TÉCNICO LISBOA

A Complexity Metric for Microservices Architecture Migration

Nuno Alexandre Vieira Santos

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. Luís Manuel Antunes Veiga
Supervisor: Prof. António Manuel Ferreira Rito da Silva
Member of the Committee: Profª Maria Dulce Pedroso Domingos

October 2019

Acknowledgments

First i would like to thank my parents, sister and grandparents for all the support over all these years, encouraging me to follow my dreams, helping me grow as a person, financially supporting me and giving me the opportunity to go through and conclude this phase.

I would also like to acknowledge my dissertation supervisor Prof. António Rito Silva for his insight, support, perseverance and sharing of knowledge that has made this Thesis achievable. Without his deep understanding and knowledge on this subject this work wouldn't be possible.

Last but not least, to all my friends and colleagues that helped me throughout this course and were always there for me during the good and bad times in my life.

To each and every one of you – Thank you.

Abstract

Monolithic applications tend to typically be difficult to deploy, difficult to upgrade and maintain, and difficult to understand. Microservices, on the other hand, have the advantages of being independently developed, tested, deployed, and scaled and, more importantly, easier to change and maintain. This thesis is focused on the migration from a monolithic to a microservices architecture. In our work we address two new research questions: (1) Can we define the cost of decomposition in terms of the effort to decompose a functionality, implemented in the monolith as an ACID transaction, into several distributed transactions? (2) Will the use of similarity measures that consider information about reads and writes and the sequences of accesses provide a better decomposition? To answer our first research question, we propose a complexity metric for each functionality, in the context of a particular decomposition, in order to provide a better insight into the cost of the decomposition. Regarding our second research question, we propose the experimentation with four similarity measures, each based on a different type of information collected from the monolith. We evaluated our approach with three monolith systems and compared our complexity metric against two industry metrics of cohesion and coupling. We evaluated our similarity measures against the complexity of the decomposition obtained. We also compared, for each system, our generated decompositions with the ones created by an expert of the system. We were able to correctly correlate the complexity metric with other metrics of cohesion and coupling defined in other research. For the second research question, we concluded that no single combination of similarity measures outperforms the other, which is confirmed by the existing research. We found that the developed tool can help on an incremental migration to microservices approach, which is actually

defended by the industry experts.

Keywords

Microservices; Monolithic Application; Transactional Context; Architecture Migration; CAP Theorem.

Resumo

Aplicações monolíticas costumam ser difíceis de implementar, difíceis de atualizar e manter e difíceis de entender. Os microserviços, por outro lado, têm as vantagens de serem independentemente desenvolvidos, testados, implementados e escalados e, mais importante, mais fáceis de alterar e manter. Esta tese está focada na migração de uma arquitetura monolítica para uma de microserviços. No nosso trabalho, abordamos duas novas questões de investigação: (1) Podemos definir o custo da decomposição em termos do esforço para decompor uma funcionalidade, implementada no monólito como uma transação ACID, em várias transações distribuídas? (2) O uso de medidas de similaridade que considerem informações sobre leituras e escritas e as sequências de acessos proporcionará uma melhor decomposição? Para responder à nossa primeira pergunta de investigação, propomos uma métrica de complexidade para cada funcionalidade, no contexto de uma decomposição específica, a fim de fornecer uma melhor visão do custo da decomposição. Em relação à nossa segunda questão de investigação, propomos a experimentação com quatro medidas de similaridade, cada uma baseada em um tipo diferente de informação recolhida do monólito. Avaliamos a nossa abordagem com três sistemas monolíticos e comparamos a nossa métrica de complexidade com duas métricas de coesão e acoplamento da indústria. Avaliamos as nossas medidas de similaridade em relação à complexidade da decomposição obtida. Também comparamos, para cada sistema, as nossas decomposições geradas com as criadas por um especialista do sistema. Conseguimos correlacionar corretamente a métrica de complexidade com outras métricas de coesão e acoplamento definidas em outros artigos. Para a segunda questão de investigação, concluímos que nenhuma combinação única de medidas de similaridade supera a outra, o que é confirmado pela investigação existente. Descobrimos que a ferramenta desenvolvida pode ajudar na abordagem de migração incremental para microserviços, o que é

defendido pelos especialistas da indústria.

Palavras Chave

Microserviços; Aplicação monolítica; Contexto Transacional; Migração de Arquitetura; Teorema CAP.

