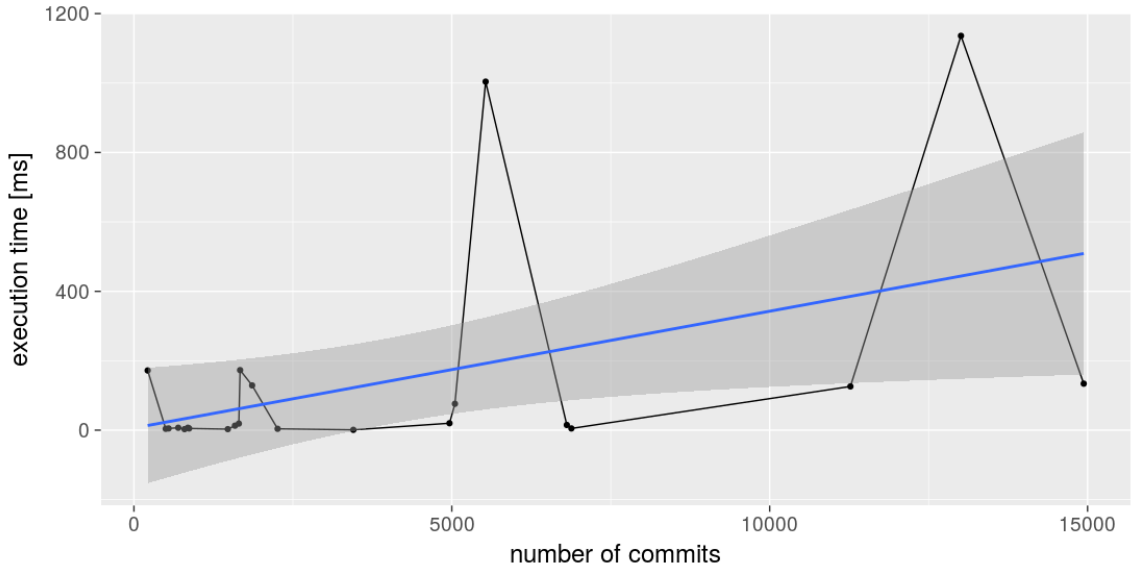


Figure 8.3: Execution time as a function of the history length for the contributor coupling strategy

A plot of the execution time for the contributor coupling strategy depending on the size of history in number of commits is shown in Figure 8.3. The trend line in blue and the 95 % confidence interval marked by the grey area illustrate that the execution time scales in a flat linear manner with the number of commits in the history of the repository that is being decomposed. There are two clear outliers at around 6000 commits and 14000 commits with significantly higher execution times.
By consulting the complexity derived for the implementation in Section 7.3.4, the observations can be rationalized. The total complexity amounts to $O(|H_M|)) + O(|C|^2)$ where $|H_M|$ is the size of the history in number of commits and $|C|$ denotes the number of class files in the repository that is being decomposed. While the impact of the growing history size H_M can be clearly observed in the plot 8.3, the extreme outliers are attributed to sample projects where the number of class files $|C|$ is unusually high and therefore dominates the effect of the history size and leads to higher execution times.

8.2.3 Semantic Coupling Strategy

Recalling the definition (cf. Section 5.2) and implementation (cf. Section 7.3.3), it becomes apparent that the computation of the semantic coupling values is mainly dependent on the amount of tokens contained in the input data set. In other words, the amount of code or identifiers is a direct influence on the execution time of the semantic coupling strategy.

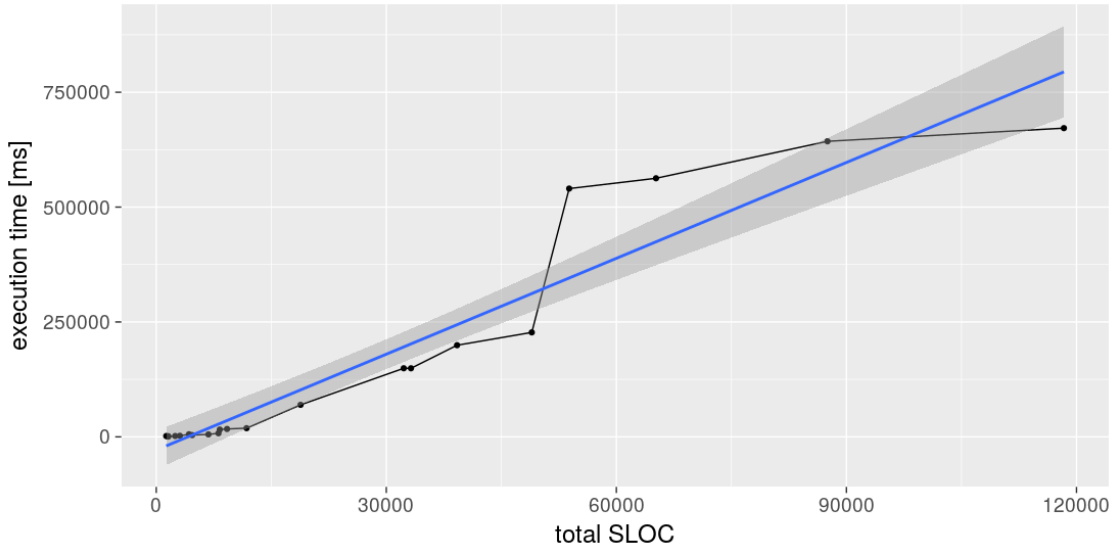


Figure 8.4: Execution time as a function of the code size for the semantic coupling strategy

To illustrate the scalability of the computation, Figure 8.4 shows the semantic coupling execution time of the projects presented in table 8.1 as a function of the size of the projects in SLOC (source lines of code). The semantic coupling execution time is recorded as the difference between the start timestamp t_{start} immediately before the routine starts traversing the file tree of the repository M and the end timestamp t_{end} immediately after the semantic coupling weights on the edges have been returned.
The performance indicated by the execution times measured on the sample projects confirm the intuition that the performance of the semantic coupling strategy is mainly and directly impacted by the total size of the code in the repository in source lines of code (SLOC). The growth takes rather linear form, which means the SLOC factor dominates the impact of the number of classes in the monlith, which – according to the asymptotic analysis presented in Section 7.3.3 – also influences the complexity of the semantic coupling procedure in a quadratic manner.

8.3 Quality Metrics

Apart from the execution performance of the implemented prototype, the actual recommendations for microservice candidates to be extracted are the key aspect that has to be evaluated in order to assess the value of such a tool in a microservice refactoring scenario. An optimal assessment of the quality of the recommended extractions involves performing the recommended extractions and then deploying the redesigned application. One could then monitor key metrics such as response time, failure rate and effects on development team composition and communication overhead between development teams. Such an assessment is clearly not feasible to perform at any non-anecdotal scale. Therefore, automatically measurable (static) metrics are devised and

used to indicate quality aspects of the computed extraction recommendation.

8.3.1 Size Aspect

Microservice granularity and size is one of the most intensely debated topics in the microservice community, as for instance in [Til14, Mor15, Ła15, Bel16]. While the industry has proposed multiple rules of thumb to determine the optimal size for a microservice, there seems to exist no established consensus on microservice sizing and granularity. The microservice sizes proposed by those informal rules varies from low microservices with only tens of SLOC, while others recommend microservices with up to thousands of SLOC.
A look at previous scientific work in the field reveals that there have not been many studies that give attention to the subject of microservice size and granularity. One of the first papers to investigate microservice practices in the industry empirically is [SCL15]. In their work, the authors performed a survey and among other things also investigated the typical size of microservices employed in practice [SCL15]. Their results indicate that microservice sizes under 100 SLOC – as proposed by some industry proponents [Cre14] – are very rare and are only reported by around 3 % of the respondents. The occurrence of very large microservice with more than 10000 SLOC is equally low according to [SCL15], leaving the vast majority of microservices in the range from 100 SLOC to 10000 SLOC.

Average Microservice Size

Consequently, the resulting microservice sizes are checked against the values reported in [SCL15] in order to indicate whether the granularity of the recommended microservices corresponds with the granularity of microservices in practice. To that end, the *average microservice size* metric is introduced, denoted by ams. Let $S_1, S_2, \ldots, S_n$ be the resulting microservices in a recommendation d_M for a monolith M. Furthermore, let $loc(S_i)$ be a function that returns the size of a microservice in SLOC (source lines of code). Then, the ams metric is defined as:

$$ams(d_M) = \frac{\sum_{i=1}^{n} loc(S_i)}{n} \tag{8.1}$$

8.3.2 Team Aspect

In the background chapter, one of the factors that was introduced as a benefit and motivation behind the concept of microservices was the improved team structure argument. Migrating to a microservice architecture has the potential of minimizing the interaction with other teams responsible for other services and domain capabilities. Minimizing the interaction with external teams directly translates to a reduced communication overhead and thus more productivity and focus of the team can be directed towards the actual domain problem and service it is responsible for. As a proxy for communication overhead and interaction complexity between the teams

of different microservices , the evaluation in this thesis uses team size metrics. In the following, two metrics are defined that help to evaluate the resulting microservice recommendations with respect to team and communication complexity.

