TÉCNICO LISBOA

Monolith Development History for Microservices Identification: a Comparative Analysis

João Pedro de Oliveira Estudante Lourenço

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. António Manuel Ferreira Rito da Silva

Examination Committee

Chairperson: Prof. Nuno Miguel Carvalho dos Santos
Supervisor: Prof. António Manuel Ferreira Rito da Silva
Member of the Committee: Prof. Olaf Zimmermann

October 2022

Acknowledgments

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) through projects UIDB/50021/2020 (INESC-ID) and PTDC/CCI-COM/2156/2021 (DACOMICO).

I would like to express my gratitude to my supervisor, professor António Rito Silva. Without his ideas, attention to detail, and unwavering support and dedication ever since our first meeting, this thesis wouldn't exist.

Thanks should also go to Caniné, Gonçalo, and André, who were my partners in crime for many projects and classes, but above all, who were great supportive friends and always there when needed.

Many thanks to my family, who has always supported me in every step of my personal and academical journey, and made sure my university experience was as stress free as possible.

Lastly, a special thanks to Lara, who has always listened to my rants about computers and work despite not understanding much, and who could always cheer me up even when far.

Abstract

Recent research has proposed different approaches on the automated identification of candidate microservices on monolith systems, which vary on the monolith representation, similarity criteria, and quality metrics used. On the other hand, they are generally limited in the number of codebases and decompositions evaluated, and few comparisons between approaches exist. Considering the emerging trend in software engineering in techniques based on the analysis of codebases' evolution, we compare a representation based on the monolith code structure, in particular the sequences of accesses to domain entities, with representations based on the monolith development history (file changes and changes authorship). From the analysis on a total of 468k decompositions of 28 codebases, using five quality metrics that evaluate modularity, minimization of the number of transactions per functionality, and reduction of teams and communication, we conclude that the best decompositions on each metric were made by combining data from the sequences of accesses and the development history representations. We also found that the changes authorship representation of codebases with many authors achieves comparable or better results than the sequence of accesses representation of codebases with few authors with respect to minimization of the number of transactions per functionality and the reduction of teams. This allows the usage of a collection technique that is easier to apply, as it is independent of the language or framework chosen for the monolith.

Keywords

Monolith; Microservices; Microservices Identification; Architecture Migration; Repository Mining

Resumo

Nos últimos tempos, têem sido propostas várias abordagens distintas com vista à automatização da identificação de candidatos para microserviços de sistemas monolíticos, variando na representação do monólito considerada, critérios de similaridade usados, e métricas de qualidade aplicadas. No entanto, estas abordagens geralmente são limitadas no número de projectos e decomposições avaliados, e existem poucas comparações feitas entre abordagens. Tendo em conta a tendência emergente em engenharia de software do uso de técnicas baseadas na análise da evolução do código de um projecto, nós comparamos uma representação baseada nas sequências de acessos, em particular a entidades de domínios, com representações baseadas na história de desenvolvimento do monólito (alterações de ficheiros e autoria de alterações). Através da análise de 468k decomposições de 28 projectos, e usando cinco métricas de qualidade que avaliam modularidade, minimização do número de transações por funcionalidade, e redução das equipas e comunicação, concluímos que as melhores decomposições de cada métrica foram conseguidas através da mistura de dados da representação de sequências de acessos e da história de desenvolvimento. Também descobrimos que a representação da autoria de alterações em projectos com um número elevado de autores consegue obter resultados comparáveis ou melhores à representação da sequência de acessos em projectos com poucos autores, no que diz respeito à minimização do número de transações por funcionalidade e na redução do tamanho das equipas. Isto permite a utilização de uma técnica de colecção que tem uma aplicação mais abrangente, dado que é independente da linguagem ou framework escolhido para o monólito.

Palavras Chave

Monólito; Microserviços; Identificação de Microserviços; Mineração de repositórios; Migração de Arquitetura

Contents

1 Introduction 1
1.1 Problem . . . 3
1.2 Contribution and research questions . . . 4
1.3 Outline . . . 5

2 Background 7
2.1 Similarity Measures . . . 9
2.2 Quality metrics . . . 11
2.2.1 Complexity . . . 11
2.2.2 Uniform Complexity . . . 12
2.2.3 Cohesion and coupling . . . 13
2.2.4 Team size reduction . . . 13
2.2.5 Combined . . . 14

3 Related Work 15
3.1 Monolith Decomposition Techniques . . . 17
3.1.1 Overview . . . 17
3.1.2 Using development history . . . 18
3.2 Comparison of Techniques . . . 20
3.3 Research Gap . . . 21
3.4 Usage of commit logs for software engineering research . . . 21

4 Implementation 23
4.1 Overview . . . 25
4.2 Data Collection . . . 26
4.3 Data Cleaning . . . 28
4.4 Coupling Computation . . . 29
4.5 Performing a decomposition . . . 29
4.6 Mono2Micro improvements . . . 31

5 Evaluation 33
5.1 Codebase selection and characterization 36
5.2 Results 38
5.2.1 Uniform complexity 39
5.2.2 Cohesion 42
5.2.3 Coupling 45
5.2.4 Team size reduction ratio 48
5.2.5 Combined 51
5.3 Evaluation conclusions 54
5.4 Threats to validity 55
6 Conclusion 57

Acronyms

tsr team size reduction

1

Introduction

Contents

1.1 Problem 3
1.2 Contribution and research questions 4
1.3 Outline 5

In 2014, Lewis and Fowler [1] described microservices, a new architectural style, which was applied at Amazon and Netflix [2]. In such systems, instead of a single unit (a ”monolith”) being responsible for handling all the business logic with a single database, sets of functionalities that implement the logic execute separately in independent services. This brings plenty of advantages, like increased developer productivity, scalability, reliability, maintainability, separation of concerns, and ease of deployment [3]. As such, migrating a monolithic application to a microservice architecture is appealing, and different automated approaches have been proposed [4–6].

These approaches tend to follow a common procedure: (1) collect data; (2) generate a representation of the monolith; (3) define one or more similarity measures for the monolith's elements based on some criteria, and use them to cluster the collected data and generate a decomposition based on the representation and the measures; (4) evaluate the decomposition using one or more quality metrics. However, the techniques at each step vary: as an example, the data collection can be based on the monolith's specifications [5], code static analysis [7], system execution analysis [8, 9], development history [10–13], among others. The monolith representation can be based on a graph [10, 12], a tree [11], or a sequence of accesses [7]. Multiple criteria can used to identify the services of the decomposition [5], like modularity [10], minimization of the number of distributed transactions per functionality [14], or even the reduction of each service's team size [13]. The evaluation metrics also vary significantly [15].

The benefits of a microservice migration come at a cost, namely in terms of performance and complexity with the management of transactions. When migrating from a monolith to microservices, functionalities with ACID properties (Atomicity, Consistency, Isolation, Durability) may be separated. This requires the introduction of checks to ensure that the whole functionality has successfully ran across all microservices, as well as the possibility to roll-back data in case it hasn't. The migration itself also often requires expert architects and tends to be done manually [16], which is tedious and prone to errors and biases. To tackle the problem of microservice identification, the Mono2Micro tool was developed [17, 18]. The tool analyzes a codebase and identifies services based on transactional contexts. This way, it reduces the complexity associated with a migration by keeping in the same microservice the entities that are accessed with the same functionalities, in order to minimize the number of transactions required to implement a functionality.

1.1 Problem

There has been research on the comparison of the use of code static analysis and system execution analysis data collection techniques [19], on development history and lexical monolith's representations [13], and on modularity, team size, and number of transactions criteria [7, 13]. However, this is limited on the number of possible combinations, and so, more research has to be done. In particular,

due to the emerging trend in software engineering in techniques based on the analysis of codebases' development history [20], it became especially interesting to compare approaches that use the monolith code structure, in particular the sequences of accesses to domain entities, which is one the monolith representations more often used, e.g. [21], with representations based on the monolith development history. The latter simplify the collection step, because they are independent of the programming language and technology used in the monolith implementation.

1.2 Contribution and research questions

Considering the lack of in-depth comparisons between a development history monolith representation and others, in terms of number of codebases and decompositions, as well as no data regarding how this representation behaves in terms of reducing the complexity of a migration, we intend to respond to the following general research question:

- **RQ1:** How do monolith microservices identification approaches that use the monolith development history based representations perform when compared with approaches that use the monolith functionalities sequences of accesses representation?

To answer **RQ1**, we started by developing a script capable of parsing a monolith's development history efficiently. We then extended the Mono2Micro monolith decomposition tool so that it was capable of creating decompositions using the data from this new representation, and return quality metrics for each decomposition. The decompositions are then evaluated in great detail in a reproducible fashion using R, according to the criteria of modularity, minimization of the number of transactions per functionality, and reduction of teams and communication, where metrics are used for the assessment. A total of 28 monolith codebases are used for the empirical study.

We leverage the Mono2Micro tool and contribute with:

- The possibility for the tool to decompose monoliths using data from its development history;
- Various significant performance improvements of the tool, by adding parallelization and memoization;
- The collected data from all codebases;
- All the python and bash code used to collect data, with instructions on how to run it, available in Mono2Micro's repository[1];
- A reproducible evaluation package, meant to verify the generation of the same plots and obtained values, which is also available in Mono2Micro's repository[2];

[1] Commit Collection @ socialsoftware/mono2micro
[2] Reproducible package @ socialsoftware/mono2micro

- A comprehensive comparison between the quality metrics of different representations.

1.3 Outline

This chapter presented the context of our work, and the rest of the thesis is structured as follows:

- Chapter 2 describes the similarity measures and the quality metrics we consider.
- Chapter 3 offers an overview of related research work, the existing research gap that we attempt to close, and how we can leverage the existing work to improve our solution.
- Chapter 4 has detailed descriptions and examples of how we implemented our data collection, and how it is used to suggest decompositions of monolith systems. It also describes the improvements made to the Mono2Micro tool.
- Chapter 5 contains a detailed analysis of each quality metric, evaluating the obtained values when considering decompositions from different monolith representations.
- Chapter 6 summarizes the main results and proposes topics for future research.

2

Background

Contents

2.1 Similarity Measures 9
2.2 Quality metrics 11

We leverage on previous work to compare the use of the different monolith representation models for the identification of microservices in monolith systems: (1) the sequence of accesses [7], and (2) code evolution [13]. While the former representation is one of the most widely used by different approaches, the later seems promising in terms of the new trends on code repositories mining and the relative independence of programming languages and software frameworks used.

The use of sequences of accesses to represent a monolith requires the identification of its set of functionalities, F, and the accessed domain entities, E. For each functionality $f \in F$ there is a callgraph, $f.graph$ that captures the sequences of accesses done to the domain entities. Each access, $a \in A$, is a pair (e, m), where $e \in E$ and $m \in \{r, w\}$. Given an entity $e \in E$, $e.funct(m)$ denotes the set of functionalities that have an access in the entity according to access mode m; if m is omitted, then it can be any type of access. Note that this representation can be obtained either through a static analysis or a system execution analysis of the monolith.

On the other hand, the use of the development history monolith representation requires the identification of its set of commits, C, where each commit, $c \in C$, contains the set of files, $c.files \in Fl$ that where changed together, where Fl represents the set of all files in the codebase. Additionally, a commit $c \in C$ author, $c.author \in Au$, belongs to the set Au of codebase developers. Given a file $f \in Fl$, $f.authors$ denote the set of authors that have a commit in the file. Finally, a commit $c \in C$ also contains the time when it occurred, $c.time$.

A decomposition is a partition of the monolith domain entities. A decomposition $d \in D$, where D represents the set of all decompositions, is a set of clusters, $d.clusters \in 2^E$, of the monolith domain entities. Therefore, $\forall_{cl_i, cl_i \in d.clusters} cl_i \cap cl_i = \emptyset$ and $\bigcup_{cl \in d.clusters} cl = E$. Note that for decompositions generated using the development history representation, it is necessary, in some part of the processing pipeline, to filter the files that correspond to the domain entities.

2.1 Similarity Measures

A decomposition generation is driven by similarity measures between representation elements, which can be domain entities or files. The smaller the distance between them, the higher is the likelihood they belong to the same microservice.

The similarity measures for the sequence of accesses representation are defined over the relation between the functionalities and the entities they access, such that the number of distributed transactions per functionality is minimized. Each measure is a value between 0 and 1, where 1 represents very high similarity, and 0 represents no similarity at all.

- Access measure - The access similarity of two entities $e_i, e_j \in E$ depends on the likelihood of

functionalities accessing (by reading or writing) both e_i and e_j:

$$sm_{access}(e_i, e_j) = \frac{\#(e_i.funct \cap e_j.funct)}{\#e_i.funct} \tag{2.1}$$

- Read measure - The read similarity of $e_i, e_j \in E$ depends on the likelihood of functionalities reading both e_i and e_j:

$$sm_{read}(e_i, e_j) = \frac{\#(e_i.funct(r) \cap e_j.funct(r))}{\#e_i.funct(r)} \tag{2.2}$$

- Write measure - The write similarity of $e_i, e_j \in E$ depends on the likelihood of functionalities writing both e_i and e_j:

$$sm_{write}(e_i, e_j) = \frac{\#(e_i.funct(w) \cap e_j.funct(w))}{\#e_i.funct(w)} \tag{2.3}$$

- Sequence measure - The sequence similarity of $e_i, e_j \in E$ depends on the number of consecutive accesses to both entities across all functionalities:

$$sm_{sequence}(e_i, e_j) = \frac{sumPairs(e_i, e_j)}{maxPairs} \tag{2.4}$$

  where $sumPairs$ represents number the consecutive occurrence of accesses to the entities, and $maxPairs$ the maximum of all sums of pairs.

For the development history based representations, the similarity measures are defined on the co-occurrences of changes between two files, and the common authorship of files changes.