Contents

List of Figures

List of Tables

1

Introduction

Contents

Microservices are designed around business capabilities such that they can be assigned to agile small cross-functional teams, which reduces the conceptual gap between business requirements, functionality implementation, service deployment and operation. This is achieved through the definition of independently deployable distributed services, which have well-defined interfaces, very often maintaining several versions of the interface to reduce inter-team communication, and using lightweight communication mechanisms [1]. Contrarily, monolithic applications have limitations in terms of agility, they have a large shared domain model through which all teams have to communicate, do not support different levels of horizontal scalability per service type, which compromises availability and fault-tolerance. Those are the main reasons that triggered the wide adoption of the microservices architecture, such that, well-known companies such as Netflix, Amazon and Ebay have decided to make the transition to microservices [2] and more existing monolith systems are being evolved to a microservices architecture.

However, the migration of monolith systems to microservices architecture raises new problems associated with the need to get microservices boundaries right, the complexity associated with asynchronous distributed computing, and the relaxing of atomic transactional behavior. Additionally, microservices architecture is not a silver bullet of software engineering and, therefore, it is important to evaluate whether it is worth migrating the monolith to microservices.

1.1 Context

This thesis follows a previous one [3] where a strategy was created to provide a microservice decomposition and a tool was developed as a proof of concept. The strategy is separated in three main stages:

1. Collect the callgraph of the monolith system using static analysis. The callgraph provides the method call chain (callees and callers of a method) which can be used to retrieve the domain entities accessed by each functionality. This was done using the JavaCallgraph[1] tool.

2. Define clusters using a similarity measure. The similarity is calculated by attributing a weight to the relationship between two domain classes. The weight from a class C1 to a class C2 is the quotient between the number of controllers that call both classes and the total number of controllers that call C1. An Hierarchical Clustering algorithm[2] is used to define the clusters, where each cluster corresponds to a microservice.

3. Present in a visualization tool the different partitioning hypothesis in the form of a graph, where the developer can analyse and assess each one.

[1] github.com/gousiosg/java-callgraph
[2] docs.scipy.org/doc/scipy/reference/cluster.html

1.2 Problem

Transaction management is more complex in a microservices architecture, because each service has its own database, and the implementation of a functionality is scattered across several distributed microservices. On the other hand, functionalities in monolithic applications are simpler to implement because they have a single database and their transactional manager provides ACID properties which the developers can rely on, such that they can focus on the implementation of the business logic.

Therefore, when introducing microservices, it is necessary to decide on the functionalities trade-off between consistency and availability, as explained by the CAP theorem [4], where the decision to have strict consistency can only be achieved in a distributed transactions context, by the application of a two-phase commit protocol which does not scale to support many replicas.

Consequently, in a microservices architecture it is necessary to decide what is the attainable level of consistency to associate with functionality, given the requirements on availability. Currently, the practice of microservices development uses patterns to handle this problem, like the saga pattern [5]. However, the use of these techniques is not free of cost, both in terms of the system behavior and its implementation:

- From an end-user perspective, there is an impact on the perceived behavior of the system, because intermediate transactional states may be observed due to the relaxed consistency and the occurrence of failures. Therefore it is necessary to do requirement analysis activities with some of the application stakeholders.

- Functionality implementation becomes more complex, because the developer, besides focusing on the business logic, also has to consider all the situations of possible faults in the microservices that implement a functionality, and write code that handles them, like the implementation of compensating transactions. Additionally, the functionality business logic easily become intertwined with the handling of all faults, which make its implementation more cumbersome.

1.3 Research Questions

In this thesis we address the problem of microservices identification in a monolith while managing the impact that the relaxing of atomic transactional behavior has on the redesign and implementation of the new system from the monolith.

Relaxing the transactional behavior associated with business transactions in a monolith system impacts on the application functionality due to the change in the consistency of information, which may require the redesign of functionalities. On the other hand, the implementation of the distributed business

transactions is costly because, to achieve the desired level of consistency, it is necessary to intertwine distributed systems techniques with the functionality business logic.

Interestingly, this has been neglected in the literature where most proposals ignore its impact on the migration process, both in the identification of candidate microservices and in the migration implementation. This situation has been coined in [6] as the *Forgetting about the CAP Theorem* migration smell, which states that there is a trade-off between consistency and availability [7].