Average Contributors per Microservice

The *average contributors per microservice*, denoted by cpm, denotes the average number of code authors that worked on class files in a microservice. This metric is always computed for an entire recommendation. To define the metric formally, let $S_1, S_2, \ldots, S_n$ be the recommended microservices in a recommendation d_M for a certain monolith M. Furthermore, let $c(S_i)$ be a function that returns the number of contributors that authored changes to a microservice S_i. Then, the contributors per microservice $cpm(d_M)$ for that recommendation is defined as:

$$cpm(d_M) = \frac{\sum_{i=1}^{n} c(S_i)}{n} \tag{8.2}$$

Contributor Overlapping between Microservices

The second metric named *contributor coupling between microservices*, denoted by cov, describes the average number of authors that are shared between the resulting microservices. Shared contributors act as an indicator of communication between teams of different microservices. A high number of shared authors or contributors implies a large communication overhead between the teams of the microservices, while a low number of shared number of authors consequently indicates a cleaner seperation and allocation of teams to microservices. Again, let $S_1, S_2, \ldots, S_n$ be the recommended microservices in an recommendation d_M for a certain monolith M and let $cont(S_i)$ be a function that returns the set of contributors that authored changes to a microservice S_i. Informally, the contributor overlapping $cov(d_M)$ for a recommendation d_M is defined as the sum of all cardinalities of the intersections between all possible pairs of microservices, divided by the number of possible pairs of microservices. Formally, this means:

$$cov(d_M) = \frac{\sum_{i \neq j} |cont(S_i) \cap cont(S_j)|}{\frac{n \cdot (n-1)}{2}} \tag{8.3}$$

The denominator in Equation 8.3 can be derived by looking at the possible combination of distinct microservices as a graph. If there are n microservices that we imagine as nodes in a graph, the fully meshed graph made up of those nodes will have $\frac{n \cdot (n-1)}{2}$ edges or pairings between them.

Combined Team Metrics

Since there are no absolute values against which the two above-mentioned team metrics can be evaluated, they need to be set into a relative context for increased expressiveness.

External Communication Ratio
To assess how good or bad the relation between the internal team size and the communication overhead for each team of the microservice is, the *external communication ratio* is introduced and denoted by ecr. It is defined as the ratio between the contributor overlapping metric $cov(d_M)$ of a recommendation and the average contributor per microservice metric $cpm(d_M)$:

$$ecr(d_M) = \frac{cov(d_M)}{cpm(d_M)} \tag{8.4}$$

An ecr value of 1 means that the resulting microservice recommendation does a poor job of allocating teams to microservices, since the amount of external communication overhead has the same scale as the size of the microservice teams themselves. An ecr value of 0 would mean an optimal – and in practice impossible – case where the microservice recommendation requires no communication between the teams at all. On this normalized scale between 0 and 1, the evaluation considers values below 0.5 to be *acceptable* while recommendations with an $ecr > 0.5$ should be *rejected*.

Team Size Reduction Ratio
In order to evaluate if the team complexity and team size has decreased and hence improved, the original team size of the monolith M is compared to the average contributors per microservice in the recommendation d_M of the monolith. To that end, the *team size reduction ratio* denoted by tsr is introduced. Let $cont(M)$ be a function that returns the size of the team that worked on the monolith M. Then the tsr is defined as:

$$tsr(d_M) = \frac{cont(M)}{cpm(d_M)} \tag{8.5}$$

8.3.3 Domain Aspect

As outlined in Section 2.2.1 and following, one of the main design principles when developing applications in a microservice architecture is the concept of bounded context. According to domain-driven design [Eva04], the problem domain of an application can be viewed as a a set of bounded contexts, with each of the bounded contexts having clear responsibilities and clearly defined interface that defines which model entities are to be shared with other bounded contexts [New15]. Furthermore, the single responsibility principle [Mar02] commands that a microservice has a concise and clearly defined responsibility. Also, a favorable microservice design avoids duplication

of responsibilities across services. Thus, the evaluation uses a metric as a proxy to indicate the amount of domain-specific duplication or redundancy between the services.

Average Domain Redundancy

To quantify the amount of repetition and redundancy in domain-topics occurring in the microservices, the *average domain redundancy* is measured. The average domain redundancy of a recommendation d_M, denoted by $adr(d_M)$, is computed by averaging the similarities between all pairs of microservices (S_i, S_j) with $i \neq j$. The similarity between two microservices is computed by the cosine similarity method analogous to the technique presented in Section 5.2 but is applied to entire source code of the microservices instead of class files.
The similarity is computed with the tf-idf formulas 5.2 and 5.3 presented in Section 5.2 and always takes a value between 0 and 1. If $sim(S_i, S_j)$ denotes a function that returns a similarity between two microservices S_i and S_j, the average domain redundancy is computed by:

$$adr(d_M) = \frac{\sum_{i \neq j} sim(S_i, S_j)}{\frac{n \cdot (n-1)}{2}} \tag{8.6}$$

where n denotes the number of services resulting in the recommendation d_M.

8.4 Results

8.4.1 Average Microservice Size

Figure 8.5 shows a boxplot of the distribution of the average microservice size (ams) for the sample presented in the beginning of this chapter. The presented ams values are in terms of SLOC (source lines of code). Each project in the sample was tested with the prototype using 7 different extraction strategies or combinations thereof. The strategies are abbreviated by CC for the contributor coupling, LC for the logical coupling and SC for the semantic coupling strategy.
For all of the strategy combinations, the lower and upper quartile lies between 100 and 1000 LOC. This indicates that the bulk part of the evaluated samples result in microservices with average sizes comparable to the preferred microservice sizing reported by 43 % of microservice architects and developers in one of the few empirical surveys on the topic [SCL15]. This is a sign that the recommendation method presented in this thesis is sound and satisfactory with respect to microservice sizing.
The boxplot exhibits several outliers towards higher average microservice size values, with two instances having values greater than 4000 LOC. These anomalies can be attributed to the basic extraction model which considers entire classes as nodes. This means if the coupling strategy produces a graph representation where such large classes occur as nodes, the entire class will end up in a certain microservice and thus cause a rise in the average microservice size. Especially projects with larger or longer repository history, such as *redmine*, are susceptible to this kind of behaviour since the long repository history implies an aged architecture and thus potentially

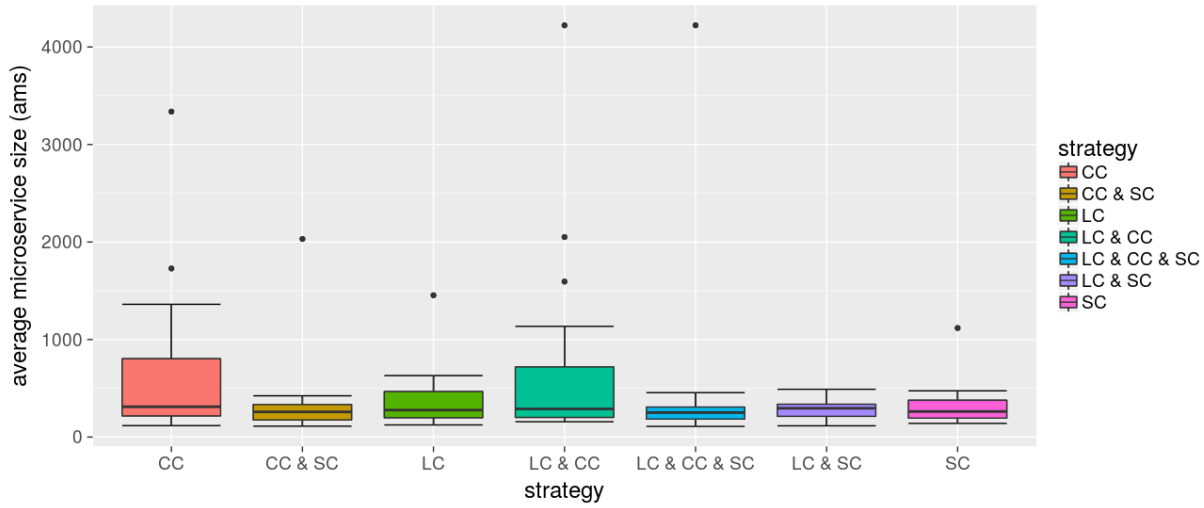


Figure 8.5: Boxplot of the average microservice size (ams) results for the sample projects

detrimental artifacts such as very large classes with a high number of changes and a high number of contributors. For both the logical and contributor coupling strategies this means that the resulting coupling graph will have a very high degree of connectivity and therefore the tendency to form microservices with higher size. It is therefore no surprise that both the extreme outliers with $ams > 4000$ LOC are observed in experiments where contributor and logical coupling strategies are combined, as in *LC & CC* and *LC & CC & SC*.