The first measure, sm_{commit}, follows the principle that files that were changed together more often are likely to stay in the same cluster. This makes it easier to reason about changes, as they are confined to a set of related files in the same service. The measure of files $f_i, f_j \in Fl$ can be defined as:

$$sm_{commit}(f_i, f_j) = \frac{\#\{c \in C : f_i, f_j \in c.files\}}{\#\{c \in C : f_i \in c.files\}} \tag{2.5}$$

The second measure, sm_{author}, follows the principle that communication overhead should be reduced in a microservice architecture, and as such, developers should be separated in teams where they focus on a reduced number of different services. Therefore, files that were changed by the same developers should stay in the same cluster, which is what the measure captures for files $f_i, f_j \in Fl$:

$$sm_{author}(f_i, f_j) = \frac{\#(f_i.authors \cap f_j.authors)}{\#f_i.authors} \tag{2.6}$$

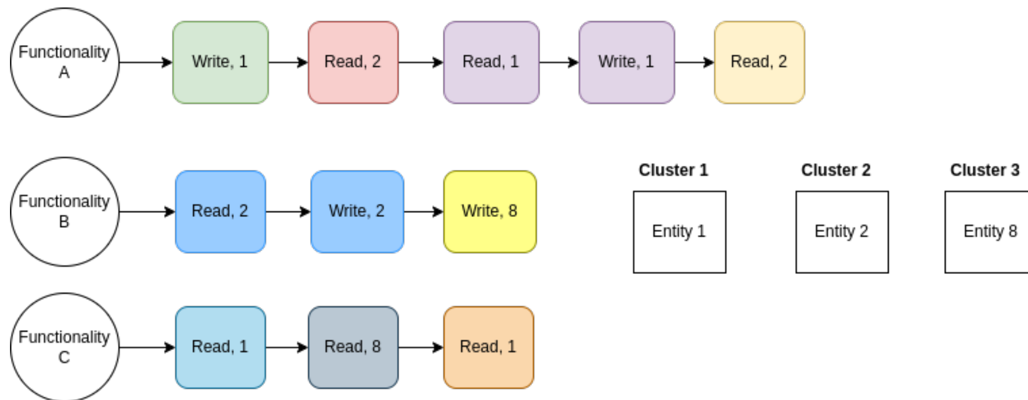

Figure 2.1: Example of a simple monolith sequences of accesses representation, and a decomposition on the right where each entity was assigned to a single cluster. Both the representation and the decomposition are illustrative and serve just as an example.

2.2 Quality metrics

To evaluate a certain decomposition, several quality metrics are defined, each focusing on a particular quality of the decomposition like modularity, migration effort or team reduction size.

2.2.1 Complexity

The complexity of the decomposition is informally defined as the effort required to perform the decomposition, due to the intermediate states that are naturally introduced, as described by the concept of Saga [22, 23] for the implementation of distributed transactions with eventual consistency [24].

Formally, the complexity of a decomposition is based on the sum of the complexity of all functionalities. The complexity of each functionality is based on the sum of the complexity of all of its local transactions. Each local transaction LT:

- is a subgraph of the functionality call graph;
- contains only accesses in a single cluster of the decomposition.
- contains all consecutive accesses to the same cluster.

The complexity of each local transaction is equal to the number of other functionalities that read or write entities that are written or read, respectively, by the local transaction. Repeated accesses of the same mode to the same domain entity are removed, and only one access is considered. If a read access occurs after a write, only the write access is considered. As an example, Figure 2.1 contains three functionalities with several accesses, and three total entities each in its own cluster. The complexity of each functionality is calculated like so:

- Functionality A has four local transactions, and five accesses (the colors represent accesses belonging to the same local transaction). The first LT is a write to Entity 1, and Functionality C performs a read to this entity at least once, so the complexity of this first LT is 1. The second LT is a read to Entity 2, and Functionality B performs a write to Entity 2, so the complexity is 1. The third LT has a read and a write to Entity 1, and Functionality C has a read to entity 1 which is different than a write, so the complexity is 1. Finally, the last LT has a read to Entity 2, and Functionality B writes Entity 2, so the complexity is also 1. At the end, the complexity of Functionality A is **4** (sum of all LT).

- Functionality B has two LT. The first one, with two accesses, a read and a write, has complexity 1: no other functionality writes Entity 2, but Functionality A reads Entity 2. The second Lt also has complexity 1, because Functionality C reads Entity 8. The complexity of Functionality B is **2**.

- Functionality C has three LT. Two of them read Entity 1, and only Functionality A writes Entity 1, so these two each have complexity of 1. The other LT reads Entity 8, and Functionality B writes Entity 8, so its complexity is 1. In total, the complexity of Functionality C is **3**.

The sum of the complexity of all functionalities is 9, which is divided by the number of functionalities to obtain the complexity of the decomposition: 9/3 = 3. The value of the complexity metric [18] increases with the number of intermediate states created by the distributed transactions, because they have to be considered in the implementation of the functionalities business logic, due to the lack of isolation.

2.2.2 Uniform Complexity

As seen before, the complexity of a decomposition is an absolute value, varying between 0 and a number dependent on the codebase. This particularity makes it hard to compare the complexities of decompositions from different codebases.

The uniform complexity is a metric based on the complexity, and is computed by dividing the complexity of a decomposition by the maximum possible complexity that a given codebase can have:

$$uniform_complexity(d) = \frac{complexity(d)}{max_complexity(codebase)} \tag{2.7}$$

This maximum value is achieved when there are as many clusters as domain entities.

The metric varies between 0 and 1, making it suitable for comparing different decompositions and codebases.

2.2.3 Cohesion and coupling

The cohesion indicates the percentage of entities of a microservice accessed whenever there is an access per functionality, and so it varies between 0 and 1. If the cohesion of a decomposition is 1, then each functionality that accesses the cluster, accesses all its entities, so it strongly follows the Single Responsibility Principle [25]. It is defined as:

$$cohesion(c) = \frac{\sum_{f \in funct(c)} \frac{\#\{e \in c.e : e \in G_f.A.e\}}{\#c.e}}{\#funct(c)} \tag{2.8}$$

where $funct(c)$ is the set of functionalities that access cluster c; $G_f.A.e$ is the set of entities that are accessed by functionality f; $c.e$ is the set of entities of cluster c. The cohesion of a decomposition is the average of the cohesion of all clusters.

Coupling is a metric applied to two microservices, and describes the percentage of entities that a service exposes to the other. The coupling of a decomposition corresponds to the average coupling among all pairs of microservices. A coupling of 1 means that all services expose entities to all other services, which is a very high inter-service dependency and an undesirable trait. It is defined as:

$$coupling(c_i, c_j) = \frac{\#\{e \in c_j : \exists_{ri \in RI(c_i,c_j) e = ri[2].e}\}}{\#c_j.e} \tag{2.9}$$

where $RI(c_i, c_j)$ is the set of remote invocations from cluster c_i to c_j. A remote invocation occurs when in a functionality's sequence of accesses there are two subsequent accesses, where their entities belong to different clusters.

2.2.4 Team size reduction

The team size reduction (tsr) indicates if the average team size is reduced in the migration, by comparing the average number of authors per microservice to the total number of authors [13]. A tsr of 1 would indicate no reduction in team size, as all services have the same number of authors that the original monolith had. On the other hand, a tsr close to 0 would indicate an excellent team division after the migration. It is defined as:

$$tsr(d) = \frac{\frac{\sum_{i=0}^{\#d.c} d.c_i.developers}{\#d.c}}{M.developers} \tag{2.10}$$

We divide the total number of developers of all decomposition d's clusters c_i, by the number of clusters, to get the average number of contributors per microservice in the decomposition d. This is then divided by the number of unique developers in the monolith.

2.2.5 Combined

Finally, a combined metric is used to sum up the results of all other metrics. It is a number between 0 and 1, where 0 is a perfect decomposition in all metrics, and 1 is the worst decomposition possible in all metrics. Considering that it is not possible to maximize all quality metrics of a system, this metric represents the trade-offs made. It is defined as:

$$combined(d) = \frac{uniform_complexity(d) + coupling(d) + tsr(d) - cohesion(d) + 1}{4} \quad (2.11)$$

3

Related Work

Contents

3.1 Monolith Decomposition Techniques 17
3.2 Comparison of Techniques 20
3.3 Research Gap 21
3.4 Usage of commit logs for software engineering research 21

3.1 Monolith Decomposition Techniques

3.1.1 Overview

Different approaches exist to decompose a monolith system into a service-oriented architecture, with various input sources, final service granularity, and applicabilities. A recent overview was done in [4], where four main categories are highlighted:

Meta-data aided approaches. These make use of various representations of code, and not the code itself, to suggest decompositions. In [26], it is suggested a manual conceptual approach that makes use of use-cases, security, and scalability requirements to partition a monolith. The authors in [27] make semantic evaluations on a monolith's OpenAPI specification, and suggest a decomposition of interfaces based on the similarity of terms in the specification's descriptions. They then evaluate the decompositions based on granularity, cohesion, and coupling. Service Cutter, presented in [5], is a very thorough tool making use of 16 different coupling criteria to drive decompositions. Different criteria require different inputs, which could be use cases, entity relationships models described in UML, the entities themselves, and so on. Finally, in [13], they extract information from a monolith's development history (stored in a version control system), and extract logical, authors, and lexical relationships between classes.

Static code analysis aided approaches. The focus here is on the code itself, from which a representation is generated and a decomposition is made. In [28], they parse the annotations in classes of Java Enterprise Edition applications to identify domain entities and their types. They are then organized into an Abstract Syntax Tree according to their relationships, and a clustering algorithm extracts microservices suggestions. In [29], the authors extract the dependencies between business functions, databases, and facades, and build a dependency graph based on the relationships between these elements. The microservices suggestions are done through manual code inspection. In both cases, entities play a central role in the whole process.

Workload-data aided approaches. The execution process is analysed to extract relationships, and then a decomposition is derived. Only one approach is described in this category [30], where the execution logs of a web application are parsed and pages that have higher workloads are candidates to be split into microservices.

Dynamic microservice composition approaches. These approaches keep generating a decomposition until achieving a stable and desirable state. In [9], the authors reason that it's not possible to fully capture the expected behaviour and granularity of microservices at design time. Therefore, they create architectural elements with variable boundaries, that get changed according to the behaviour when executing the monolith. In [8], the authors extract features (chunks of functionalities that implement some business logic) and register their usage and performance during the execution of the system. A genetic algorithm reorganizes features in different microservices according to the observed usage and

performance, so that the cohesion is maximized and the coupling is minimized.

In short, the approaches suggest the usage of different information sources to represent the monolith, depending on whether they focus on its structure [5, 26–29], behaviour [8, 9, 30], or development process [13]. Additionally, different decomposition criteria are used, like modularity [5, 9, 13, 26–29], performance [8, 30], team size [13], and security/usage requirements [5, 26].

3.1.2 Using development history

As the present work aims to compare the usage of commits as a means to identify services with another approach, we will explore what has been done in the literature in this regard.

In [13], a model for microservice extraction using the software change history of a monolith is presented. An initial data collection phase consists of parsing the history and extracting all the class files and which other files they changed with, as well as which authors changed each file. Then, a graph is created where the names of the files are the nodes, and the edges indicate the existence of coupling between files. A higher weight in the edge indicates a higher similarity. Three strategies to quantify the similarity of files are suggested:

1. *Logical Coupling* is based on the assumption that software elements that change together do so for the same reason, and so should be placed in the same microservice. The more often a pair of files changes together, the stronger is the similarity;

2. *Semantic Coupling* is based on a rationale that states that each microservice should match a context from the problem domain. To achieve this, classes that contain code about the same domain model entities should belong to the same microservice. NLP techniques are applied to find the degree of similarity between classes;

3. *Contributor coupling* follows from the microservice principle of cross-functional teams, centered around domain and business capabilities. To implement this strategy, all change events that modified a given class are first found. Then, the set of all authors that contributed to those events are computed. The weight of an edge is given by the cardinality of the intersection of the sets of developers that worked on the vertices of that edge.

The authors combine these strategies through a pondered sum, which allows them to define a higher importance to one or more strategy. However, they use absolute values, so the lower and higher bound of the possible values for each strategy may be very different. This would require a more manual process of fine tuning each weight, whereas we opted to normalize our similarity measures which makes applying weights and summing measures trivial. They also evaluated their approach on decompositions made always with four clusters and with the same weights for each similarity measure. In our approach, we

evaluate decompositions between three and ten clusters (depending on the codebase), and vary the weights of all measures between 0 and 100, in steps of 10. Different evaluation metrics are used, and we opted to use the *team size reduction ratio* metric in our own work, as it helps us answer our research question.

The combined usage of static, semantic, and history information to decompose a monolith into microservices was explored in [12]. The authors start by performing static code analysis by iterating over all classes and methods, and registering instantiations and method calls. Both of these result in an edge in a graph, where each class is a node. Three similarity measures are used:

1. *Response for a class* (RFC_α) describes a ratio of the utilization between two classes through method calls. The higher the ratio, the more similar the classes are;

2. *Semantic coupling*. This measure and computation is very similar to the one in [13], with a slight difference of adding *Latent semantic indexing (LSI)* to the NLP techniques;

3. *Evolutionary coupling*. With the same reasoning as the logical coupling from [13], they parse all logs and identify, for each pair of classes (c_1, c_2), the percentage of their overlap in the software development cycle - that is, the percentage of all commits that changed c_1 that also changed c_2.

The semantic and evolutionary measures are combined with the response for a class measure to define the edges' weights. Since both the semantic and evolutionary measures could theoretically be applied to pairs of classes that have no relationship in the code, a higher importance is attributed to the static code analysis - it is considered the source of truth for the relationships between classes. Our approach does not take this into account.

In [11], a mixed approach between static analysis and the software change history is proposed, and implemented with a tool. Similarly to [12], the authors evaluate method calls to define initial services. Each service is a tree of method calls, starting at the controller and ending at a domain entity. Based on the number of classes, methods, and commits in common, pairs of services with a similarity value greater than an initially defined threshold are merged, either into a new microservice, or into an existing one if it contains any of the two services. The combined usage of software change history and static analysis was also explored in [10]. The authors' process first extracts evolutionary coupling by detecting changes between commits while being consistent on files renames and moves - something that we also took care of handling in our solution. Evolutionary coupling exists if there are co-changes where two software artifacts change frequently. Then, it extracts static coupling by parsing Abstract Syntax Trees. Static coupling exists if there is inheritance, method calls, or aggregation between two classes. A software relation graph is built by connecting vertices (classes or interfaces) according to the existence of static and/or evolutionary coupling between them.