In the previous thesis, decompositions were created based on accesses only and validation was done with several systems based on the expert decomposition. This validation didn't allow to measure the quality of the decomposition which was focused in the comparison with the expert. With our approach we intend to improve on the previous work and respond to the following research questions:

1. Is it possible to calculate the cost associated with the migration to a microservices architecture due to the introduction of relaxed consistency into the business behavior?
2. Which similarity measures are more effective in the generation of a candidate microservices decomposition in terms of the cost of migration?

The visualization and modeling tool is used for the evaluation of the research questions. Therefore, the application of the approach is done to three monolith systems and the result decompositions are analysed in the visualization tool. Additionally, decompositions done by domain experts of the three systems were also introduced in the visualization tool to allow the comparison with the candidate decompositions.

1.4 Contributions

This thesis leverages on a previous work which addresses this problem by defining a similarity measure between two entities in terms of the functionalities that access them. In this thesis we extend this migration approach such that:

- Extracts from the monolith information about its business transactions, which domain entities they access, categorize the accesses in terms of reads and writes, and the sequences of domain entities accesses for each business transaction;
- Defines several similarity measures that are used to identify microservices candidates;
- Define a metric to measure the quality of a microservices systems in terms of the cost associated with having non atomic business behavior;
- Provide a visualization tool to navigate and analyse the candidate decompositions and allow the comparison between several decompositions using the previous metric;

- Enhance the visualization tool with modeling capabilities that supports the manipulation of a candidate decomposition informed by the real-time recalculation of the metrics after each change is done to the model.

The results of this thesis are the following:

- Identification of what are the best similarity measures to generate a set of candidate microservices from a monolith system.
- Definition of a complexity metric for microservices systems in terms of the impact on functionality partitioning and its implementation.

1.5 Organization of the Document

This thesis is organized as follows: Chapter 2 presents the state of the art in microservices migration research and also an analysis on metrics for microservice evaluation. In chapter 3 is described our approach to the generation of candidate microservice decompositions and our developed metrics to assess cost of migration. In chapter 4 we evaluate our approach with three monolith systems with expert decompositions and evaluate our metrics against others from the industry. Chapter 5 concludes this thesis with final remarks and an overall summary of the work, and describe the possibilities for future work.

2

Related Work

Contents

In this chapter we start by presenting existing research in microservices migration. This area of investigation is relatively new which means we are working with very recent research that is being updated on a regular basis. Most of the strategies from other approaches can be grouped in the same workflow which is first to gather information/data from the monolith system and then using that information to reason about a possible decomposition. The work in this section is discussed having in mind our approach, which means we compare the differences in both strategies. We are interested, for each related work, in what information is gathered, and how is that done, how it is processed and what are the results achieved.

The second part of this chapter is our survey into microservice metrics and other related software metrics. This metrics provide a way to assess the cost of transitioning to microservices and the additional cost this architecture can have. We analyse what are the metrics developed in the industry. Most metrics are concerned with complexity, cohesion and coupling.

2.1 Migration Approaches

In [8] they identified and reviewed ten existing strategies to break down a monolithic application into microservices. To identify this ten strategies, they used a search strategy where they query three of the most frequented scientific libraries and indexing systems in computer science: ACM Digital Library, IEEE Xplore and Google Scholar. In the query they search for articles where some words like microservices, monolith, refactor, migration, decomposition, granularity appear. Then they chose ten of the results obtained based on technical depth, recency and relevance of the content presented. They reviewed and classified each one based on the strategy used: static analysis, meta-data aided, workload-data aided and dynamic microservice decomposition. We now discuss some of these articles and others.

In [9] they present an approach to tackle the challenges of modernization in legacy JEE (Java Enterprise Edition) applications using static code analysis. They start by gathering information from the source code to produce a model representation of the application. This model is produced using MODISCO and is represented as an Abstract Syntax Tree (AST) similar to the one produced by the Eclipse JDT API. In the second step they filter and transform the elements in the model obtained previously. They start by finding the most relevant elements such as classes, interfaces and their attributes and method signatures. This step will produce a simplified version of the previous model. They proceed with two different clustering transformations. The first one is the EJB clustering. For this transformation they defined an algorithm that produces a graph where the nodes represent types and the edges represent type relationships. The algorithm starts by selecting pairs of nodes (A, B) where A is an EJB and B is an entity bean, and calculates the invocation path from A to B. After that, the algorithm finds all the invocations that start in A and finish in B, to identify the types in between. Next, for each EJB, the algorithm creates