Another observation that stands out from Figure 8.5 is the fact that the median value of the average microservice size is very similar across all strategies and combinations. The box areas from lower to upper quartile are also fairly uniform in size and position with only slight variations. This indicates that the strategy combination that leads to the microservice recommendation has only partial influence on the resulting size of the microservices. This conjecture is further confirmed by the t-tests performed to compare pairs of strategy combinations to determine whether they're significantly different. The p-values in the t-test results for the ams in table A.2 are consistently lower than 0.05 and therefore indicate that there is no significant difference between the strategy combinations with respect to average microservice size. Rather, the influencing factors are the clustering parameters, as presented in section 6. Specifically, the *numPartitions* parameter that determines how many edge-deletion iterations are performed on the minimum spanning tree of the graph representation in the clustering algorithm and the parameter s that defines the maximal connected component size for handling of extraordinarily large components have more influence on the resulting average microservice size than the strategies used.

Combining this last observation with the sizing assessment in comparison to empirical data above, one can conclude that the approach and the implemented protoype produces satisfactory recommendations when it comes to microservice sizing, as long as the input parameters for the MST clustering algorithm are chosen appropriately.

8.4.2 Team Size Reduction

The evaluation results of the team size reduction metric for the different extraction strategy combinations are provided in Figure 8.6. A tsr value of 1 would imply that the new team size per microservice is as large as the original team size for the monolith, while a lower value towards 0 would mean a clear reduction of the team size.
All of the experiment series exhibit median values below 0.5, independent of the strategy combination used. All but one of the combinations also have the upper quartile below a tsr of 0.5. These observations indicate that the extraction approach and strategies perform very well with respect to team size improvement; the team size shrinks down half of the original size or even lower in the majority of the experiments. Thus also potentially simplifying collaboration structure and development processes for the corresponding microservices.
Taking a look at the discrepancies between the results of the different basic extraction strategies, it is clear that the semantic coupling strategy shows the best results in this metric, by significantly outperforming both the logical and contributor coupling strategies. The t-test results in table A.3 support this claim: The test involving the LC and SC strategies has a p-value of 0.025, while the comparison of the distributions for the CC and SC experiments shows an even more significant difference with a p-value of 0.001.

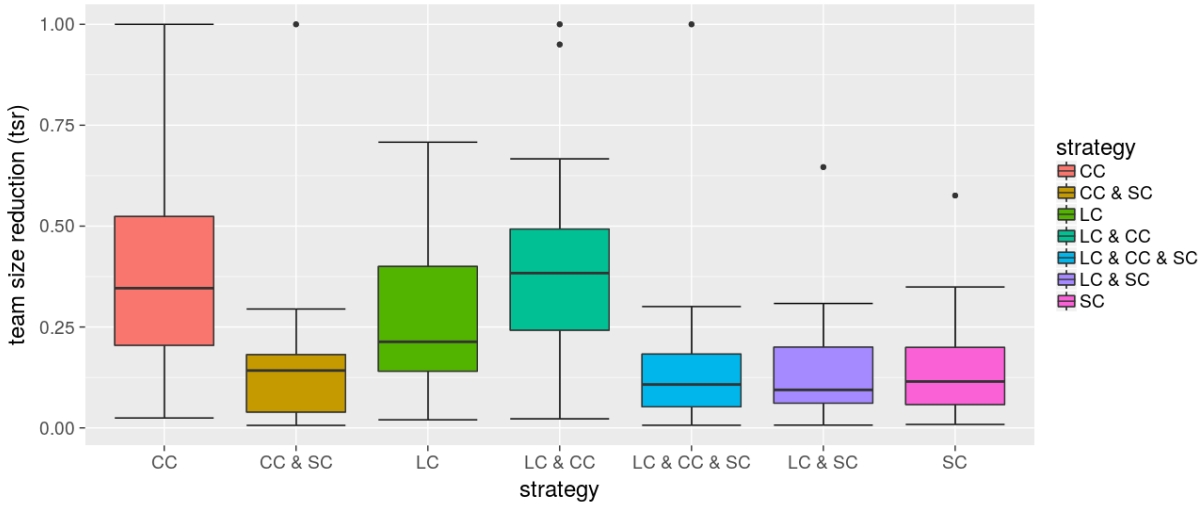


Figure 8.6: Boxplot of the team size reduction ratio (tsr) results for the sample projects

The strategy combinations where the *SC* strategy is involved appear to consistently yield bet-

ter team size reduction than the rest. A look at t-test results in table A.3 shows for instance that the distributions for the *LC & CC* and the *LC & CC & SC* are significantly different with a p-value of 0.004. Similar differences are implied by the p-values of the tests involving the other combinations of the *SC* strategy.
Besides the occasional outliers, an observation that stands out is the higher upper quartile value and the significantly higher upper whisker for the contributor coupling strategy, denoted by *CC*. Upon detailed inspection of the experiment results, it becomes apparent that there are experiments for certain projects where the team size reduction ratio is equal to 1, meaning that there has been no reduction whatsoever. It is also interesting to observe that this occurs only in experiments where the contributor coupling strategy is involved. The explanation for this is found by looking at the coupling graph of the repositories before the MST clustering algorithm is applied. In all of the cases where the tsr resulted in a value of 1, the coupling graph consisted of only one large connected component with star shape. An example for this is shown in Figure 8.7.

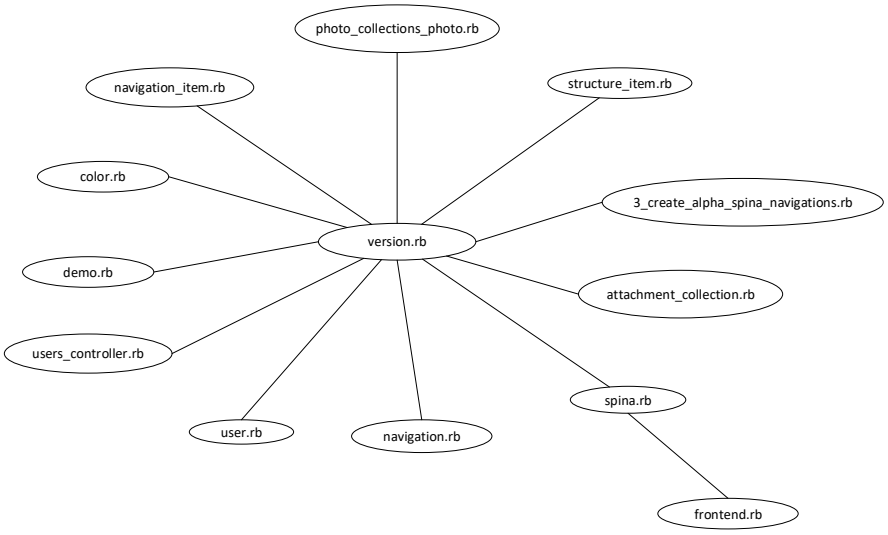


Figure 8.7: Illustrative example for centralized star structure on a coupling graph

The high-degree node in the center corresponds to a class file that has been changed by every single one of the contributors that participated in the history of the monolith. Following the definition of the contributor coupling, this file will be coupled to every other class file in the monolith. The large number of contributors on the central file will also cause the couplings towards this file to be very high in terms of weight, such that the minimum spanning tree results in a star shape

as in Figure 8.7. The weights of the edges towards the outside of the star are weaker than the ones near to the centre. This causes a degenerate behaviour of the clustering algorithm where despite deleting the lowest-weight edge in every step, the number of components does not increase but stays at 1 connected component. Only the outer nodes are iteratively cut away from the star shape. At the end, the remaining graph will still always contain the central class node and it will be the only remaining connected component and hence the only remaining microservice candidate. This remaining candidate will by definition have the same team size as the original monolith since the central node exhibits all of the contributors as the original monolith. This manifests itself in a tsr value of 1 and thus skewing the distribution of the experiments where the contributor coupling is involved towards higher tsr values. A possible countermeasure against these situations is to include the time interval information in the computation of the contributor coupling. This means, detecting couplings only when contributors have changed the same file in the same fixed time interval – similar as in the computation of the logical coupling.
Despite the described phenomenon, the performance of the presented approach with respect to the reduction of team size and complexity is clearly confirmed as satisfactory by the above results.