From this analysis, we conclude that:

- The idea that software elements that change together do so for the same reason, and so should be placed in the same microservice, is shared by all of these authors [10–13] at some level. Quantifying the number of commits in common between the elements is also a transversal trend, with satisfying results. As such, we adapt the *logical coupling* measure from [13] and use it in our work. The *contributor coupling* (from the same work) is interesting, as it captures a dimension from the repository that the remaining authors ignore. Therefore, we also adapt this measure in our work.

- All works follow the decomposition process of collecting data, building a monolith representation, defining the similarities between elements, applying a clustering algorithm, and evaluating the decomposition. We follow the same process, although we use a hierarchical clustering algorithm that is not used in the discussed works.

- The evaluation metrics are all different, and they capture different aspects of the decomposition. It's not practical to implement all of the suggested metrics in our evaluation, so we chose just the *team size reduction ratio* metric from [13] as it provides a good team perspective on our decompositions.

3.2 Comparison of Techniques

Some research has been done on the comparison of different data collection and decomposition strategies. In [19], the authors compare which of two data collection approaches, static code analysis or dynamic analysis, generate better decompositions when considering a criteria of reducing the number of distributed transactions. An evaluation was performed on two systems, and it was found that no approach outperforms the other, but the dynamic analysis required more effort. In [7], sequences of accesses to domain entities were statically collected, and four different similarity measures were defined, all driven by the goal to reduce the number of distributed transactions. The measures covered different types of accesses, like writes, reads, writes or reads, and sequences. By analysing decompositions from 121 codebases, it was found that there isn't a single measure or combination of measures that yields better results, in terms of the migration's complexity. In [13], which was already covered more in depth earlier, the authors compare their three similarity measures, based on lexical analysis, information about the contributors that changed each file and information on which files change together most often. It was found that any combination of the three measures had good results, with the contributor measure displaying more dispersion. Although the description of their approach is not as developed as the one in [11], the authors in [10] conducted an empirical evaluation on using just static analysis, just change history, and both techniques together. They found that using both techniques provided better results than just each one individually, because each of them is able to find relationships between classes that the other cannot. However, they tested their approach on just two Java codebases, and evaluated it by

comparing the generated decompositions with an expert one. We use 28 codebases and over 400.000 decompositions, and have an automated approach to the evaluation, so our results should be more generalizable.

Evaluating the effect of multiple viewpoints on the quality of a microservice decomposition was the work performed in [31]. The starting intuition would be that the more viewpoints are considered, the better, as more information is available to make a decision on how to partition the monolith. To confirm this, the author developed a Python tool that extracted static, dynamic, and semantic dependencies of seven Python projects, represented them on a graph, and applied a clustering algorithm to find highly connected and loosely coupled clusters. It was found that the static and dynamic decompositions were more loosely coupled but less functionally cohesive than semantic decompositions, and including semantic information in the analysis decreases the structural modularity quality of the decomposition. Combining static and dynamic information produced better decompositions rather than using just static or just dynamic information. This shows that there is not, necessarily, a consistent increase or decrease in the considered metrics when multiple views of the system are incorporated. In this work, we perform similar comparisons between a monolith representation based on the sequences of accesses to domain entities (obtained via static code analysis) and a representation based on file changes and file authors. However, we consider not just the modularity but also the transactional contexts and the reduction of team sizes, and we compare more codebases and decompositions.

3.3 Research Gap

There is still a lack of understanding on how different approaches compare in terms of the results they produce, and the existing comparisons generally use a small number of codebases and/or decompositions. Our work focuses on comparing the access sequence and the development history based representations, using four similarity measures based on sequences of accesses from [7], and two measures similar to the logical coupling and contributor coupling from [13]. The results are evaluated using quality metrics for modularity, complexity and team size, on a large number of codebases and decompositions.

3.4 Usage of commit logs for software engineering research

Investigating how the data from commit logs is used for different goals than our own can provide insights into improving our solution.

In [32] it is proposed an approach that, when developers are editing a file, recommends other files that may also need edits. This is done by analyzing which files have frequently changed together in the past (using a version control system), and applying a frequent pattern mining algorithm to them.

For a better analysis, they consider not only files that were committed together, but also files that were committed by the same author in a short period of time, as they most likely refer to the same task. Finally, they also filter commits with too many files changed, as these commits usually don't refer to a single task and so aren't relevant for recommendations.

A frequent pattern mining algorithm is also used in [33]. The knowledge of files frequently changed together (called "change sets") is used to uncover "traceability between source code and other artifacts", like documentation. They also use heuristics like changes in a small period of time ("**time-interval**"), changes by the same author ("**committer**"), and a mix of both ("**time-interval + committer**") to group change sets that probably refer to the same task/change.

These two works apply the notion that "files that change together should stay together". However, they go further than that by using heuristics to filter commits, like considering commits in a short period of time as one single commit and discarding commits with too many files. The goal is to capture human behaviour in their analysis and consider relationships that are not explicit. Despite the lack of a comparison of their results with and without the heuristics, we opted to also implement them in our solution.

4

Implementation

Contents

4.1 Overview 25
4.2 Data Collection 26
4.3 Data Cleaning 28
4.4 Coupling Computation 29
4.5 Performing a decomposition 29
4.6 Mono2Micro improvements 31

The investigation in this work required the development of a data collection prototype, and the modification of the Mono2Micro tool to support our two new similarity measures. The main focus of this chapter will be on how the prototype was implemented, as it is what influences the decompositions the most.

We opted to perform the data collection using Python due to its suitability as a scripting language and simplicity, and the existence of several easy to use libraries to interact with GitHub repositories. Ultimately, these libraries were prohibitively slow for our use case (and this is discussed in further sections), but we still used Python due to the remaining reasons and the familiarity of the authors with the language.

4.1 Overview

The sequences of accesses monolith representation, that we leverage to perform comparisons, is described in depth in [7]. This representation is obtained by using Spoon, a Java source code analyser. They identify controllers and domain entities in Spring-Boot monoliths, and parse method calls in each controller to obtain the types of accesses made to the domain entities. From this analysis results a JSON file containing the controllers (functionalities) as keys, and the accesses to entities that the controller can perform as values.

On the other hand, the development history based representation is obtained by parsing the output of the `git log` shell command with Python, and applying several processing techniques to ensure that data related to renamed or deleted files is accurate. The representation consists of two JSON files: one with the monolith's files as keys and the number of times each file changed with others as values, named file changes representation; and another with the files as keys and the authors that changed each file as values, named changes authorship representation. The combination of both of these representations is a development history based representation.

Then, we compute the similarity based on the monolith's representations - similarity between domain entities for the sequences of accesses and between files for development history. These are used by a hierarchical clustering algorithm to generate the decomposition.

Mono2Micro then computes the complexity, cohesion, coupling, and team size reduction ratio for a large number of decompositions, obtained by varying the weights and the number of clusters.

Figure 4.1 contains the overview of the processing pipeline necessary to create a decomposition of a monolith. It includes a detailed view of the data collection steps, and a higher-level overview of the decomposition steps. Note that for the generation of the similarity matrix (step 4.2), the data from the sequences of accesses is also used. This data is obtained from a separate data collection process.

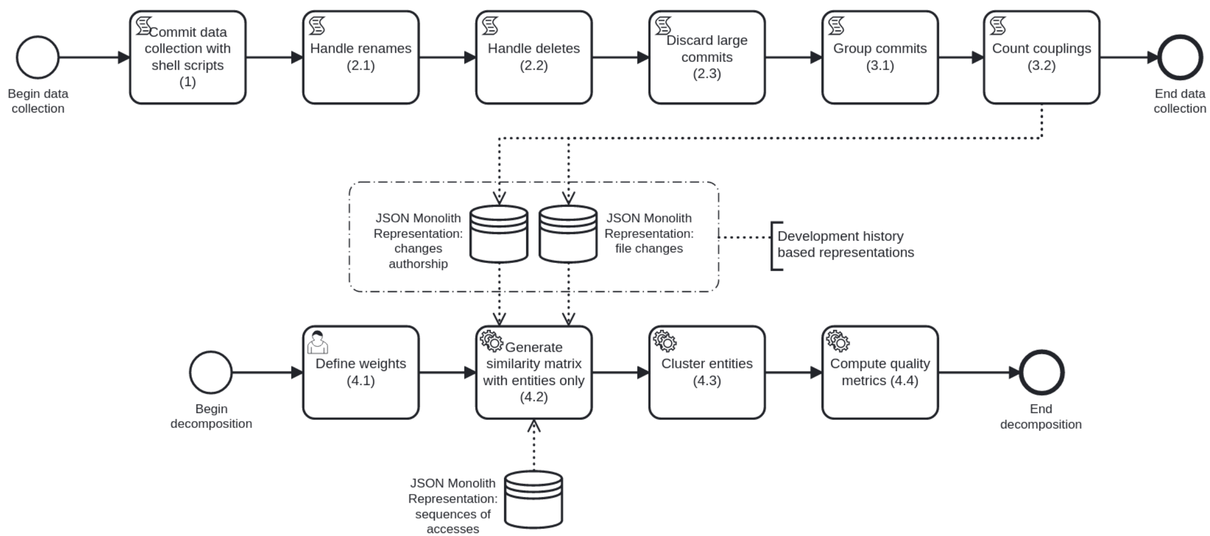

Figure 4.1: Pipeline for generating a decomposition of a single monolith system, using data extracted from the development history and data from sequences of accesses. The numbering on the different scripts/processes is referenced throughout the section.

4.2 Data Collection

As mentioned before, there are useful packages that offer good abstractions to interact with a Git repository, like GitPython[1] and PyDriller [34], but they were very slow in certain tasks. For example, using PyDriller to extract the files that changed in all commits of a large codebase (with over 25k commits) could take up to 3 hours on a laptop equipped with an SSD and an Intel Core i5 7200U. As this is a core task of our investigation, this performance made development incredibly difficult. Luckily, the fix is simple: we use Python to run and capture the output of a shell script. This brought down the execution time of that task applied to a large codebase to around 10 seconds: a 99.99% reduction.

The collection starts with a *git log* shell command:

```
1 git log --reverse --name-status --find-renames
2     --pretty=format:"commit %H %ct %ce"
```

This tells Git to return all commits in chronological order, by displaying the names and statuses of changed files, detecting and reporting renames of files, and displaying the literal string "commit", followed by the commit hash (`%H`), the commit time formatted as the UNIX timestamp (`%ct`), and the commit author's e-mail (`%ce`). Figure 4.2 contains a partial output of the script applied to the *Quizzes Tutor*[2] codebase. On the first line, you can see the string "commit", followed by the hash, the commit time, and

[1] https://github.com/gitpython-developers/GitPython
[2] https://github.com/socialsoftware/quizzes-tutor

```
commit  53d0f02fc882756c79632a2aca6c9148fc3c045f    1564418079  pedro.correia.105@gmail.com
M    frontend/src/components/QuestionComponent.vue
M    frontend/src/components/ResultComponent.vue
A    frontend/src/views/management/QuestionsView.vue
D    frontend/src/views/question/QuestionsManagement.vue
R095     frontend/src/views/QuizView.vue frontend/src/views/user/QuizView.vue
R092     frontend/src/views/ResultsView.vue   frontend/src/views/user/ResultsView.vue
R100     frontend/src/views/SetupView.vue     frontend/src/views/user/SetupView.vue
R100     frontend/src/views/StatsView.vue     frontend/src/views/user/StatsView.vue
```

Figure 4.2: Example of the output from the commit data extraction script.

the author's e-mail. This is followed by one line for each of the files changed in this particular commit. The line contains the status, which can be **A** (Added), **D** (Deleted), **M** (Modified), or **R** (Renamed), as well as the file's name. **R** is followed by a number between 50 and 100 that represents the similarity between the file in the last commit and the file in the current commit, and the line also contains the new name.

It's important to touch on the topic of lines related to renames. Git does not have a concept of file renames, because Git only tracks *changes* and not *files*. If we were to remove the `--find-renames` flag from our command, and there was no configuration telling Git to detect renames, it would report a rename as an add/delete pair. For example, in figure 4.2, the file `frontend/src/view/QuizView.vue` was renamed to `frontend/src/view/user/QuizView.vue` - it moved to a new folder. Without rename detection, we would have a line saying that the latter file was added, and the former was deleted. But we are interested in evaluating how files relate to each other - and indeed, the new file is 95% similar to the previous one (seen by the `R095` status on the left), so they are pretty much the same file with a slight change (in this case, in imports). For the purposes of our analysis, this is the same file, and relationships with the previous name should be assumed to also apply to the new name. To ensure that as many renames as possible are detected, we change Git's configuration right before the script runs with the following line: `git config diff.renameLimit 999999` . The rename limit is then reverted at the end of the script's execution with `git config --unset diff.renameLimit` .

The output of this command is stored in a temporary file, which is used as input to the following `awk` script:

```
1 $1 == "commit" {commit = $2; time = $3; author = $4}
2 $1 == "A" {printf "%s;ADDED; ;%s;%s;%s\n", commit, $2, time, author}
3 $1 == "M" {printf "%s;MODIFIED; ;%s;%s;%s\n", commit, $2, time, author }
4 $1 == "D" {printf "%s;DELETED; ;%s;%s;%s\n", commit, $2, time, author }
5 match($1,"R[0-9][0-9][0-9]") {printf
6         "%s;RENAMED;%s;%s;%s;%s\n", commit, $2, $3, time, author}
```

	commit_hash	change_type	previous_filename	filename	timestamp	author
1	60a8f2def6390e6...	ADDED		backend/sr...	1553771849	pedro.cor...
23	f941f9a59df0f68f...	MODIFIED		backend/sr...	1556909573	pedro.cor...
24	f941f9a59df0f68f...	RENAMED	spring-api/src/main/...	backend/sr...	1556909573	pedro.cor...
25	f941f9a59df0f68f...	ADDED		backend/sr...	1556909573	pedro.cor...
26	f941f9a59df0f68f...	ADDED		backend/sr...	1556909573	pedro.cor...

Figure 4.3: The head of the history dataframe of the quizzes-tutor codebase.

Each commit's log information is processed by this script, which converts it to a csv-like format. This way, we can build a script in Python to read the data into a dataframe for further processing. A dataframe is a data structure that can be seen as a very efficient table, with rows and columns, and is provided by the Pandas package. We use it extensively in our implementation, as we manage to hit up to 10x faster data processing speeds in further steps compared to native Python lists and dictionaries.