a cluster and associates to it all the types found in the path (there may be types that belong to two or more clusters). After the algorithm creates the clusters, it establishes relationships between each pair of clusters representing the percentage of types in common. The second transformation is the Microservice clustering. For this part they defined a clustering threshold that is specified by the user and used a hierarchical clustering algorithm. The algorithm iterates through all clusters defined and validates if any of its cluster relationships has a percentage higher than the clustering threshold. If it has then the clusters will be grouped in the same microservice. In the end several microservices are created containing one or more clusters. In the last step they show the information in a graphical way with several diagrams (EJB-Data, Shared Types, microservices and microservices invocation diagrams). For the Evaluation, they analysed a large JEE application. In the obtained EJB-Data diagram (after the algorithm creates the clusters) they concluded that most of the clusters (113) were coupled with each other and only 59 were loosely coupled and would result in an isolated microservice (few or none relationships with each other). In the Microservices diagram, they applied 3 different clustering thresholds of 5%, 50% and 80%. They observed that, for the 5% and 50%, the number of types contained in the microservice were very unequal. The clustering threshold that gave the best proportion of number of types was 80%, where they obtained 67 microservices, 24 of which were isolated. In the end they didn't have any analysis to verify if each proposed cluster would actually correspond to a microservice.

Their idea can be considered similar to the workflow created in the previous thesis [3]. First, they use static analysis and method introspection to reason whether two elements are highly/loosely coupled and to obtain dependencies between elements. This is equivalent to our part where we obtain the domain classes that a controller accesses. The only difference in this part is that they gather information about the entire invocation path, where we only register the first and last element of that invocation path. Relating to ours, we can consider that their EJBs are equivalent to our controllers because they are associated with a session, and the entity beans to our domain classes.

In their second part they use a clustering metric that considers, for each pair of clusters (each cluster representing an EJB), the number of entity beans accessed by both. Our metric is identical in the sense that we start from the domain classes (equivalent to the entity beans) and consider the controllers they access in common. If we applied our clustering metric in their work, their metric would be, for a pair of entity beans, the number of EJB accessed in common. We didn't mathematically prove if the two approaches are equal or not, but as far as we can tell they seem to provide similarity matrices that will lead to the same decomposition. We also consider that a transaction should be contained in a single microservice, and they also follow that idea when they group EJBs that access the same entities. They provide additional information that we don't which is, after the partition, they provide a list of dependencies between microservices that represent the invocations between them. In the end they also provide a graphical way to show the decomposition.

In [10] they describe a technique to identify and define microservices on a monolithic enterprise system. They only consider monolithic enterprise applications with three tiers: client side user interface, server side application and a database. They also consider that a large system is structured on smaller subsystems and each subsystem has a well-defined set of business responsibilities. They also assume that each subsystem has a separate data store. Their strategy can be divided in the following steps:

1. Map the database tables into subsystems where each subsystem represents a business area.
2. Create a dependency graph where vertices represent façades, business functions, or database tables, and edges represent calls from façades to business functions, or calls between business functions, or accesses from business functions to database tables.
3. Identify pairs (façade,database table) where there is a path from the façade to the database table on the dependency graph.
4. Select, for each subsystem, the pairs where the database table is present on that subsystem.
5. Identify candidates to be transformed to microservices. For each pair obtained previously they inspect the code of the façade and business functions that are on the path from the façade to the database table in the dependency graph. The candidates are then distinguished in 3 types: strong candidates, candidates with additional effort and non-candidates. If a candidate accesses a database table outside its scope then it will be considered a non-candidate. When they can easily synchronize the calls between the new microservice and the façade then it is considered a strong candidate. Otherwise, it is a candidate with additional effort.
6. Create API gateways to turn the migration to microservices transparent to clients.

In the end they applied their technique on a large system from a bank. They classified their study as well-succeeded because they could identify and classify all subsystems, and create and analyze the dependency graph that helped to identify microservices candidates.