8.4.3 External Communication Ratio

The external communication ratio acts as a proxy for the amount of interfacing or communication a microservice team is expected to have with the teams of the other recommended generated microservices. It represent the percentage of the team's members that also have links to other microservices, and hence will necessarily generate the need for inter-team communication.

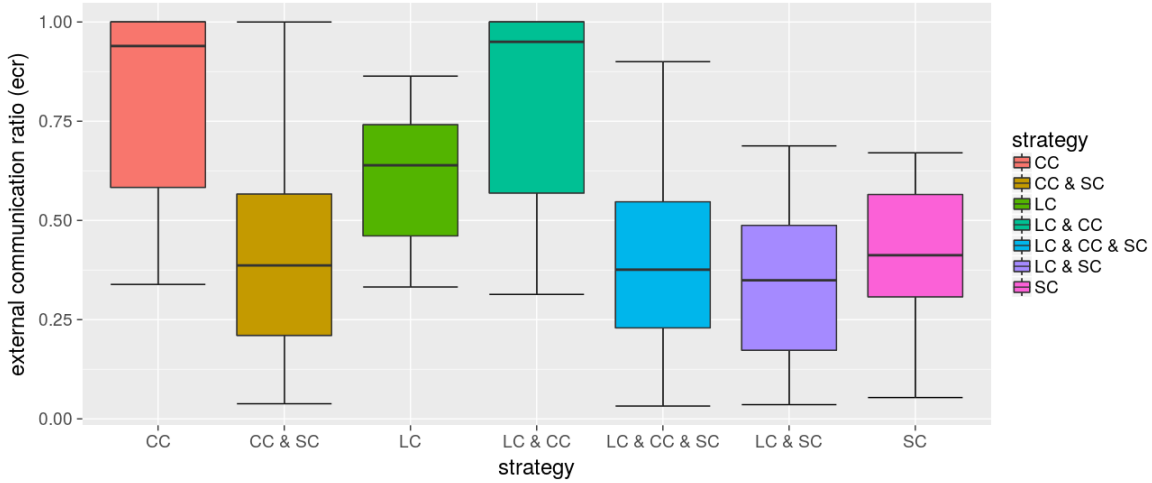


Figure 8.8: Boxplot of the external communication ratio (ecr) results for the sample projects

Figure 8.8 shows that for the majority of the strategy combinations, the external communication ratio is rather low, with median values below 0.5 for the *CC & SC*, *LC & CC & SC*, *LC & SC* and *SC* experiment series. These values can be considered quite satisfactory since the approach that achieved those results did not impose any assumptions or requirements on the prior team structure or development process but still manages to find microservice candidates that exhibit such low ecr values.
There are also less satisfying observations in the evaluation results in Figure 8.8. A closer investigation of the boxplot reveals that some of the upper whiskers and upper quartiles are very high, with values between 0.8 and 1.0 for the whiskers and values very close to 1.0 for the quartiles. Again, it is evident that all but one of the strategy combinations that exhibit these high values involve the contributor coupling strategy. The detailed experiment results provide more insight about this. It becomes apparent that the skew towards high values is caused by several repositories that exhibit ecr values of 1.0 after the microservice extraction. Upon further comparison to the numbers of the previous experiment about the team size reduction, it is clear that these values are caused by the same sample projects. As described in detail in the section above, the contributor coupling strategy produces a resulting graph that consists of only one connected component and hence recommends only that component as a microservice candidate. Since the computation of the ecr metric involves computing the overlapping set of developers among different services, for the special case of one service the value will default to 1.0.
The top 3 performing strategy combinations with respect to the external communication ratio all involve the semantic coupling strategy. This is a pattern that was already observed in the other team-oriented metric in Section 8.4.2. A partial explanation for this lies in the difference between the semantic coupling on one hand, and the other two strategies on the other hand. Both the logical and contributor coupling rely on the revision history of the monolith to generate the coupling weights for the graph. This means, class files related to higher development activities – *i.e.*, larger number of developers contributing to them or a larger number of commits modifying those files – have a higher tendency of ending up in the minimum spanning tree and hence in the microservice recommendations. Therefore, classes coupled by these strategies might have a general tendency to have a higher amount of contributors and hence microservices including these classes will necessarily exhibit larger teams and hence also a tendency of larger communication overhead. Meanwhile the semantic coupling strategy does not favor classes with larger or lower number of contributors or changes in the revision history. It is based solely on the semantic content of the class code. The semantic similarity proves to be a very well-performing heuristic for microservice recommendation with respect to team-aspects, as illustrated by the evaluation results for the tsr and ecr metrics. These observations are statistically confirmed by the t-test results for the ecr distributions in table A.4. Whenever the results for a combination involving the *SC* strategy are compared to series without the *SC* strategy, the p-values are very low, implying significant differences.

8.4.4 Average Domain Redundancy

Section 8.3.3 introduced and motivated the average domain redundancy metric as a proxy for the amount of repetition and duplication with respect to domain concepts in the source code of different microservices. Recalling the often-used catchphrase to define microservices as separately deployable units of componentization that focus on *doing one thing well* [Fow14], it is clear that it is desirable for the average domain redundancy to be as low as possible.
Figure 8.9 shows a boxplot of the adr metric for the evaluation sample computed for all different combinations of the three basic extraction strategies.

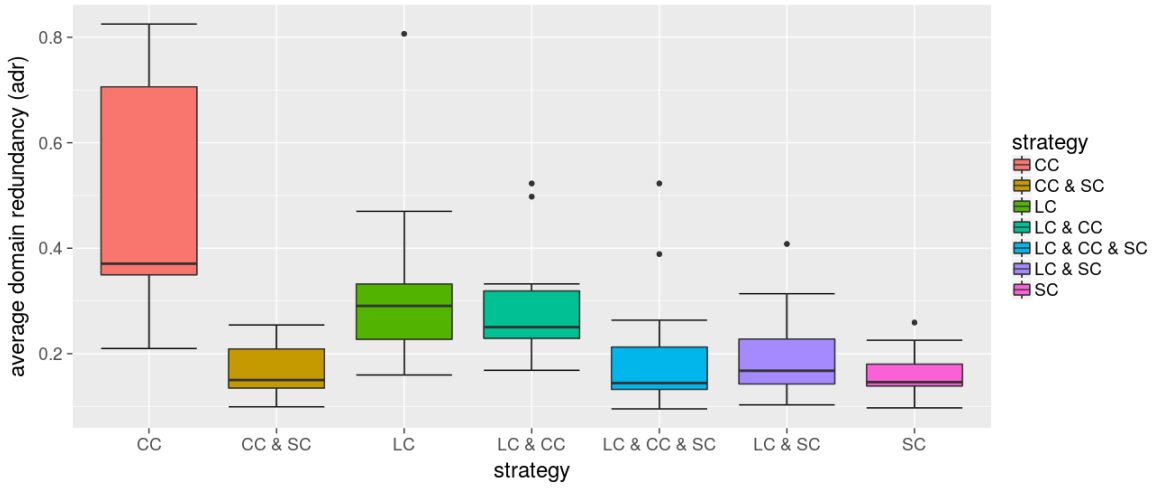


Figure 8.9: Boxplot of the average domain redundancy (adr) results for the sample projects

The boxplot in Figure 8.9 exhibits a few very interesting patterns. The first observation is the fact that 6 out of the 7 strategy combinations deliver very satisfactory domain redundancy distributions. The median of 4 out of those 6 experiment series is significantly lower than 0.25, meaning that considerably less than a quarter of the domain content in the microservice source code is redundant between the services in the recommended extraction. Furthermore, one might intuitively assume that the semantic coupling strategy will perform very well on this metric – and rightfully so – because it optimizes the coupling graph for domain-specific clustering. A look at the results in Figure 8.9 supports this intuition: The adr values produced by the *SC* strategy in isolation have the lowest median. The t-test results partially confirm this observation with p-values below 0.05 for 3 out of 5 comparisons involving the isolated *SC* experiments. The two cases where the p-value is above 0.05 involve the *SC* strategy in both comparison data sets, once isolated and

once in combination with other strategies. This explains the less significant difference in these t-tests.
The only strategy that falls out of the consistently satisfying results is the contributor coupling strategy (CC). Despite the median of adr for the CC experiments being well below the hard threshold of 0.5, the upper quartile is extremely high and the results show generally a stronger skew to higher values than for the other strategies. This phenomenon can be attributed to the fact that the skill distribution among developers in open-source projects is not domain-oriented but rather technology-dependent. This means, front-end developers tend to contribute to the same files independent of the domain content of those files. The same holds analogously for back-end or database developers. Due to the way the contributor coupling is computed, this situation leads to class files being coupled into the same microservice despite having low semantic and domain-specific cohesiveness. Consequently, this leads to higher domain redundancy among different service candidates.
Generally, the observed adr results are very promising and show that both the change information of a repository and the contents of its source code are yielding sources of information to generate microservice recommendations that are well-separated in a domain-oriented sense.