A final *grep* command filters out any file that does not end in `.java`. This is easily configured, making the data collection part independent of the language used in the monolith.

4.3 Data Cleaning

At this point, the data from the logs is not yet ready to be used. There are two situations regarding files getting renamed and files getting deleted that require special care.

If a file is renamed from `A.java` to `B.java`, and no care is taken, the final decomposition would contain both files because there have been changes to both files (before and after the rename) in the history. This is incorrect because `A.java` no longer exists. But since it is the same file as `B.java`, we replace all instances of `A.java` with `B.java` (step 2.1). A similar strategy is followed by Mazlami in [13].

Files may be deleted at timestamp X but then appear as added or modified in timestamp $X + Y$, either by getting merged from another branch or by being re-added. If this happens, we don't want to delete those files: there is relevant information after their supposed deletion, and they still exist in the current snapshot of the repo. So, we delete any occurences of files that show up at least once with a `DELETED` status, but *only if* they don't show up at a later timestamp with a different status (step 2.2).

Finally, just like in [32], we discard any commits with more than 100 modified files for the purposes of data analysis - but *not* for the purpose of identifying renames and deletes. The reasoning is that these types of commits are usually associated with refactor operations, where a large batch of files is renamed, deleted, or otherwise changed. But if we were to completely ignore these commits when identifying renames and deletes, we would be throwing out key moments in history that help us in this task.

At the end of this process, we have a dataframe containing one row for each file change. The first few rows of the quizzes-tutor codebase can be found in figure 4.3. For every file that was Added, Modified, or

Renamed, there's a row in the dataframe containing the commit hash where it appeared, the type of the change, the previous filename (if the change is a Rename), the filename, the timestamp of the change and the change's author. With this, we have every information we need to compute the similarities between files/entities.

4.4 Coupling Computation

The goal is to convert the history dataframe obtained before into a JSON structure that states which files have been modified together, and another JSON that states which authors have modified a given file. These are the monolith representations used afterwards to generate a decomposition.

Grouping commits is the first step towards this goal. Different works mention bundling together commits performed by the same author in a short timeframe [13,32,33]. The reasoning for this is that it's likely that all those commits relate to the same task, so they should be considered as just one commit. We opted to also follow this reasoning, and consider all sequential commits made by the same author in the period of one hour as just one commit. Then, for each commit, we generate permutations of size 2 of all files that were modified in that commit. For each of these pairs, a JSON structure is updated with new coupling counts, and another JSON structure is updated with authors.

Figure 4.4 illustrates how this structure is updated and how it looks like in a simple situation with two commits. For each file found in the file changes representation, we can very simply query how many times it changed with other files, as well as the number of commits it appeared in, which is all the data we need to compute the commit similarity measure during the decomposition phase.

4.5 Performing a decomposition

Decomposing a monolith into microservices is done with an automated tool and follows a similar procedure as to what is described in the literature, applying the similarity measures in equations (2.1) to (2.4). In what concerns the similarity measures in equations (2.5) and (2.6), only the files corresponding to domain entities are automatically selected from the monolith's data representations.

The decomposition process is to first build a similarity matrix, and then apply a clustering algorithm to it. The similarity matrix contains one line and one column for each of the domain entities we want to consider. Each entry in the matrix contains a number that represents how similar the two entities are, and combines both the data from the file changes representation and the authorship representation according to a certain weight. We can choose a value between 0 and 100 for the weight, as long as the sum of weights equals 100.

For example, following the earlier example, `A.java` has changed two times with `B.java`, and these

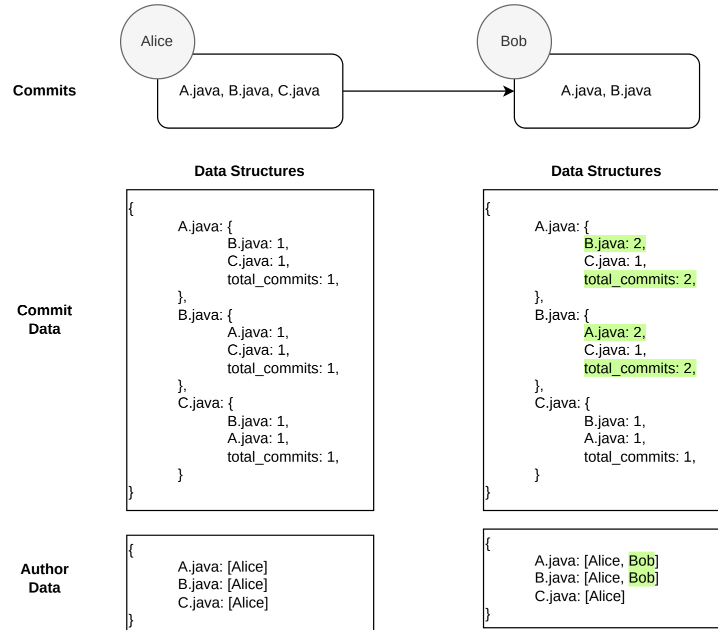

Figure 4.4: How the structures containing the commit and author data look like, and how they change when a new commit is parsed. The change from the first to the second commit is highlighted in green.

files have two authors in common. Both were also modified in two commits, and by two authors in total. This means their similarity in both cases is 1. As such, if we attribute equal weights (0.5) to both representations, the entry (`A.java`, `B.java`) has the value $1 \times 0.5 + 1 \times 0.5 = 1$ - they are fully similar. A similar logic applied to the rest of the pairs results in matrix 4.1.

$$
\begin{array}{c} \\ \texttt{A.java} \\ \texttt{B.java} \\ \texttt{C.java} \end{array}
\begin{array}{c} \begin{array}{ccc} \texttt{A.java} & \texttt{B.java} & \texttt{C.java} \end{array} \\ \begin{bmatrix} 1 & 1 & 0.5 \\ 1 & 1 & 0.5 \\ 1 & 1 & 1 \end{bmatrix} \end{array} \tag{4.1}
$$

We then apply a hierarchical clustering algorithm to this matrix using the SciPy Python library [35], and perform a cut with a certain number N. This results in a decomposition of the monolith's domain entities into N microservices, each one containing at least one entity. Pairs of entities that have higher values in the matrix will generally be placed in the same microservice.

With this strategy, it's easy to perform decompositions that also take into account data from the static analysis. Recall from chapter 2 that 4 different measures from the static analysis exist. Since both our measures and these static measures vary between 0 and 1, we can perform a pondered sum of their values, and the final similarity of any two entities will also vary between 0 and 1. The ability to do this serves as the basis for the evaluation and comparisons made in chapter 5.

4.6 Mono2Micro improvements

Several changes were implemented in Mono2Micro that greatly improved its performance.

The first change has to do with parallelization at the complexity computation level. This computation requires iterating over all functionalities and local transactions multiple times. However, it is not necessary to know the complexity of functionality #N to compute the complexity of functionality #N + 1. As such, whenever an iteration over all functionalities or local transactions was performed, it was replaced by Java's parallel streams. To sum the values, synchronization on a single cumulative variable was added.

The analyser is a part of Mono2Micro responsible for (1) the generation of all decompositions of a codebase, and (2) the computation of the quality metrics for each decomposition. (1) is implemented in Python, and was parallelized due to similar reasons as above: each decomposition is independent of other decompositions, so several decompositions can be generated at the same time. This was achieved with the `multiprocessing` library. (2) is also highly parellelizable, as each cluster is independent. We first convert each decomposition into an object, and then added parallel streams when iterating over all generated decompositions. We then maintain a synchronized structure to store the metrics of each decomposition. An important finding was that many decompositions were exactly the same even with different weights. Therefore, before computing the metrics of a new decomposition, we check if it is equal to a previously computed one (same number of clusters, and the same entities in the same clusters). If it is, we fetch its quality metrics' values and set them as the quality metrics' values of the new decomposition. In essence, we built a cache of decompositions, which achieves hit rates between 67% and 99%, depending on the number of entities of the codebase.

5

Evaluation

Contents

5.1 Codebase selection and characterization . . . 36
5.2 Results . . . 38
5.3 Evaluation conclusions . . . 54
5.4 Threats to validity . . . 55

The research goal of this thesis, as stated in the introduction, is the following:

How do monolith microservices identification approaches that use the monolith development history based representations perform when compared with approaches that use the monolith functionalities sequences of accesses representation?

To address this research question, we generate a large set of decompositions and analyse how the decomposition qualities vary depending on different monolith representations and similarity measures.

Whenever we create a decomposition, we choose the weights to attribute to our similarity measures. If we choose 0 as the weight for the commit and the author similarity, then the decomposition is created using only data from the sequences of accesses representation. On the other hand, if we choose 0 as the weight for all four sequences of accesses measures (access, read, write, sequence), then the decomposition is created using only data from the development history based representations. In the remaining combinations, the decompositions are created with data from both sources. By filtering the results of all generated decompositions, we can obtain five distinct groups of decompositions based on the weights used to create them: only data from the file changes representation; only data from the changes authorship representation; only data from the sequences of accesses representation; only data from the file changes representation and changes authorship; data from all representations. This allows us to then compare the groups' quality metrics and draw conclusions.

The comparisons will first be made by evaluating the median and the dispersion of each quality metric in each of these five groups, which gives us an overview of how, on average, each representation behaves. We also evaluate the median values of the metrics in the case of the best decompositions - that is, the decomposition of each codebase with the best value for each metric. This gives us a different perspective by focusing on which group performs best when the ideal weights for the similarity measures are found. Finally, we assess if our findings hold when we compare codebases with more commits and authors than the mean with codebases with less commits and authors than the mean. This is done by considering the best decompositions, as it gives greater strength to our findings.

The comparisons are, initially, done visually through the analysis of boxplots. Whenever it's not visually obvious that there is a large difference between groups, and considering that there are different amounts of decompositions in each group and the quality metric values don't follow a normal distribution, we use a Welch T-test with the following hypotheses:

- H_0 - There are no significant differences between the mean ${QUALITY METRIC} of ${GROUP} decompositions and ${OTHER GROUP}.
- H_1 - The mean ${QUALITY METRIC} of ${GROUP} decompositions is greater than ${OTHER GROUP}.

5.1 Codebase selection and characterization

The sequences of accesses monolith representation we are comparing was developed in [7]. In that work, a total of 121 codebases using the Hibernate ORM were selected, according to the following procedure:

1. Get all GitHub repositories that list the Spring Data JPA library as a dependency;
2. Filter out repositories that did not contain at least 5 files whose name ended in `Controller.java`, and at least 5 files whose name did not contain `Dao` or `Repository`.
3. The remaining repositories were ordered by the number of GitHub stars, and 118 codebases were manually selected. Repositories from lessons or tutorials, and repositories that did not use just Spring Data JPA were disregarded.

The authors from [7] made available a .zip file with the source code of all codebases, as well as the collected sequences of accesses. This data was gathered in 2020, and most of the codebases have maintained an active development cycle since then. As such, the collected data might not be entirely accurate to the present day version of the codebase, so we cannot just clone the codebase at the latest commit and collect data related to the development history. Therefore, for the sake of our comparisons being as accurate as possible, we filter the codebases made available according to the following criteria:

1. We can only know the latest commit of each codebase if a `.git` subdirectory is available. Therefore, we filter out all folders that do not contain this directory.
2. As we are looking for an active development history, we filter out all codebases with less than 100 commits and less than 2 authors. The shell command `git rev-list --count HEAD` lists the number of commits reachable from the `HEAD` commit, and corresponds to the number of commits displayed in the GitHub page of the repository. The number of authors is obtained with the command `git log --pretty='%ae' | sort | uniq | wc -l`, which lists all author's emails from all commits, sorts them, removes duplicated ones, and then returns just the number of authors.
3. We manually discarded a few codebases that did not follow assumed conventions, like the file of an entity having the same name as the entity. For example, a `BookDepository` class in a `Bookdepository.java`.

After this filtering, we ended up keeping a total of 28 codebases. Figure 5.1 displays the number of functionalities as a function of domain entities, as well as the number of authors as a function of commits, and the number of entities as a function of commits. Although the number of functionalities tends to increase as more entities exist, there isn't such a clear relationship between the number of authors and the commits of a codebase, as well as the number of commits and the number of entities.

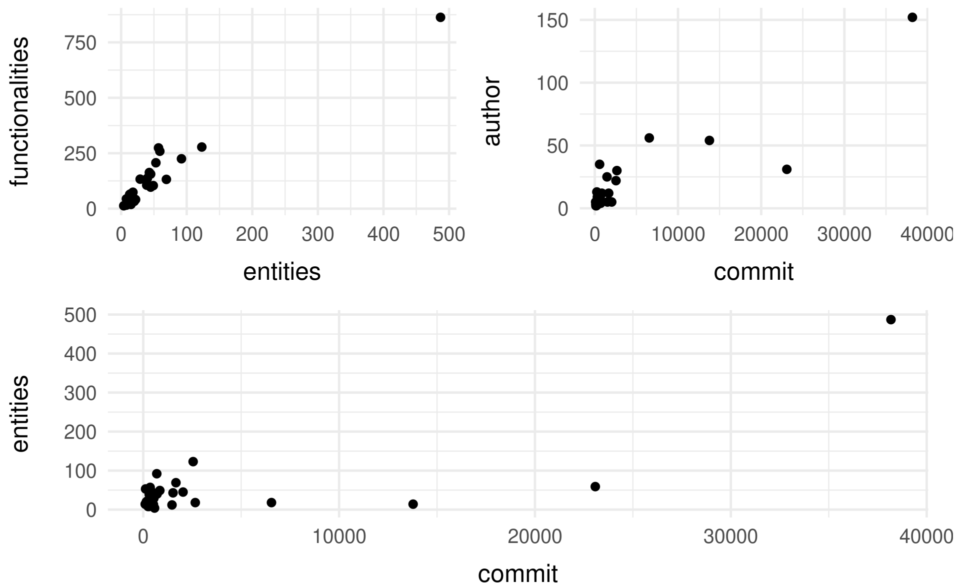

Figure 5.1: Plots showcasing some relationships of codebases' features: commit, authors, entities, and functionalities.