Their idea can be correlated with ours, where their database tables represent our accesses to the Fénix Framework, the façades represent the entry points in the application which in our case are the controllers, and the business functions in between are our domain classes. In a first phase, they don't do any sort of automatic code analysis. From our understanding, they manually inspect source code in order to obtain the calls from the façades to the database tables and to build the dependency graph. The information they obtained can be correlated to ours, in the sense that the calls from the façades to the database tables represent the same idea as our controller calls to domain classes. In their second part, they don't use any clustering algorithm and metric. They only mention that they manually inspect the code for each (façade,db table) tuple in order to identify which business rules actually depend on the database table. In our interpretation, they only identify the candidates for microservices through

manual inspection. Since they don't have any clustering metric, they can only give one possible decomposition, and therefore don't allow to experiment different levels of granularity in order to produce different decompositions. Overall, they define a microservice as containing a single business function that occurs in the beginning or in the end of a façade (in the middle needs more effort), and doesn't access any tables of other subsystems. Their assumption in the beginning that the system needs to be divided in smaller subsystems where each has a well defined set of responsibilities facilitates the work they present, because the subsystems already represent the transactional context. In the end they create API Gateways to materialize the migration. This part is not in the scope of our work, because we only present a decomposition hypothesis but we don't try to execute it, where in their case they present a technique to actually execute the migration.

In [11] they propose a methodology for microservice decomposition from system requirements (functional and non-functional), with emphasis to the security and scalability requirements. Besides the security and scalability requirements they also require the dependencies between them as inputs to their methodology.

They start by manually determining the level of scalability required for each functional requirement. Then, they manually assess the impact of each security requirement. They assess the impact of each requirement by attributing a level to it (High, Medium or Low). Their methodology starts by considering each functional requirement as a microservice candidate. Afterwards they classify each candidate in terms of the impact of the non-functional requirements of scalability and security calculated in the beginning and the level of dependency between them. The candidates for which it is expected to have a high volume of requests are considered to require scalability. Then, for those with high and medium scalability it is verified the level of dependence with the other candidates, where an high level of dependence corresponds to the frequency of invocations between them. If they are highly dependent and require security, which results in a high overhead, the candidates will be merged into a single microservice.

In comparison to our work, we can consider that each requirement/use case they have corresponds to one of our controllers. They manually identify the impact of the security and scalability requirements for each use case, and also manually identify the dependencies between use cases. In our work we obtain this dependencies programatically in our first phase, which corresponds to the static analysis. We are not able to identify the frequency of the invocations on microservices and between microservices, though we can use the number of calls between microservices as a measure of frequency, but only a monitoring technique could automatically allow to obtain these values. On the other hand, we don't address the security issue because we don't consider the security overhead created by the invocations between microservices. Their main emphasis is on the reduction of the overhead in the invocations between microservices, and on the identification of the services that need to scale. They don't do any kind of clustering, they identify microservices based only on the methodology referenced before. The

corresponding for our clustering metric would be the level of impact attributed to each requirement. They don't mention any visualization capabilities for the decomposition, but they show a figure with the different microservices obtained and the dependencies between them. In the end, they only show a case study to illustrate their work, but they don't do any evaluation.

They conclude that the security and scalability are not the only factors that should be considered in microservice decomposition. Other factors such as cost, maintainability and structure of development teams should also be considered.

In [12] they propose a solution based on the semantic similarity of interfaces described through OpenAPI specifications. They require as input the reference vocabulary and the OpenAPI descriptions of the interfaces which can be automatically generated if needed. The process starts with mapping available OpenAPI specifications onto the entries of a reference vocabulary by means of a fitness function. This step will generate a mapping between each operation in the input and a reference concept in the vocabulary, that is, the concept that most accurately describes the operation. The idea is that operations that share the same reference concept are highly cohesive, and should be grouped together, which means that they are looking for single responsibility. Then, the suggested decomposition comprises one candidate microservice per identified reference concept, which is associated with a set of operations. Each microservice is defined through its operations and their parameters, (public) complex types, and return values. The user is able to play with different values for the granularity level, identify different groupings, and analyse them. They evaluated their approach using a cargo tracking application. They compared their decomposition with the one produced by Service Cutter [13]. They generated different decompositions but they said that neither of the approaches returned the expected decomposition. The main differences were that they decomposed a part of the application in a more fine-grained way in comparison to Service Cutter. They also experimented with other microservice applications and obtained results that suggest that their approach is able to detect correct candidate microservices for around 80% of an application's functionality, given that the expected decomposition (gold standard) was known beforehand. Regarding the limitations, they noticed that the input artifacts may be mapped to too few concepts of the shared vocabulary, and thus the decomposition would generate coarse-grained microservices. Also, their approach relies on well defined and described interfaces that provide meaningful names and follow programming naming conventions such as camel casing and hyphenation, which isn't always the case.