Chapter 9

Conclusion

This thesis focused on finding a formal way to extract microservice recommendations from the source code and revision history of monolith code bases. Chapter 2 introduces the basic concepts for monoliths and microservice architectures and outlines the major trade-offs. The related work is presented in Chapter 3. Promising techniques, similar approaches and previous attempts at related problems from disciplines such as software maintenance, reverse engineering and service decomposition are outlined and discussed. The prototype and the underlying formal extraction model presented in this thesis is the first of its kind in the area of microservice extraction from monoliths. As discussed in more detail in Section 3, there have been recent attempts at the same problem such as *ServiceCutter* [GKGZ16], but none of these rely on an algorithmic analysis of static repository information such as source code and revision history. The related approaches discussed in Chapter 3 leave the generation of the necessary coupling information and representation of the monolith to the user, while the extraction model and the strategies presented in this thesis provide an automatic and implementable way of mining repositories for microservice extraction, thus greatly reducing the complexity and overhead for the architect in a refactoring scenario.

Chapter 4 introduces the two-stage extraction model, which provides the formal answer to research question **RQ1**. The first sub-question **RQ1.1** concerning the extraction strategies that aggregate structured information from static sources about a monolithic code base is answered in detail in Chapter 5, where three strategies are presented: the logical coupling strategy, the contributor coupling strategy and the semantic coupling strategy. The graph-based model allows for future work on the extraction strategies to be applied and integrated to the extraction model very easily. Chapter 6 explains the algorithm used to extract microservice candidate from the constructed graph model and thus constitutes the contribution for **RQ1.2**.

The next chapters 7 and 8 contribute answers for the second research question and its sub goals. Information with respect to **RQ2** about the implementation such as its architectural design and complexity are given in Chapter 7. Chapter 8 is concerned with the evaluation of the prototype in order to answer the research questions **RQ2.1** concerning the performance evaluation, and **RQ2.2** for the evaluation of the quality of the recommended microservice candidates.

9.1 Outcomes

In this section, the outcomes and answers to the research questions are discussed.

RQ1: What is the design of a formal extraction model that uses static information to extract microservices from monolithic code bases?

Outcome: The constructed formal extraction model represents a monolith as a undirected, weighted graph of class nodes, where the edge weights are determined by coupling strategies that mine the monolith for static information.

RQ1.1: What formal strategies can be constructed to mine monolithic code bases for information that helps in the extraction of microservices?

Outcome: Three extraction strategies were found: The *logical coupling*, based on co-change of source files, the *contributor coupling* based on the sets of shared authors among source files, and the *semantic coupling* strategy which is based on the domain-specific identifier contents of the source files.

RQ1.2: What algorithm can use the information aggregated by the extraction strategies to extract microservice candidates?

Outcome: The extraction is performed by a minimum-spanning-tree based edge-deletion algorithm that removes the edges with the lowest couplings until the desired structure is achieved. The resulting connected components correspond to microservice recommendations.

RQ2: How can we build a research prototype that implements the formal extraction model and automatically detects candidates for microservice extraction?

Outcome: The prototype is implemented as a three-tiered web application based on RESTful principles. It uses the Spring framework as a basis for the back-end and AngularJS with HTML5 as a front-end technology.

RQ2.1: What is the performance of the implemented prototype with respect to execution time?

Outcome: The performance of the strategies depends on different factors such as the history size of the monolith, the number of classes in the monolith and the total number of LOC. The overall performance is satisfying and there are no instances of unacceptable performance behaviour in the sample.

RQ2.2: What is the quality of the microservice recommendations generated by the prototype?

Outcome: Four quality metrics were introduced to evaluate the microservice recommendation quality. With respect *average microservice size*, the results indicate that the recommended microservices conform with microservice sizing reported by empirical studies of industry practices. The *team size reduction ratio* shows that the each of the strategy combinations produces microservices that allow the team size to be lower than half the original monolith's team size. The *average domain redundancy* results indicate that a majority of the strategy combinations yield microservices that have very low domain redundancy and repetition.

Discussion

For the logical coupling and contributor coupling, the performance evaluation indicates that there is a growth of the execution time with the size of the revision history that is being analyzed. Anyhow, this is not the entire picture. The graphs in Section 8.2.1 and 8.2.2 show variations and spikes that are caused by other factors such as the number of classes in the monolith or the size of the average change set. Nevertheless, all the performance experiments indicate that the protoype has a satisfying performance for interactive usage in a refactoring scenario.

There are several outcomes and observations from the evaluation of the resulting microservice properties. The *average microservice size* experiment results indicate that the size of the recommended microservices does not vary significantly with different extraction strategies. Rather, the clustering algorithm and its parameter configuration is the defining factor. The achieved microservice sizes comply with previously empirically surveyed microservice sizing data, thus delivering satisfying microservice sizing. The *team size reduction* measurements show that the extraction tool manages to drive down the microservice team size to a quarter of the monolith's team size or even lower. This holds for all extraction strategies except the contributor coupling strategy. It was discovered that the contributor coupling strategy creates degenerate graphs when there are special class files that have had at least one contribution by every author in the monolith's team. It leads to a star-shaped coupling graph that is always reduced to one microservice recommendation with poor team-aspect metrics. Nevertheless, the general team size reduction factor across all experiments is very promising and satisfying. Another team-oriented property of the recommended microservices is investigated with the *external communication ratio*. While the results in this category are not as convincing as for the other metrics, a majority of the strategy combinations still exhibit satisfactory median values where the external communication ratio is kept at 0.5 or lower. This indicates that the communication overhead for the generated microservice teams is kept at an acceptable rate. Again, the mentioned weakness of the contributor coupling strategy manifests itself in skewed results for the combinations involving that strategy. Redundancy of domain-aspects between the resulting services was the fourth metric used to evaluate the recommendations for the sample projects. Six of the seven strategy combinations perform very well in this metric; the reported median values for domain redundancy are consistently at 0.3 or even lower for most cases. Only recommendations generated with the contributor coupling alone without any other strategy show more widely distributed results that are skewed towards

higher redundancy. Surprisingly, a combination of the logical, contributor and semantic coupling performs slightly better than the semantic coupling alone, even though the semantic coupling metric is intuitively expected to outperform all other variants in this metric.
In general, the outcomes can be summarized as follows. The observations during the derivation of the algorithms and the evaluation show that the choice and design of the coupling strategies is only one side of the coin. The clustering algorithm used on the coupling graph in the second stage proved to be equally determining when it comes to the recommendation results. Over all four qualitative metrics, the microservice recommendations for the sample showed satisfying performance, thus successfully validating the approach presented here. There were relative discrepancies between the strategies and combinations thereof. Across all the experiments, the contributor coupling strategy was consistently underperforming the other strategies. On the other end, the *SC*, *LC & SC* and *LC & CC & SC* combinations consistently outperformed the other strategies, while not showing any significant differences between each other.