Overall, the average number of commits is ~ 3578, with a standard deviation of ~ 8373. Considering this, we can say that we have a somewhat diverse distribution of number of commits, which helps our conclusions be more generally applicable. For future reproducibility of our work, table 5.1 contains the name of the codebases used, their repository url, and the hash of the latest commit we are considering.

Table 5.1: Information about the considered codebases.

Codebase name	URL (github.com)	HEAD commit hash
Acme-Champions	Agusnez/D05-Acme-Champions	1f903258
APMHome	devhotmail/APMHome	fe087f78
Axon-trader	AxonFramework/Axon-trader	1e987bb1
blended-workflow	socialsoftware/blended-workflow	53082487
cloudstreetmarket.com	alex-bretet/cloudstreetmarket.com	76de2e0d
cloudunit	Treeptik/cloudunit	5079ab85
echo	cardinal76/echo	ac40d97e
edition	socialsoftware/edition	1b2a2b0c
ExtremeWorld	pengchao1989/ExtremeWorld	34d34542
FengHuang	TimYi/FengHuang	17acbe8a
fenixedu-academic	FenixEdu/fenixedu-academic	47eb18f8
hexie	linknabor/hexie	c1d9a85a
irida	phac-nml/irida	b1ea3274
jewelry	masanchezr/jewelry	9dbe6cb8
keta-custom	ketayao/keta-custom	ef8bc0b9
luna	huacha/luna	ab42988e
market-manage	JoleneOL/market-manage	0f3baa4f
quizzes-tutor	socialsoftware/quizzes-tutor	6dcf6684
reddit-app	Baeldung/reddit-app	c7af951a
ruanfan	phyche/ruanfan	ba6e9f45
Skoolie	Kaydub00/Skoolie	8336599f
SoloMusic	jualopmun/SoloMusic	2cf3b54d
splunkwithaws	vinothkrishna15/splunkwithaws	52759335
spring-cloud-gray	SpringCloud/spring-cloud-gray	17cdbd1f
spring-framework-petclinic	spring-petclinic/spring-framework-petclinic	50a219c7
TwitterAutomationWebApp	brianmarey/TwitterAutomationWebApp	dd739539
webofneeds	researchstudio-sat/webofneeds	8fa92d0c
xs2a	adorsys/xs2a	d0642091

5.2 Results

For each codebase, we create decompositions with 3 to 10 clusters, according to the number of entities it contains: $3 \leq n_entities < 10 =$ 3 clusters; $10 < n_entities < 20 =$ 3, 4, and 5 clusters; $n_entities \geq 20 =$ 3 to 10 clusters. For each number of clusters, we generate decompositions with varying weights on the six measures: from 0 to 100, with increments of 10. The distribution of the number of generated decompositions can be found in Table 5.2.

Table 5.2: The number of generated decompositions across all codebases.

#Entities	#Clusters	#Codebases	#Decompositions
3 to 9	3	4	11982
10 to 19	3 to 5	8	72072
20+	3 to 10	16	384384
Total		**28**	**468438**

5.2.1 Uniform complexity

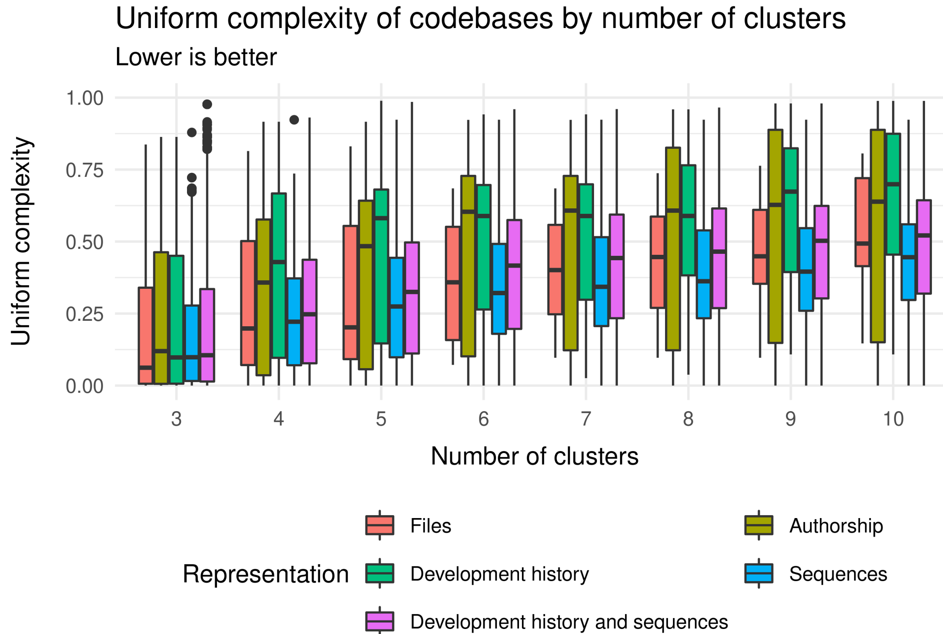

Figure 5.2: Median uniform complexity of the codebases, per number of clusters and per representation.

We start the analysis of the uniform complexity by comparing the values of all codebases per cluster.

From a visual inspection of Figure 5.2, it looks like the changes authorship and the file changes representations generate decompositions with higher complexity than the sequence of accesses representation, in all clusters. To confirm this, we apply a Welch T-Test with the following hypotheses:

- H_0 - There are no significant differences between the median uniform complexity of file changes decompositions and sequences of accesses.
- H_1 - The median uniform complexity of file changes decompositions is greater than sequences of accesses.

We only obtain a p-value < 0.05 in the case of 10 clusters, so we can only state with confidence that the median uniform complexity of the file changes decompositions is greater in the case of 10 clusters. Performing a similar test with the authorship representation instead of the file changes representation leads us to a similar conclusion for the case of 4 and 5 clusters (p-values of 0.039 and 0.023, respectively).

The development history representation presents a statistically significant higher median complexity than the sequences of accesses and the file changes, in all clusters. The combined development history and sequences is also statistically significantly higher than the sequences of accesses. All of

this indicates that the usage of any development history based representation, generally, generates decompositions with worse complexity than those using the sequences of accesses.

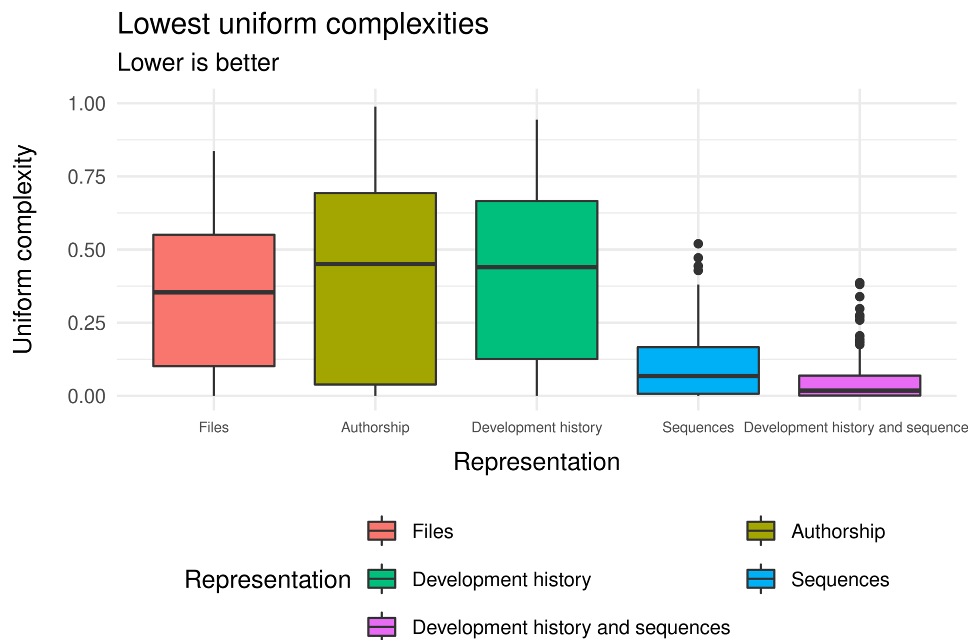

Figure 5.3: Median uniform complexity of each representation's best uniform complexity decompositions.

The previous analysis showed that, on average, the sequences of accesses representation is better in terms of complexity. However, when selecting the best decomposition of each codebase for each number of clusters, we find that 92.95% of decompositions contain data from a development history representation. This indicates that a detailed analysis on the best decompositions should be made to verify if the same results are obtained.

Figure 5.3 displays the median uniform complexity of the best decompositions (in terms of complexity) of each representation. What we find is that in this case, the development history and sequences representation is the best, as all other representations have statistically significant greater median. In the case of the development history and sequences representation, 85.9% of the decompositions have a weight in the author measure smaller than 25%. This means that data from the changes authorship representation is generally not given much importance in the best decompositions of the development history and sequences representation. Considering that we cannot state that the development history representation by itself displays higher median than the changes authorship representation, but it does display higher median than the file changes representation, the presence of authorship data seems to often generate decompositions with worse uniform complexity, even in the case of the best decompositions.

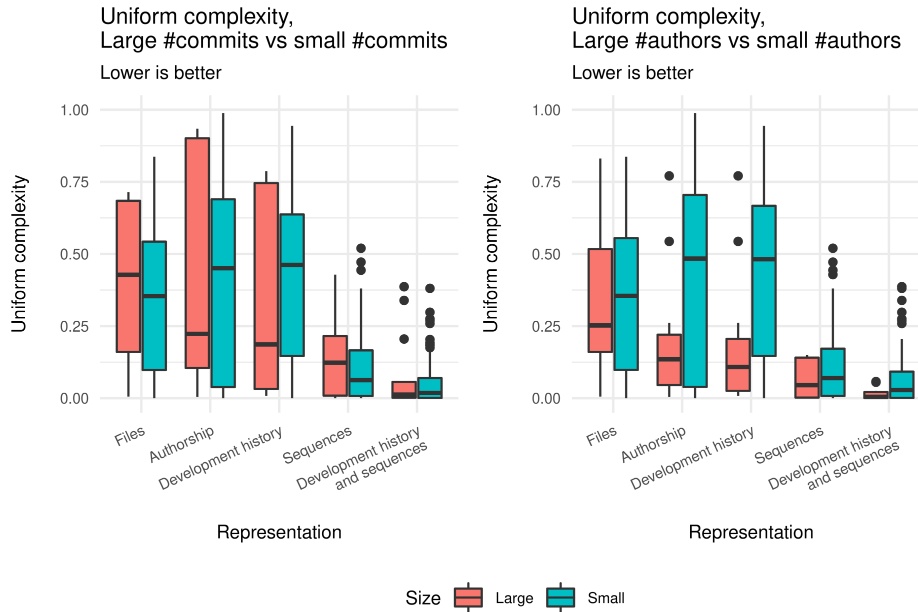

Figure 5.4: Comparison between the uniform complexity of codebases with a large number of commits with codebases with a small number of commits, and codebases with a large number of authors with a small number of authors. Only the best decompositions were considered.

From the analysis of the best decompositions, we found that combining the sequence of accesses representation with the development history based representation yielded the best results. We also found that the presence of authorship data tended to be detrimental to the complexity value. We considered all codebases, but some codebases have a much larger number of commits and/or authors than the mean. Do the same conclusions as before follow when comparing these large codebases with codebases with less commits and/or authors than the mean?

Figure 5.4 displays the median uniform complexity of these two groups of codebases. We can say with statistical significance that the median uniform complexity of small codebases in the number of authors is greater than that of large codebases in the number of authors, when comparing the authorship representation for both cases (p-value: 0). We can also state that there are no significant differences between the median uniform complexity of the authorship representation in large codebases to the median uniform complexity of the sequences representation in small codebases, whereas there is a difference if we compare these two representations in small codebases. This is a surprising result, as it suggests the effectiveness of the authorship representation, when compared with sequences, if more authors are present.

In the case of large and small codebases in the number of commits, we cannot state with confidence that there are significant differences between both types of codebases. We find very large quartiles in the boxplots of the development history based decompositions, which means that the uniform complexity

obtained is very dependent on the codebase itself.

To summarize: on average, the sequences of accesses representation is better in terms of complexity than any development history based representation; when considering the best decompositions, combining the sequence of accesses representation with the development history representation yields better results, and authorship data worsens the results; codebases with more authors than the mean display better complexity than codebases with less authors than the mean, in the authorship representation, and this representation in large codebases is comparable to the sequences of accesses in smaller codebases.

5.2.2 Cohesion

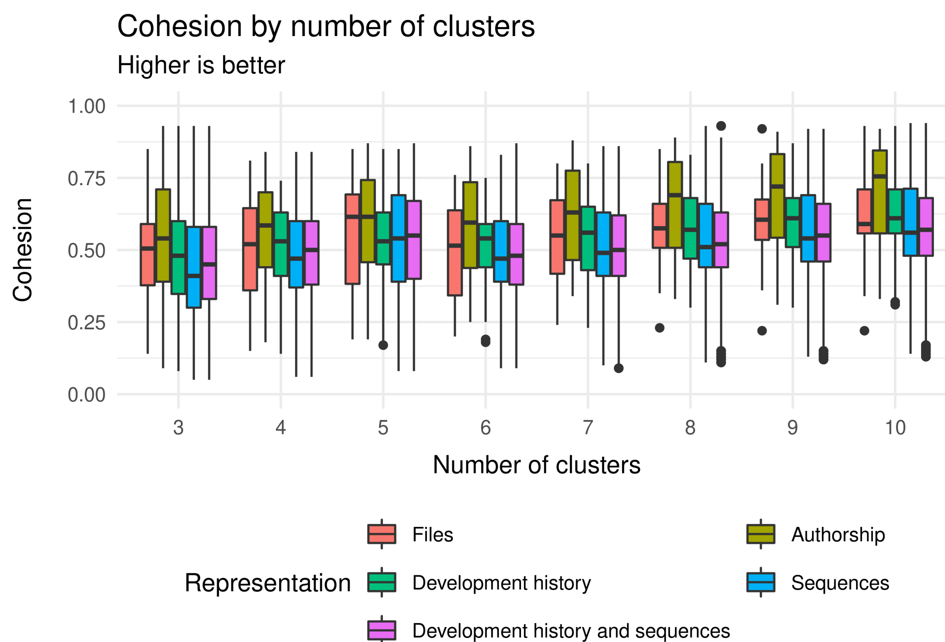

Figure 5.5: Median cohesion of the codebases, per number of clusters and per representation.

In terms of cohesion, we can visually see in Figure 5.5 that the sequences representation is very similar to the combined development history and sequences. We can confirm this via a Welch T-test with the following hypotheses:

- H_0 - There are no significant differences between the median cohesion of sequences of accesses decompositions and the development history and sequences.
- H_1 - The median complexity of the development history and sequences decompositions is greater than the sequences of accesses.

We obtain a p-value < 0.05 in the case of 3 and 4 clusters only, so we can only reject H_0 in these two instances. This means that in the majority of clusters, we cannot say that the development history and sequences has better (higher) cohesion than the sequences. We do find, however, that the changes authorship representation has statistically significant higher cohesion than the sequences and the development history and sequences representations, for all clusters except 5. Finally, we cannot confidently state that the file changes has greater median than any other representation, suggesting that it is comparable to them in this metric.

Figure 5.6: Median cohesion of each representation's best cohesion decompositions.

On average, the previous analysis showed that all representations (other than the changes authorship) are more or less comparable. Following the reasoning from the uniform complexity analysis, we looked at the best decompositions of each codebase and number of clusters, and found that 80.77% have some data from development history based representations. Therefore, it also makes sense to investigate these decompositions in terms of cohesion. The median values for the various representations are displayed in Figure 5.6.

We find with statistical significance that the best decompositions of the development history and sequences representation have the best values in this metric. We can confidently say that the sequences of accesses is also higher than the three development history based representations. Therefore, combining both data from the development history and data from the sequences of accesses yields best results than just using one of the representations, although the median differences between representations are not as extreme as in the case of uniform complexity.

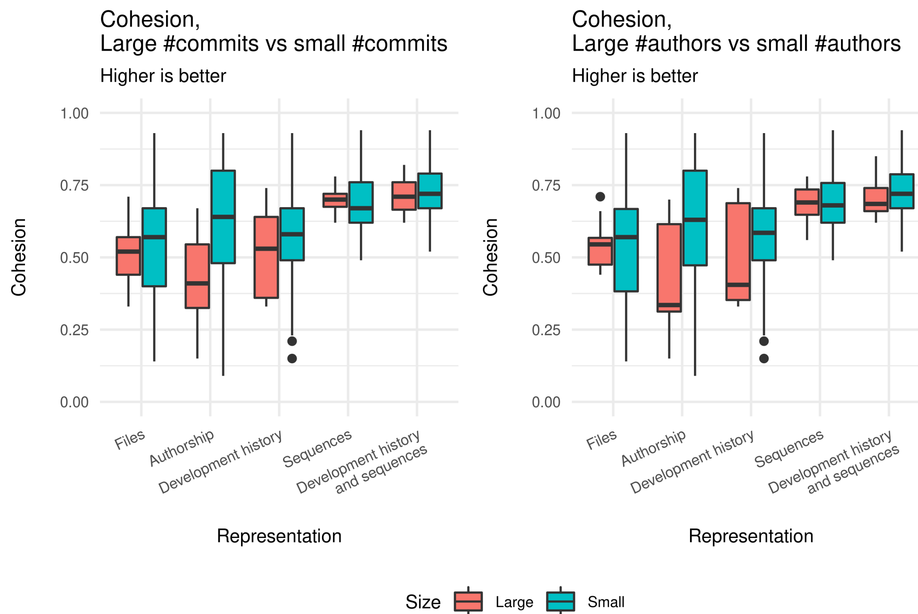

Figure 5.7: Comparison between the cohesion of codebases with a large number of commits with codebases with a small number of commits, and codebases with a large number of authors with a small number of authors. Only the best decompositions were considered.

Given the poorer performance of the file changes and the changes authorship representations in the best decompositions for all codebases, we now evaluate if there are significant differences between codebases with more commits and/or authors than the mean and codebases with less commits and/or authors than the mean. Figure 5.7 visualizes the median values of the cohesion, for these two groups, and for each representation.

We can confidently state that the median cohesion of small codebases in the number of commits, in the case of the authorship representation, is greater than the large codebases in the number of commits for the same representation. For the file changes, development history, and development history and sequences, we cannot state that smaller codebases in the number of commits have greater cohesion than the large, and for the sequences of accesses representation, we cannot state that the large codebases display higher median than the smaller.

In the case of codebases with many and few authors, we reach the exact same conclusions. Considering this, the presence of more authors or more commits does not significantly improve cohesion, but less authors seems to generate better decompositions when using the authorship representation. We also find that, both in the case of commits and authors comparisons:

- Comparing the representations of larger codebases, the development history and sequences representation has a statistically significant greater median cohesion than the development history based representations;

- The development history and sequences representation of larger codebases has statistically significant greater median than all other representations of smaller codebases, except the development history and sequences.

This leads us to conclude that combining development history data of large codebases with the sequences of accesses leads to better or comparable results to combining development history data of smaller codebases with sequences of accesses. Just like in the case of uniform complexity, we also see this representation as the best overall.

To summarize: on average, all representations (other than the changes authorship) are more or less comparable; the combined sequence of accesses and development history representation is better than all other representations in the case of the best decompositions; this representation is better in large codebases than in smaller codebases; large codebases, both in number of commits and number of authors, are not better than smaller codebases: smaller codebases in the number of authors display better cohesion in the authorship representation.

5.2.3 Coupling

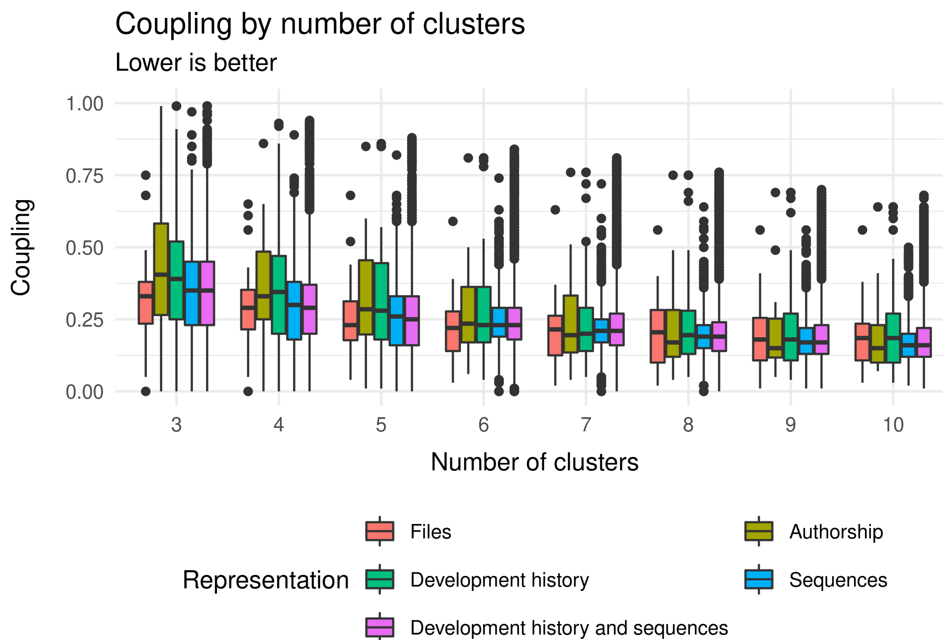

Figure 5.8: Median coupling of the codebases, per number of clusters and per representation.

Although the coupling looks similar across all representations (see Figure 5.8), with some variations for some clusters, an analysis with statistical tests reveals that:

- The development history and sequences representation median is higher than the sequences, for all clusters;
- The development history is also higher than the sequences, for all clusters, and is higher than the development history and sequences.

For the remaining combinations of representations, we cannot confidently state than any of them is higher than other consistently. As such, the sequences representation is better than any development history based decomposition for coupling.

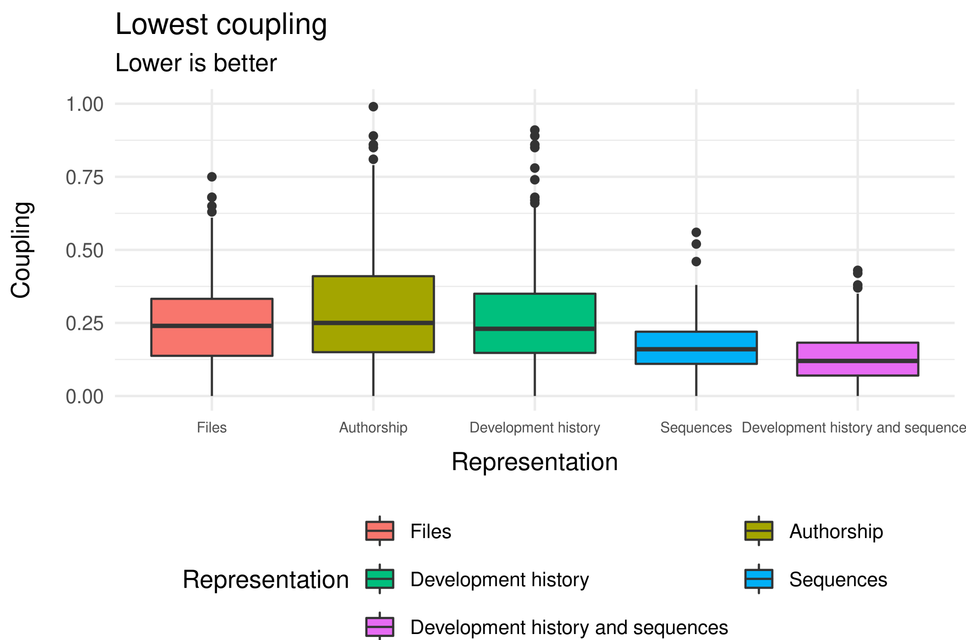

Figure 5.9: Median coupling of each representation's best coupling decompositions.

Similarly to the previous metrics, 95.51% of the best decompositions in terms of coupling are obtained using the development history and sequences of accesses representation. This warrants investigating the best decompositions of each representation, to evaluate how they compare. A visual comparison is present in Figure 5.9.

We find that just like the previous metrics, the combination of development history and the sequences of accesses generates the best values, as we always obtain a p-value of 0 when testing if any other representation has higher median. Although we can state, through statistical tests, that the sequences of accesses representation has a higher median value than the development history and sequences representation, we cannot state that it has a higher value than the remaining representations, which does mean that the inclusion of development history data makes the already good sequences of accesses representation even better.

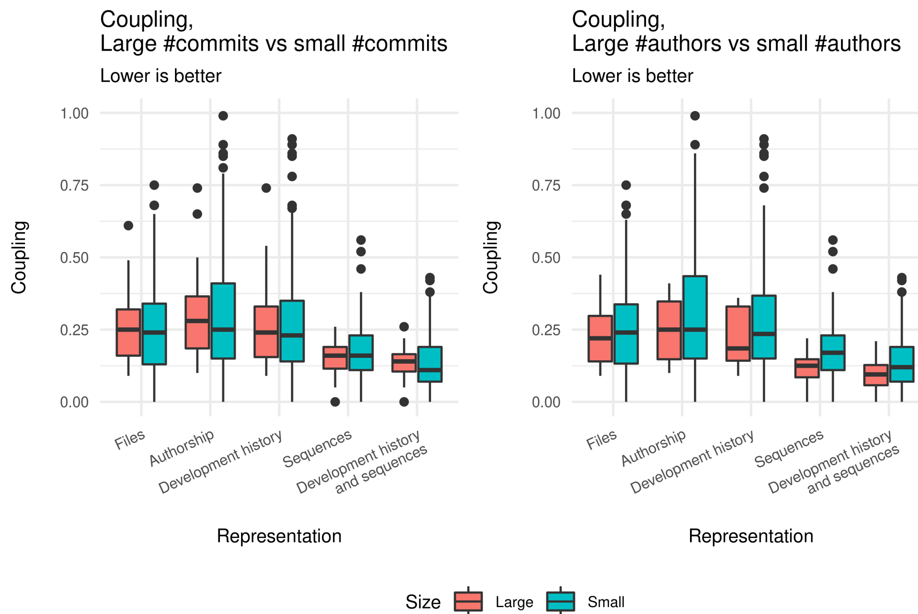

Figure 5.10: Comparison between the coupling of codebases with a large number of commits with codebases with a small number of commits, and codebases with a large number of authors with a small number of authors. Only the best decompositions were considered.

Since we were not able to state the existence of significant differences between the file changes, changes authorship, and development history representations, it is worth exploring if the presence of more or less commits and/or authors yields different conclusions.

In the case of large and small codebases in the the number of authors, we find that the obtained values in the files and authorship representations are very similar for both categories of codebases, visually (Figure 5.10) and with statistical significance, but the smaller codebases display worse values in the development history, sequences, and the development history and sequences representations. For large and small codebases in the number of commits, we cannot confidently state that the larger codebases have greater median than the smaller.

To summarize: on average, the sequences of accesses representation performs better than the remaining representations; the best decompositions made with the development history and sequences of accesses representation are better than any other representation; smaller codebases in the number of authors are worse for some representations, but no differences were found between larger and smaller codebases in the number of commits.