9.2 Limitations and Future Work

Despite the generally satisfying observations in the evaluation, the approach presented here has its limitations. One limitation is the fact that the extraction model is based on classes as the atomic unit of computation in the strategies and the coupling graph. While this premise lends itself nicely to a graph-based extraction, it limits the available leeway when refactoring monoliths. Using methods, procedures or functions as atomic units of extraction would potentially greatly improve the granularity and precision of the code rearrangement and reorganization. The presented strategies and algorithms would not change much conceptually, but the implementation of the approach on a method level would be much more involved and the execution time performance would very likely suffer.
The microservice candidates generated by the tool presented in this thesis are only to be interpreted as recommendations and not are not to be viewed as finished and fixed microservices that can be applied. Rather, it acts as a refactoring support tool that points the architect in the right direction when trying to discover which classes belong to the same microservices. While the graph-based representation of the monoliths and microservices is a very flexible and promising for formal analysis and algorithms such as the ones performed here, it abstracts away a lot of the aspects that play a role in a microservices environment. In reality, microservices are more than just collections of classes. Usually, a properly designed microservice has a clearly defined external interface and a data storage that it owns.
The extraction model presented here circumvents the data and storage decomposition challenge by assuming the presence of an object-relational-mapping system that represents data model entities as classes that are treated just like any ordinary class by the clustering algorithm. Of course, microservices in practice will often lack such an ORM infrastructure, and hence the unsolved problem of how to share or assign pre-existing databases to different services remains a limitation of this thesis and a challenge for future work in the area.
Furthermore, the resulting microservice candidates lack any information about which parts of the

classes contained in the microservices are exposed to external consumers via an API and which are kept encapsulated inside the service. While one could argue that the collection of all public interfaces of all the classes in a microservice could be viewed as that services' public API, this would lead to service API's that are too fine granular and expose too much functionality. This is clearly a limitation of the approach presented here, and leaves room for further work on the problem of automatically generating service API's from monolithic source code. A limitation that is somewhat related to the previously mentioned limitation of lacking public service APIs has to do with the inter-service dependencies in the recommendation. The resulting microservice candidates carry no information whatsoever as to which connections between the services themselves exist. This is due to the fact that the extraction model builds a graph representation of the monolith based solely on static repository and source code information. In a microservices world, links between microservices – *e.g.*, services calling another to compose use cases or functionality – are a inherently dynamic aspect, and therefore cannot be recovered using static techniques as in this thesis. A possible way to tackle this limitation is to let the user of the refactoring tool provide the necessary calling and link information.

The clustering algorithm that divides the coupling graph into microservice candidates operates on the minimum spanning tree of the coupling graph, and not on the entire graph. This was introduced to guarantee that every edge deletion will lead to an increase in the number of connected components. There is a drawback with this technique. By only considering the minimum spanning tree, a great part of the couplings computed before are simply left out of the further extraction. In other words, some coupling information is lost. Of course, the edges and weights that are ignored are the lowest by definition of the minimum spanning tree and hence the effect of those edges might have been limited anyway if they were further considered in the extraction. Nevertheless, it is an option for future work to look for clustering algorithms that make better use of the aggregated coupling information.

The evaluation presented for the prototype evaluates the implemented tool with respect to performance and recommendation quality. But evaluation is strongly geared towards the extraction strategies. A more detailed study of the impact of the clustering algorithm parameters, such as the number of partitions or the maximum connected component size would likely lead to a better understanding of the extraction behaviour. Similarly, the configuration of the combination weights for the different strategies is also not yet studied in depth. A quantitative analysis of experiments run with different weights for the different strategies when using all the strategies in combination is necessary to utilize the presented approach to its optimal potential. To make the clustering approach less sensitive to initial conditions and remove the need for initial parametrization of the clustering steps, a score-guided system is a promising approach. In such a system, the clustering algorithm continuously computes quality metrics such as the ones presented in the evaluation chapter. The extraction ends when satisfying metric values are achieved. Implementation-wise, the graph based procedures used throughout this work would profit from the use of a proper graph-processing framework. Graph processing frameworks such as *Signal/Collect* [SBC10] enable graph processing at a much larger scale and provide efficient abstractions for graph-related problems and helpful infrastructure such as score-guided execution and convergence detection.

Appendix A

Appendix

A.1 Repository Source List for Sample Projects

Table A.1: Repository locations for the sample projects

Project Name	Source
BroadLeaf DemoSite	https://github.com/BroadleafCommerce/DemoSite
Mayocat Shop	https://github.com/mayocat/mayocat-shop
OpenCMS	https://github.com/alkacon/opencms-core
TNTConcept	https://github.com/autentia/TNTConcept
PetClinic	https://github.com/spring-projects/spring-petclinic
Sunrise Shop	https://github.com/commercetools/commercetools-sunrise-java
Helpy	https://github.com/scott/helpy
Spina	https://github.com/denkGroot/Spina
Sharetribe	https://github.com/sharetribe/sharetribe
Hours	https://github.com/DefactoSoftware/Hours
Rstat.Us	https://github.com/hotsh/rstat.us
Kandan	https://github.com/kandanapp/kandan
Fulcrum	https://github.com/fulcrum-agile/fulcrum
Redmine	https://github.com/edavis10/redmine
Chiliproject	https://github.com/chiliproject/chiliproject
DjangoCMS	https://github.com/divio/django-cms/
Django Fiber	https://github.com/ridethepony/django-fiber
Mezzanine	https://github.com/stephenmcd/mezzanine/
Wagtail	https://github.com/wagtail/wagtail
Mayan	https://gitlab.com/mayan-edms/mayan-edms
Django-Shop	https://github.com/awesto/django-shop
Django Oscar	https://github.com/django-oscar/django-oscar
Taiga Black	https://github.com/taigaio/taiga-back
Django-Wiki	https://github.com/django-wiki/django-wiki

A.2 Welch T-test results for the average microservice size (ams)

Table A.2: T-test results for all pairs of strategy combinations with respect to average microservice size

data set x	data set y	mean x	mean y	t-value	p-value	df	95 % conf. interval
LC	CC	372.823	641.915	-1.519	0.141	25.719	[-633.394, 95.208]
LC	SC	372.823	315.950	0.732	0.469	36.180	[-100.647, 214.392]
CC	SC	641.915	315.950	1.901	0.070	22.959	[-28.867, 680.798]
LC,CC	LC	720.307	372.823	1.529	0.141	22.172	[-123.707, 818.677]
LC,CC	CC	720.307	641.915	0.286	0.776	35.856	[-477.286, 634.070]
LC,CC	SC	720.307	315.950	1.814	0.084	20.623	[-59.812, 868.527]
LC,CC	LC,SC	720.307	281.256	2.000	0.060	19.420	[-19.776, 897.878]
LC,CC	CC,SC	720.307	344.224	1.576	0.127	26.069	[-114.357, 866.523]
LC,CC	LC,CC,SC	720.307	446.580	0.925	0.361	37.705	[-325.583, 873.037]
LC,SC	LC	281.256	372.823	-1.361	0.186	25.156	[-230.053, 46.921]
LC,SC	CC	281.256	641.915	-2.159	0.043	20.769	[281.256, 641.915]
LC,SC	SC	281.256	315.950	-0.685	0.499	29.596	[-138.153, 68.766]
LC,SC	CC,SC	281.256	344.224	-0.636	0.532	20.041	[-269.425, 143.490]
LC,SC	LC,CC,SC	281.256	446.580	-0.822	0.421	19.501	[-585.563, 254.915]
CC,SC	LC	344.224	372.823	-0.248	0.806	31.584	[-263.363, 206.166]
CC,SC	CC	344.224	641.915	-1.555	0.130	31.789	[-687.754, 92.371]
CC,SC	SC	344.224	315.950	0.266	0.793	25.662	[-190.457, 247.005]
CC,SC	LC,CC,SC	344.224	446.580	-0.461	0.648	27.295	[-557.237, 352.525]
LC,CC,SC	LC	446.580	372.823	0.352	0.728	22.778	[-360.028, 507.543]
LC,CC,SC	CC	446.580	641.915	-0.753	0.456	37.321	[-720.846, 330.175]

A.3 Welch T-test results for the team size reduction ratio (tsr)