5.2.4 Team size reduction ratio

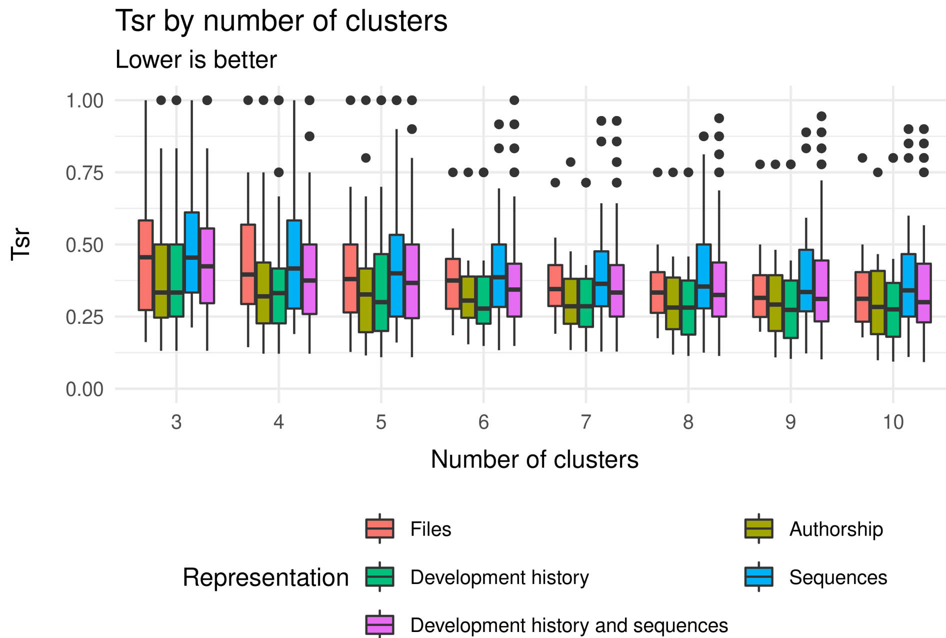

Figure 5.11: Median tsr of the codebases, per number of clusters and per representation.

In this metric, we find more consistent results across all clusters, which can be seen in Figure 5.11. Similarly to the cohesion, including data from the changes authorship representation produces better results than not including it. The changes authorship representation is significantly similar to the development history based representation, indicating that the usage of authorship data has a great influence when combined with the file changes data. As this metric evaluates the reduction in team size per microservice, when compared with the monolith's original team, it makes sense that decompositions made with author data perform better. However, it's interesting to see that the decompositions made with the sequences of accesses representation, despite having no data regarding authors, display an acceptable median value under 0.5, and the quartiles in decompositions with 6 or more clusters are all under 0.5 as well.

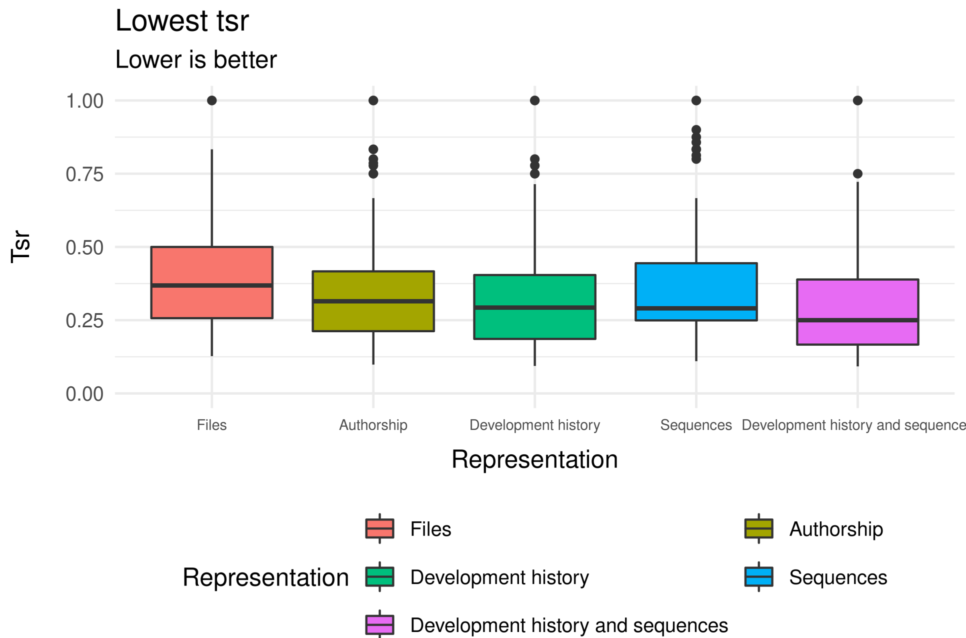

Figure 5.12: Median tsr of each representation's best tsr decompositions.

In total, 95.51% of the best decompositions in terms of tsr were achieved with the development history and sequences representation, which is an indication that, once again, we should compare the remaining representations in this context.

We cannot state that the development history representation has a higher median than the development history and sequence of accesses representation, highlighting that there is little benefit to include the sequences of accesses. Indeed, around 74% of the best development history and sequences of accesses decompositions have a sum of weights for the author and commit similarity measures greater than 50.

We can state that the authorship representation has higher median than the development history and sequences representation, but cannot state that it is higher for any other representation. We also cannot state that the file changes representation has higher median than the sequences of accesses, but it does have higher median than the remaining representations.

All in all, the development history and sequences representation is still the best, as all other representations are significantly higher. The sequences of accesses representation, despite having no author data, still displays comparable results to the authorship representation, which has only author data.

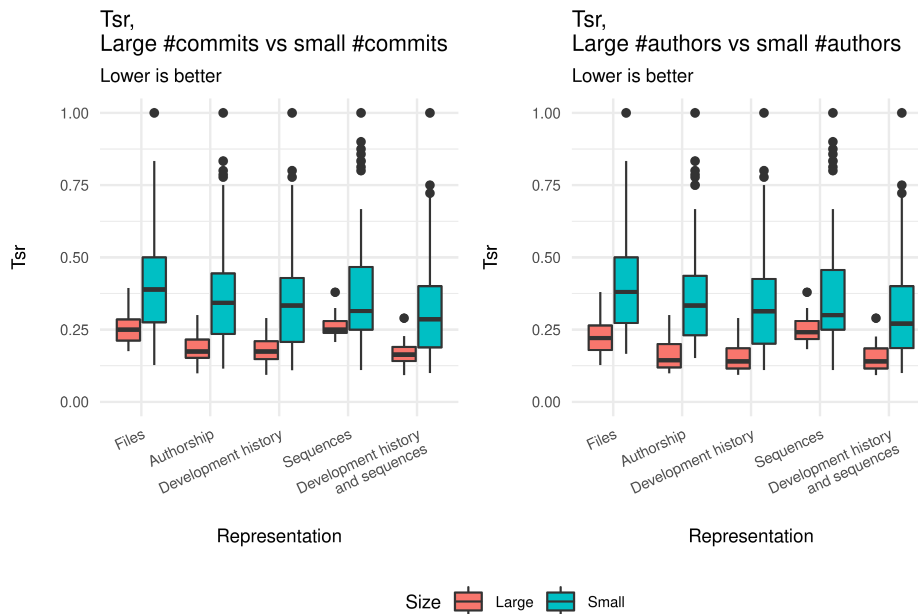

Figure 5.13: Comparison between the team size reduction of codebases with a large number of commits with codebases with a small number of commits, and codebases with a large number of authors with a small number of authors. Only the best decompositions were considered.

Considering that we cannot state the existence of differences between the changes authorship and the sequences of accesses representations, it is worth checking if the presence of more authors changes this.

Comparing large and small codebases in the number of authors, we can say that the tsr of the smaller codebases in the number of authors is higher, and therefore worse, than the large ones in all representations. Statistical tests reveal that there are no significant differences between the authorship, development history, and the combined development history and sequences representations for the small codebases, highlighting the effect that the changes authorship data has in these representations. Similarly to the uniform complexity, the development history based representations of large codebases perform better than the sequence of accesses of small codebases, whereas they perform worse in the case of smaller codebases, so the presence of more authors does change the conclusions of the previous analysis of the best decompositions.

Large codebases in the number of commits also display improved tsr median values across all representations, when compared with small codebases in the number of commits.

To summarize: on average, decompositions which include changes authorship data perform better than those that don't; in the best decompositions, the development history and sequences of accesses representation does not present siginificant differences to the development history representation, and the sequences of accesses representation is not significantly different from the authorship representa-

tion, despite having no author data; the presence of more commits and/or more authors significantly improves this metric in all representations.

5.2.5 Combined

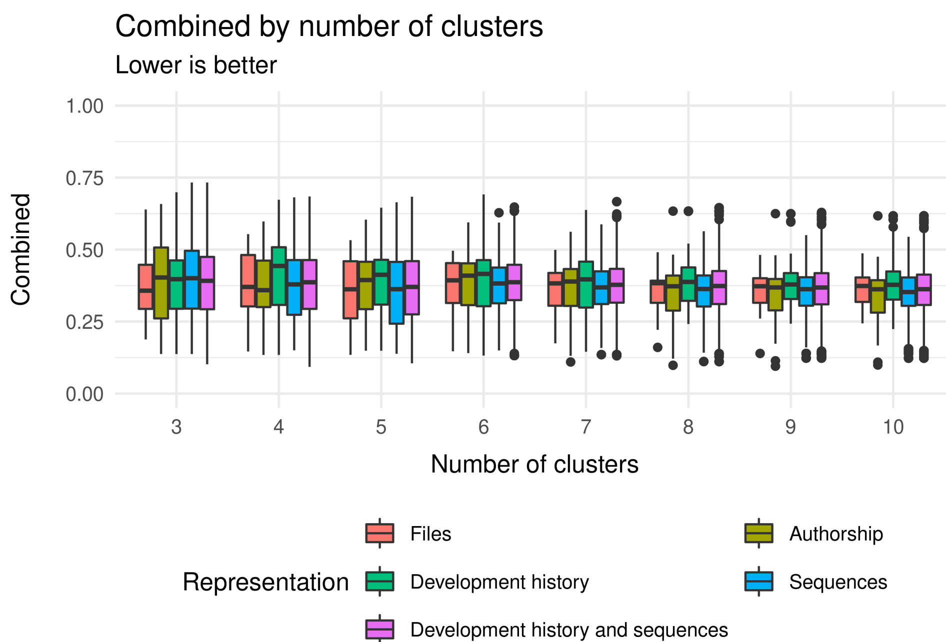

Figure 5.14: Median combined of the codebases, per number of clusters and per representation.

This metric offers an overview of all metrics and the trade-offs made in the decomposition, by combining the previous metrics into a single value. There is not a single type of representation that can be said to be the best for all clusters and metrics - from what we've covered so far, some representations excel at some metrics more than others. Therefore, it is natural that most decompositions display median values in each type of representation of around 0.4, with quartiles going up to 0.5. This can be seen in Figure 5.14.

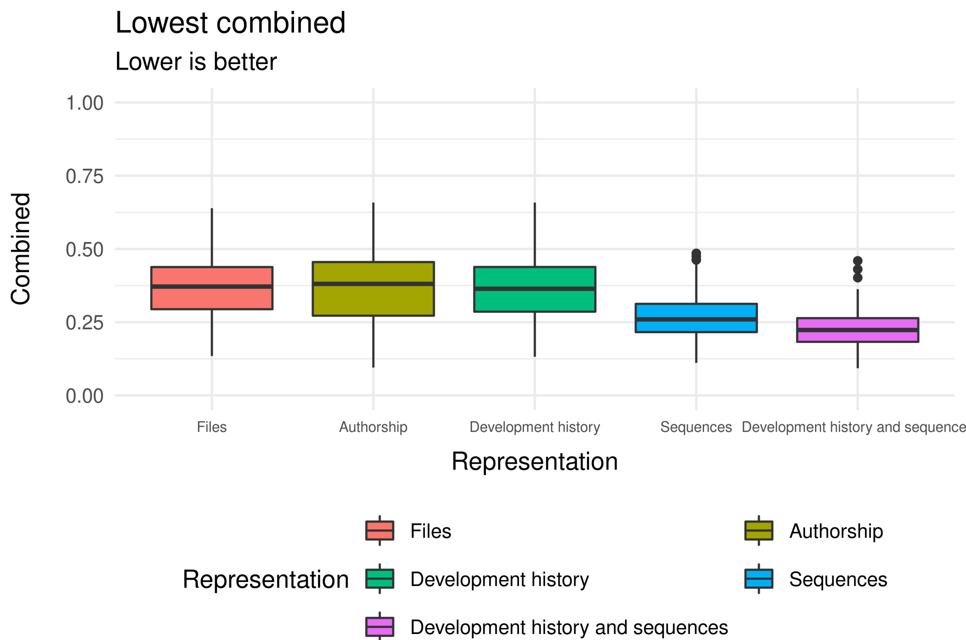

Figure 5.15: Median combined of each representation's best combined decompositions.

Since we cannot state that, on average, a single representation is better than the others, we can look at the best decompositions (in Figure 5.15) to get another perspective.

The analysis of this plot confirms the analysis of the previous metrics: the best decompositions of the development history and sequences representation are the best across all metrics, as all others representations display statistically significant higher values. The sequences representation, due to the better values in uniform complexity, cohesion, and coupling, is also good, as it has a higher median than the development history and sequences representation, but the other representations have a higher median the the sequences of accesses. Between the file changes, authorship, and development history representations, we cannot state that any of them is higher than another, which highlights the better performance of some representations in some metrics, but worse performance in others.

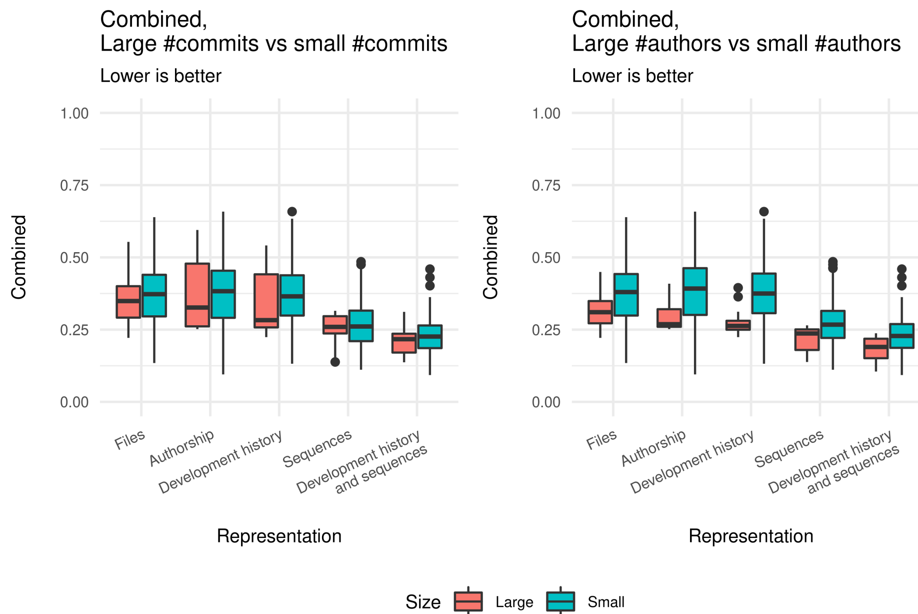

Figure 5.16: Comparison between the combined of codebases with a large number of commits with codebases with a small number of commits, and codebases with a large number of authors with a small number of authors. Only the best decompositions were considered.

Considering the similarity between the file changes, changes authorship, and development history representations, we can try to find if there are any differences when considering large and small codebases in the number of commits and/or authors. Figure 5.16 contains a visualization of this comparison.

The good performance of the larger codebases in the number of authors for previous metrics explains the improvement of the combined metric, as the representations of smaller codebases have statistically significant greater median values than the large codebases. For the case of commits, we cannot state that the smaller codebases display larger values in any representations, which means that overall, the quality of codebases with more commits is comparable to the quality of codebases with less commits.

To summarize: on average, we cannot state that any representation is better than others; the development history and sequences of accesses is the best representation when analyzing the best decompositions, which is expected considering that this representation was consistently the best in the previous metrics; large codebases in the number of authors are better than smaller, whilst no significant differences were found betweeen large codebases in the number of commits and small codebases in the number of commits.

5.3 Evaluation conclusions

We were able to properly compare the various representations in previous sections, through visual analysis and statistical analysis. The median quality metrics values across all codebases and clusters provided us with an overview of the most common scenario. But this is a limited view, as it does not fully represent what happens if an ideal decomposition is found. To fix that, we also analyzed the best decompositions of each metric and representation, where often different results were obtained. Additionally, we often could not find significant differences between the development history based representations, so we separated the best decompositions according to the number of commits and authors of their codebases, which gave us better insights in some situations.

This whole analysis provided us with several conclusions that allow us to answer the research question of this thesis, focused around how the sequences of accesses representation compares with development history based representations:

- On average, the sequence of accesses representation is better in the case of complexity and coupling. For cohesion, no significant differences were found between representations, with the exception of the changes authorship representation, which performs better than all others. Representations with change authorship data, like the changes authorship, development history, and development history and sequences, perform better than the sequence of accesses for tsr, although we have to highlight the still acceptable performance of the latter. These trade-offs are captured by the combined metric and confirmed through statistical tests, as we cannot state that any representation is better than the others at any number of clusters.

- When looking at the best decompositions, the combined development history and sequence of accesses representation yielded the best values for all metrics. A notable exception is in the tsr metric, where we could not state with statistical significance that this representation is better than the development history representation. Nevertheless, these conclusions give support to the notion that the presence of more data from different sources improves results. However, considering that this does not happen when looking at the vast majority of decompositions, it means that obtaining good results when using data from the development history is very dependent on choosing the ideal weights when creating the decomposition.

- Through the comparison between large and small codebases in the number of commits and/or authors, we cannot state that the complexity of the changes authorship representation of large codebases in the number of authors is higher than the sequences of accesses representation of small codebases in the number of authors. Regarding cohesion and coupling, we found that the presence of more commits or authors does not improve results. On the other hand, more commits

or authors does significantly improve the tsr values, as any representation of larger codebases has a better median than the sequences of accesses representation of smaller codebases.

5.4 Threats to validity

Out of all decompositions, we find that 0.37% were made exclusively with data from the development history, and 9.52% exclusively with data from the sequences of accesses. The remaining 90.11% were made with combined data. This is a consequence of using four measures related to the sequences of accesses, but only two related to the development history. To ensure this does not affect our findings, we opted to use a statistical test that performs well even comparing groups with different sample sizes, and do not rely only on boxplots to draw conclusions. Additionally, note that these differences only applied for the first analysis, where we consider to all the decompositions, which was rather inconclusive. For all the other analyses, the best decompositions were chosen and so, we have only one decomposition per representation and number of clusters.

Not all repositories have a clean and linear history, with some presenting many branches, refactors, and merges. This affects the detection and processing of deletes and renames, which makes development history based decompositions less efficient. Nevertheless, we obtained good results for the development history representations.

We found that sometimes, files presented an `ADD` or `MODIFY` change event after a `DELETE` event. In some situations, this means we could be considering two distinct files as the same one, if they happened to have the same filename and one of them was deleted before the other was added. However, in all cases we found, the files still existed in the latest repository snapshot and did correspond to the same file that was deleted. Considering that this situation is unlikely, and the existence of a `DELETE` event after an `ADD` or `MODIFY` change event usually occurs due to merges, we opted to still consider these files in our analysis, and we don't discard them.

Our data collection approach was to gather data about all `.java` files across all commits, and then discard non domain entities files only in the decomposition phase. An alternative would be to filter all commits and select those where only domain entities were changed. Our approach is richer, as we have more data available and are not deleting potentially useful relationships between files.

We adapted the logical coupling and the contributor coupling measures from [13], by considering a fraction rather than an absolute value. This was made to facilitate the integration of other measures without much experimentation on the ideal weights that would be required if absolute values were considered.

6

Conclusion

As the development of a monolith system progresses, it tends to get more complex, and introducing new features and bug fixes becomes harder. An architecture based on microservices allows for better scaling, so a migration from a monolith to this architecture brings plenty of advantages. To help with the migration, various automated approaches with different strengths, inputs, and evaluation metrics have been proposed, but they were generally tested on a reduced number of codebases and there is a lack of research on the comparison of different approaches.

In this work, we evaluated a total of 468k decompositions of 28 codebases, and compared their quality according to 5 metrics when created with different monolith representations: file changes, changes authorship, development history (which combines the previous two), sequences of accesses, and a combined development history and sequences of accesses.

On average, according to our quality metrics, development history based representations are not better than a sequences of accesses representation. With respect to the best decompositions according to each metric, the vast majority of them (over 80%) were generated with the combined development history and sequences of accesses representation. This means that even if this combination does not produce the best results on average, when compared with other representations, it is very likely that the best weights configuration for a given metric considers both representations. Interestingly, we also found that codebases with a high number of authors present better decompositions, with the authorship representation of larger codebases achieving comparable results to the sequences of accesses of smaller codebases. On the other hand, the number of commits does not have a significant impact on the quality of the generated decompositions.

Following the results obtained and what we learned, we propose for future work including more data related to repositories to evaluate similarities, like branches, GitHub Pull Requests, GitHub Issues; exploring the effect of different clustering algorithms on the decompositions; adding more codebases to further confirm our results.

Bibliography

[1] J. Lewis and M. Fowler, "Microservices: a definition of this new architectural term," *MartinFowler.com*, vol. 25, pp. 14–26, 2014.

[2] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015.

[3] N. Alshuqayran, N. Ali, and R. Evans, "A Systematic Mapping Study in Microservice Architecture," in *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. Macau, China: IEEE, Nov. 2016, pp. 44–51.

[4] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From monolith to microservices: A classification of refactoring approaches," in *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, J.-M. Bruel, M. Mazzara, and B. Meyer, Eds. Cham: Springer International Publishing, 2019, pp. 128–141.

[5] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service cutter: A systematic approach to service decomposition," in *Service-Oriented and Cloud Computing*, M. Aiello, E. B. Johnsen, S. Dustdar, and I. Georgievski, Eds. Cham: Springer International Publishing, 2016, pp. 185–200.

[6] M. Abdellatif, A. Shatnawi, H. Mili, N. Moha, G. E. Boussaidi, G. Hecht, J. Privat, and Y.-G. Guéhéneuc, "A taxonomy of service identification approaches for legacy software systems modernization," *Journal of Systems and Software*, vol. 173, p. 110868, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302582

[7] S. Santos and A. R. Silva, "Microservices identification in monolith systems: Functionality redesign complexity and evaluation of similarity measures," *Journal of Web Engineering*, Aug. 2022. [Online]. Available: https://doi.org/10.13052/jwe1540-9589.2158

[8] S. Klock, J. M. E. M. van der Werf, J. P. Guelen, and S. Jansen, "Workload-based clustering of coherent feature sets in microservice architectures," in *2017 IEEE International Conference on Software Architecture (ICSA)*, 2017, pp. 11–20.

[9] S. Hassan, N. Ali, and R. Bahsoon, “Microservice ambients: An architectural meta-modelling approach for microservice granularity,” in *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE, Apr. 2017. [Online]. Available: https://doi.org/10.1109/icsa.2017.32

[10] S. Eski and F. Buzluca, “An automatic extraction approach: transition to microservices architecture from monolithic application,” in *Proceedings of the 19th International Conference on Agile Software Development: Companion*. Porto Portugal: ACM, May 2018, pp. 1–6.

[11] A. Santos and H. Paula, “Microservice decomposition and evaluation using dependency graph and silhouette coefficient,” in *15th Brazilian Symposium on Software Components, Architectures, and Reuse*. Joinville Brazil: ACM, Sep. 2021, pp. 51–60.

[12] J. Löhnertz and A.-M. Oprescu, “Steinmetz: Toward Automatic Decomposition of Monolithic Software Into Microservices,” in *Proceedings of the 13th Seminar Series on Advanced Techniques & Tools for Software Evolution, Amsterdam, The Netherlands, July 1-2, 2020 (due to COVID-19: virtual event)*, ser. CEUR Workshop Proceedings, E. Constantinou, Ed., vol. 2754. CEUR-WS.org, 2020.

[13] G. Mazlami, J. Cito, and P. Leitner, “Extraction of Microservices from Monolithic Software Architectures,” in *2017 IEEE International Conference on Web Services (ICWS)*. Honolulu, HI, USA: IEEE, Jun. 2017, pp. 524–531.

[14] L. Nunes, N. Santos, and A. Rito Silva, “From a monolith to a microservices architecture: An approach based on transactional contexts,” in *Software Architecture*, T. Bures, L. Duchien, and P. Inverardi, Eds. Cham: Springer International Publishing, 2019, pp. 37–52.

[15] J. Bogner, S. Wagner, and A. Zimmermann, “Automatically measuring the maintainability of service- and microservice-based systems: A literature review,” in *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, ser. IWSM Mensura ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 107–115. [Online]. Available: https://doi.org/10.1145/3143434.3143443

[16] G. Kecskemeti, A. C. Marosi, and A. Kertesz, “The ENTICE approach to decompose monolithic services into microservices,” in *2016 International Conference on High Performance Computing & Simulation (HPCS)*. Innsbruck, Austria: IEEE, Jul. 2016, pp. 591–596.

[17] L. Nunes, N. Santos, and A. Rito Silva, “From a Monolith to a Microservices Architecture: An Approach Based on Transactional Contexts,” in *Software Architecture*, T. Bures, L. Duchien, and P. Inverardi, Eds. Cham: Springer International Publishing, 2019, vol. 11681, pp. 37–52, series Title: Lecture Notes in Computer Science.

[18] N. Santos and A. Rito Silva, "A Complexity Metric for Microservices Architecture Migration," in *2020 IEEE International Conference on Software Architecture (ICSA)*. Salvador, Brazil: IEEE, Mar. 2020, pp. 169–178.

[19] B. Andrade, S. Santos, and A. R. Silva, "From monolith to microservices: Static and dynamic analysis comparison," 2022. [Online]. Available: https://arxiv.org/abs/2204.11844

[20] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "An in-depth study of the promises and perils of mining github," *Empirical Software Engineering*, vol. 21, p. 2035–2071, 2016.

[21] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng, "Service candidate identification from monolithic systems based on execution traces," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 987–1007, 2021.

[22] H. Garcia-Molina and K. Salem, "Sagas," *SIGMOD Rec.*, vol. 16, no. 3, p. 249–259, Dec. 1987. [Online]. Available: https://doi.org/10.1145/38714.38742

[23] C. Richardson, *Microservices Patterns: With examples in Java*. Manning, 2018. [Online]. Available: https://books.google.pt/books?id=UeK1swEACAAJ

[24] M. Shapiro and B. Kemme, *Eventual Consistency*. Boston, MA: Springer US, 2009, pp. 1071–1072. [Online]. Available: https://doi.org/10.1007/978-0-387-39940-9{_}1366

[25] R. C. Martin, *Agile software development, principles, patterns, and practices*, ser. Alan Apt series. Upper Saddle River, NJ: Pearson, Oct. 2005.

[26] M. Ahmadvand and A. Ibrahim, "Requirements reconciliation for scalable and secure microservice (de)composition," in *2016 IEEE 24th International Requirements Engineering Conference Workshops (REW)*. IEEE, Sep. 2016. [Online]. Available: https://doi.org/10.1109/rew.2016.026

[27] L. Baresi, M. Garriga, and A. D. Renzis, "Microservices identification through interface analysis," in *Service-Oriented and Cloud Computing*. Springer International Publishing, 2017, pp. 19–33. [Online]. Available: https://doi.org/10.1007/978-3-319-67262-5_2

[28] D. Escobar, D. Cardenas, R. Amarillo, E. Castro, K. Garces, C. Parra, and R. Casallas, "Towards the understanding and evolution of monolithic applications as microservices," in *2016 XLII Latin American Computing Conference (CLEI)*. IEEE, Oct. 2016. [Online]. Available: https://doi.org/10.1109/clei.2016.7833410

[29] A. Levcovitz, R. Terra, and M. T. Valente, "Towards a technique for extracting microservices from monolithic enterprise systems," 2016. [Online]. Available: https://arxiv.org/abs/1605.03175

[30] O. Mustafa, J. M. Gómez, M. Hamed, and H. Pargmann, "GranMicro: A black-box based approach for optimizing microservices based applications," in *Progress in IS*. Springer International Publishing, Aug. 2017, pp. 283–294. [Online]. Available: https://doi.org/10.1007/978-3-319-65687-8_25

[31] L. v. Asseldonk, "From a Monolith to Microservices: the Effect of Multi-view Clustering," M, Utrecht University, 2021. [Online]. Available: https://studenttheses.uu.nl/handle/20.500.12932/148

[32] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 574–586, Sep. 2004.

[33] H. Kagdi, J. Maletic, and B. Sharif, "Mining software repositories for traceability links," in *15th IEEE International Conference on Program Comprehension (ICPC '07)*. Banff, Alberta, BC: IEEE, Jun. 2007, pp. 145–154.

[34] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. New York, New York, USA: ACM Press, 2018, pp. 908–911. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3236024.3264598

[35] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.