Table A.3: T-test results for all pairs of strategy combinations with respect to team size reduction

data set x	data set y	mean x	mean y	t-value	p-value	df	95 % conf. interval
LC	CC	0.276	0.390	-1.591	0.120	36.364	[-0.259, 0.031]
LC	SC	0.276	0.156	2.344	0.025	36.027	[0.016, 0.225]
CC	SC	0.390	0.156	3.590	0.001	29.758	[0.101, 0.368]
LC,CC	LC	0.399	0.276	1.711	0.096	34.890	[-0.023, 0.268]
LC,CC	CC	0.399	0.390	0.106	0.916	38.972	[-0.158, 0.175]
LC,CC	SC	0.399	0.156	3.719	28.337	0.001	[0.109, 0.377]
LC,CC	LC,SC	0.399	0.147	3.776	0.001	29.918	[0.116, 0.389]
LC,CC	CC,SC	0.399	0.164	3.047	0.004	36.516	[0.079, 0.391]
LC,CC	LC,CC,SC	0.399	0.165	3.092	0.004	36.836	[0.081, 0.388]
LC,SC	LC	0.147	0.276	-2.435	0.020	37.251	[-0.237, -0.022]
LC,SC	CC	0.147	0.390	-3.649	0.001	31.368	[-0.380, -0.107]
LC,SC	SC	0.147	0.156	-0.207	0.837	38.416	[-0.099, 0.080]
LC,SC	CC,SC	0.147	0.164	-0.290	0.774	31.114	[-0.140, 0.105]
LC,SC	LC,CC,SC	0.147	0.165	-0.301	0.766	33.276	[-0.137, 0.102]
CC,SC	LC	0.164	0.276	-1.708	0.096	35.986	[-0.245, 0.021]
CC,SC	CC	0.164	0.390	-2.936	0.006	37.706	[-0.382, -0.070]
CC,SC	SC	0.164	0.156	0.142	0.888	29.435	[-0.111, 0.128]
CC,SC	LC,CC,SC	0.164	0.165	-0.002	0.998	36.851	[-0.142, 0.142]
LC,CC,SC	LC	0.165	0.276	-1.742	0.090	37.857	[-0.242, 0.018]
LC,CC,SC	CC	0.165	0.390	-2.979	0.005	38.132	[-0.379, -0.072]

A.4 Welch T-test results for the external communication ratio (ecr)

Table A.4: T-test results for all pairs of strategy combinations with respect to external communication ratio

data set x	data set y	mean x	mean y	t-value	p-value	df	95 % conf. interval
LC	CC	0.494	0.823	-5.448	2.68E-06	40.716	[-0.452, -0.207]
LC	SC	0.494	0.397	1.791	0.080	48.026	[-0.012, 0.205]
CC	SC	0.823	0.397	6.688	6.02E-08	38.660	[0.297, 0.555]
LC,CC	LC	0.808	0.494	4.878	2.20E-05	35.815	[0.184, 0.445]
LC,CC	CC	0.808	0.823	-0.212	0.834	38.468	[-0.163, 0.132]
LC,CC	SC	0.808	0.397	6.090	5.38E-07	35.797	[0.274, 0.548]
LC,CC	LC,SC	0.808	0.345	6.690	7.95E-08	36.417	[0.323, 0.603]
LC,CC	CC,SC	0.808	0.405	4.872	2.25E-05	35.802	[0.236, 0.572]
LC,CC	LC,CC,SC	0.808	0.378	5.878	8.57E-07	37.752	[0.282, 0.578]
LC,SC	LC	0.345	0.494	-2.654	0.011	43.400	[-0.262, -0.036]
LC,SC	CC	0.345	0.823	-7.294	8.93E-09	38.664	[-0.611, -0.346]
LC,SC	SC	0.345	0.397	-0.880	0.384	38.655	[-0.173, 0.068]
LC,SC	CC,SC	0.345	0.405	-0.779	0.442	32.331	[-0.215, 0.096]
LC,SC	LC,CC,SC	0.345	0.378	-0.498	0.621	37.371	[-0.167, 0.101]
CC,SC	LC	0.405	0.494	-1.237	0.226	0.226	[-0.237, 0.058]
CC,SC	CC	0.405	0.823	-5.248	7.62E-06	34.914	[-0.581, -0.257]
CC,SC	SC	0.405	0.397	0.096	0.924	31.245	[-0.146, 0.160]
CC,SC	LC,CC,SC	0.405	0.378	0.333	0.741	34.641	[-0.136, 0.189]
LC,CC,SC	LC	0.378	0.494	-1.904	0.064	38.522	[-0.239, 0.007]
LC,CC,SC	CC	0.378	0.823	-6.386	1.52E-07	38.948	[-0.587, -0.304]

A.5 Welch T-test results for the average domain redundancy (adr)

Table A.5: T-test results for all pairs of strategy combinations with respect to average domain redundancy

data set x	data set y	mean x	mean y	t-value	p-value	df	95 % conf. interval
LC	CC	0.304	0.494	-2.668	0.018	14.255	[-0.343, -0.038]
LC	SC	0.304	0.160	4.671	0.000	23.314	[0.080, 0.207]
CC	SC	0.494	0.160	5.097	0.000	10.346	[0.189, 0.479]
LC,CC	LC	0.296	0.304	-0.154	0.880	19.672	[-0.109, 0.094]
LC,CC	CC	0.296	0.494	-2.621	0.019	16.044	[-0.358, -0.038]
LC,CC	SC	0.296	0.160	3.457	0.006	9.897	[0.048, 0.224]
LC,CC	LC,SC	0.296	0.195	2.379	0.033	13.010	[0.009, 0.192]
LC,CC	CC,SC	0.296	0.164	3.314	0.007	10.436	[0.044, 0.221]
LC,CC	LC,CC,SC	0.296	0.188	2.419	0.028	15.723	[0.013, 0.204]
LC,SC	LC	0.195	0.304	-3.138	0.004	32.687	[-0.178, -0.038]
LC,SC	CC	0.195	0.494	-4.433	0.001	11.532	[-0.446, -0.151]
LC,SC	SC	0.195	0.160	1.785	0.085	27.318	[-0.005, 0.076]
LC,SC	CC,SC	0.195	0.164	1.511	0.141	30.568	[-0.011, 0.074]
LC,SC	LC,CC,SC	0.195	0.188	0.264	0.793	35.698	[-0.052, 0.067]
CC,SC	LC	0.164	0.304	-4.446	0.000	25.131	[-0.204, -0.075]
CC,SC	CC	0.164	0.494	-5.013	0.000	10.554	[-0.476, -0.184]
CC,SC	SC	0.164	0.160	0.277	0.784	32.200	[-0.024, 0.032]
CC,SC	LC,CC,SC	0.164	0.188	-0.931	0.360	26.657	[-0.076, 0.029]
LC,CC,SC	LC	0.188	0.304	-3.092	0.004	37.297	[-0.192, -0.040]
LC,CC,SC	CC	0.188	0.494	-4.443	0.001	12.591	[-0.456, -0.157]

A.6 Installation and Setup of the Prototype

The accompanying prototype comes in two different repositories: `microserviceExtraction-backend` which contains the Spring back-end application and `microserviceExtraction-frontend` containing the front-end written in AngularJS. In the following, the set up and execution of the two components is outlined. Note that the back-end project has to be built and executed before the front-end application.

A.6.1 Back-End

The back-end requires at least *Maven* [1] version 3.3.9 and *Java* version 1.8 to be successfully built and executed.
Navigate to the root of the `microserviceExtraction-backend` directory. Then, build the project by executing:

```
mvn install
```

After the Maven install task reports a successful build, the back-end server can be run with the integrated *Tomcat* container using Maven's Spring-Boot plugin:

```
mvn spring-boot:run
```

If the startup command is successful, the Maven output will report:

```
Tomcat started on port(s):  8080 (http)
```

A.6.2 Front-End

The front-end requires a distribution of the `nodeJS` [2] environment version 6.2.1 or higher and the `npm` [3] build tool on version 2.14.7 or higher.
First, install all the necessary JavaScript dependencies using `npm`:

```
npm install
```

The installation of the dependencies might take a few minutes. Upon successful installation of the npm dependencies, the output will display a list of all the installed dependencies.
Finally, the front-end application can be started using:

```
npm start
```

A successful startup lists the `Access URLs` on the standard output. The default location of the front-end is `http://localhost:5555` and can be accessed with any modern web browser.

[1] `https://maven.apache.org/`
[2] `https://nodejs.org/en/`
[3] `https://www.npmjs.com/`

Bibliography

[Ła15] Weronika Łabaj. Goodbye microservices, hello right-sized services. http://particular.net/blog/goodbye-microservices-hello-right-sized-services, 2015. Accessed: 2016-08-16.

[Bal99] Thomas Ball. The concept of dynamic analysis. In *Software Engineering—ESEC/FSE'99*, pages 216–234. Springer, 1999.

[BDLMO10] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. Software re-modularization based on structural and semantic metrics. In *2010 17th Working Conference on Reverse Engineering*, pages 195–204. IEEE, 2010.

[BDLMO13] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. Using structural and semantic measures to improve software modularization. *Empirical Software Engineering*, 18(5):901–932, 2013.

[Bel16] Donald Belcham. Microservice sizing. http://www.westerndevs.com/microservices-sizing/, 2016. Accessed: 2017-01-25.

[BH98] Ivan T Bowman and Richard C Holt. Software architecture recovery using conway's law. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, page 6. IBM Press, 1998.

[BHJ15] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices migration patterns. Technical report, Tech. Rep. TR-SUTCE-ASE-2015-01, Automated Software Engineering Group, Sharif University of Technology, Tehran, Iran, 2015.

[BHJ16] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[BWZ15] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[BYV+09] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for

delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616, 2009.

[CC90] Elliot J. Chikofsky and James H Cross. Reverse engineering and design recovery: A taxonomy. *IEEE software*, 7(1):13–17, 1990.

[CK94] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[Con68] Melvin E Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

[Cor09] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[Cre14] Stephen Cresswell. The granularity of a microservice. `https://www.guidesmiths.com/blog/post/the-granularity-of-a-microservice`, 2014. Accessed: 2017-01-25.

[Dum04] Susan T Dumais. Latent semantic analysis. *Annual review of information science and technology*, 38(1):188–230, 2004.

[Eva04] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[Fow14] Martin Fowler. Microservices: a definition of this new architectural term. `http://martinfowler.com/articles/microservices.html`, 2014. Accessed: 2016-08-16.

[Gam95] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[GHJ98] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 190–198. IEEE, 1998.

[GJK03] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, pages 13–23. IEEE, 2003.

[GKGZ16] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. Service cutter: A systematic approach to service decomposition. In *European Conference on Service-Oriented and Cloud Computing*, pages 185–200. Springer, 2016.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[GN02] Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002.

[GT08] Michael T Goodrich and Roberto Tamassia. *Data structures and algorithms in Java*. John Wiley & Sons, 2008.

[Hof15] Todd Hoff. Deep lessons from google and ebay on building ecosystems of microservices. `http://highscalability.com/blog/2015/12/1/deep-lessons-from-google-and-ebay-on-building-ecosystems-of.html`, 2015. Accessed: 2016-08-16.

[KAR+11] Ali Kazemi, Ali Nasirzadeh Azizkandi, Ali Rostampour, Hassan Haghighi, Pooyan Jamshidi, and Fereidoon Shams. Measuring the conceptual coupling of services using latent semantic indexing. In *Services Computing (SCC), 2011 IEEE International Conference on*, pages 504–511. IEEE, 2011.

[KDG07] Adrian Kuhn, Stéphane Ducasse, and Tudor Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[KNS+] Raghavan Komondoor, V Krishna Nandivada, Saurabh S Sinha, John Field, and Watson Bangalore-Delhi-TJ. Using slicing to extract online services from batch programs.

[KR87] Dennis Kafura and Geereddy R. Reddy. The use of software complexity metrics in software maintenance. *IEEE Transactions on Software Engineering*, 13(3):335, 1987.

[Lap08] Jean-Claude Laprie. From dependability to resilience. In *38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks*, pages G8–G9. Citeseer, 2008.

[LJK+01] Jong Kook Lee, Seung Jae Jung, Soo Dong Kim, Woo Hyun Jang, and Dong Han Ham. Component identification method with coupling and cohesion. In *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific*, pages 79–86. IEEE, 2001.

[LRM14] Antonio Lima, Luca Rossi, and Mirco Musolesi. Coding together at scale: Github as a collaborative social network. *arXiv preprint arXiv:1407.2535*, 2014.

[LTSL09] Man Lan, Chew Lim Tan, Jian Su, and Yue Lu. Supervised and traditional term weighting methods for automatic text categorization. *IEEE transactions on pattern analysis and machine intelligence*, 31(4):721–735, 2009.

[LTV16] Alessandra Levcovitz, Ricardo Terra, and Marco Tulio Valente. Towards a technique for extracting microservices from monolithic enterprise systems. *arXiv preprint arXiv:1605.03175*, 2016.

[Mar02] Robert C Martin. The single responsibility principle. *The Principles, Patterns, and Practices of Agile Software Development*, pages 149–154, 2002.

[Mau15] Tony Mauro. Adopting microservices at netflix: Lessons for architectural design. `https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/`, 2015. Accessed: 2016-08-16.

[MM01] Andrian Marcus and Jonathan I Maletic. Identification of high-level concept clones in source code. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 107–114. IEEE, 2001.

[Mor15] Ben Morris. How big is a microservice? `http://www.ben-morris.com/how-big-is-a-microservice/`, 2015. Accessed: 2017-01-25.

[MV99] Jonathan I Maletic and Naveen Valluri. Automatic software clustering via latent semantic analysis. In *Automated Software Engineering, 1999. 14th IEEE International Conference on.*, pages 251–254. IEEE, 1999.

[NBZ06] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461. ACM, 2006.

[New15] Sam Newman. *Building Microservices*. " O'Reilly Media, Inc.", 2015.

[Nyg07] Michael Nygard. *Release it!: design and deploy production-ready software*. Pragmatic Bookshelf, 2007.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[PJ16] Claus Pahl and Pooyan Jamshidi. Microservices: A systematic mapping study. In *Proceedings of the 6th International Conference on Cloud Computing and Services Science*, pages 137–146, 2016.

[PM06] Denys Poshyvanyk and Andrian Marcus. The conceptual coupling metrics for object-oriented systems. In *ICSM*, volume 6, pages 469–478, 2006.

[Pre08] Bruno R Preiss. *Data structures and algorithms with object-oriented design patterns in C++*. John Wiley & Sons, 2008.

[PRFT07] Mikhail Perepletchikov, Caspar Ryan, Keith Frampton, and Zahir Tari. Coupling metrics for predicting maintainability in service-oriented designs. In *Software Engineering Conference, 2007. ASWEC 2007. 18th Australian*, pages 329–340. IEEE, 2007.

[Ram03] Juan Ramos. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, 2003.

[Ric14] Chris Richardson. Microservices: Decomposing applications for deployability and scalability. `https://www.infoq.com/articles/microservices-intro`, 2014. Accessed: 2016-08-16.

[RKS+11] A Rostampour, A Kazemi, F Shams, P Jamshidi, and A Nasirzadeh Azizkandi. Measures of structural complexity and service autonomy. In *Advanced Communication Technology (ICACT), 2011 13th International Conference on*, pages 1462–1467. IEEE, 2011.

[RPL08] Romain Robbes, Damien Pollet, and Michele Lanza. Logical coupling based on fine-grained change information. In *2008 15th Working Conference on Reverse Engineering*, pages 42–46. IEEE, 2008.

[RV04] Hajo A Reijers and Irene TP Vanderfeesten. Cohesion and coupling metrics for workflow process design. In *International Conference on Business Process Management*, pages 290–305. Springer, 2004.

[SBC10] Philip Stutz, Abraham Bernstein, and William Cohen. Signal/collect: graph algorithms for the (semantic) web. In *International Semantic Web Conference*, pages 764–780. Springer, 2010.

[SCL15] Gerald Schermann, Jürgen Cito, and Philipp Leitner. All the services large and micro: Revisiting industrial practice in services computing. In *International Conference on Service-Oriented Computing*, pages 36–47. Springer, 2015.

[SRK+09] Santonu Sarkar, Shubha Ramachandran, G Sathish Kumar, Madhu K Iyengar, K Rangarajan, and Saravanan Sivagnanam. Modularization of a large-scale business application: A case study. *IEEE software*, 26(2):28–35, 2009.

[SS02] Eleni Stroulia and Tarja Systä. Dynamic analysis for reverse engineering and program understanding. *ACM SIGAPP Applied Computing Review*, 10(1):8–17, 2002.

[SSB12] Robert F Stärk, Joachim Schmid, and Egon Börger. *Java and the Java virtual machine: definition, verification, validation*. Springer Science & Business Media, 2012.

[Thö15] Johannes Thönes. Microservices. *IEEE Software*, 32(1):116–116, 2015.

[Til14] Stefan Tilkov. How small should your microservice be? `https://www.innoq.com/blog/st/2014/11/how-small-should-your-microservice-be/`, 2014. Accessed: 2017-01-25.

[TVB12] Ricardo Terra, Marco Túlio Valente, and Roberto S Bigonha. An approach for extracting modules from monolithic software architectures. In *IX Workshop de Manutenção de Software Moderna (WMSWM)*, pages 1–8, 2012.

[VRMB11] Luis M Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):45–52, 2011.

[Whe01] David A Wheeler. More than a gigabuck: Estimating gnu/linux's size, 2001.

[ZW04] Thomas Zimmermann and Peter Weißgerber. Preprocessing cvs data for fine-grained analysis. In *Proceedings of the First International Workshop on Mining Software Repositories*, pages 2–6. sn, 2004.