Relating to our work, they also gather information of the application in the first phase. They use the OpenAPI specifications where we make use of static analysis to obtain controller calls. For this phase they require the OpenAPI specifications where we need the source code of the application. They don't use a clustering algorithm, instead they group operations based on whether they share or not the same reference concept. They define similarity scores between two terms to determine whether they are

similar or not which corresponds to the part where we define weights between domain classes (using the number of controllers in common). The corresponding to our clustering metric would be the threshold they define for the similarity score. As far as we can tell they also offer visualization capabilities and the possibility to change the granularity level of the decomposition.

In [14] they present a technique where they first transform the monolith into a graph representation (extraction step) by analysing the applications source code and the version history (provided by a version control system as Git, etc..). In the second part they decompose the graph representation into microservice candidates (clustering step) using a clustering algorithm. For the first step, they present three different extraction strategies:

- Logical Coupling, meaning that class files that change together should consequently also belong to the same microservice. They do this by analysing the commits from the version history to obtain information about which classes are associated with the same commits. This strategy provides the weights to the graph where a higher weight means that the classes usually change in the same commits.
- Semantic Coupling, couples together classes that contain code about the same domain model entities based on the semantic similarity between them. They try to identify high level topics or domain concepts based on identifiers and expressions in the source code. They use the term-frequency inverse-document-frequency (TFIDF) method and compute a scalar vector for a document given a predefined set of words. Using these methods they can compute a score that indicates how related two files are in terms of domain concepts expressed in code and identifiers.
- Contributor Coupling, aims to incorporate team-based factors into a formal procedure that can be used to cluster class files by analyzing the authors of changes in version control history.

In the second part they present a clustering algorithm to partition the graph described as follows:

1. Obtain the Minimum Spanning Tree of the graph using the Kruskal algorithm.
2. Then they do a iterative edge deletion loop from the MST. The number of iterations is obtained as a user input parameter and it will define the number of partitions that the algorithm attempts. In each loop they delete the edge with the minimum weight (lowest coupling) from the MST.
3. In the end they obtain a set of connected components where each represent a candidate microservice.

They evaluated their approach on a open-source Java project and their main concerns were on performance and quality. They experimented with each one of the three extraction strategies presented. For the performance, they measured the execution times of each one of the extraction strategies in sample

projects. They concluded that all the extraction strategies showed satisfying performance levels. For the quality evaluation they used the team size reduction metric and the average domain redundancy metric. The team size reduction metric is computed as the average team size across all microservice candidates in the microservice decomposition divided by the team size of the original monolith. This metric shows the possible reductions in team sizes that the new microservice architecture will offer. The average domain redundancy metric indicates the amount of domain-specific duplication or redundancy between the proposed microservices. For both the metrics, they concluded that the semantic coupling strategy clearly showed the best results.

Their approach can be considered similar to ours. Both approaches follow a similar process, extraction and clustering, but they don't do any visualization. In the extraction phase, both approaches make use of the applications source code. On the other hand, they use different measures, which are based on the version history, except for the semantic coupling, which is based on the TFIDF method, while ours is focused on the relation between controllers and domain classes. Both approaches create a graph representation where the weights represent the level of coupling between vertexes. In the clustering phase, they create partitions based on cutting low weighted edges from the graph, while we define a threshold to create those partitions.

In [15] they study the economic impact caused by granularity level in microservices based web applications. They examine the impact of different granularity levels on performance levels and resource utilization levels, for the same workload. Their strategy is based on a black box technique where they gather information about the web logs in order to identify loaded services. The web logs are used in a fuzzy clustering algorithm to create a workload pattern and consequently identify the services that are having high workloads. In the end, they decide to do the decomposition of only loaded services. They concluded that splitting the loaded services decreased the applications response time and CPU utilization. They don't mention the metric used to decide whether a service is loaded or not. They only do an empirical evaluation of their idea but they didn't implement it.

Their approach is based on a black box technique where ours is based on a white box technique. They obtain information from web logs where we do source code analysis. Fundamentally, the two approaches are different. We are making the migration from a monolithic architecture to a microservices one, where they already start from a microservices architecture and focus on the introduction of concurrency by reimplementing the web pages with a high workload with a set of concurrent microservices.

In [16] they propose a deployment of a microservices application that improves the performance for a given workload using a genetic algorithm. They require a feature model that represents a graph of features and the dependencies between them where each feature has a set of properties. They consider a very particular architectural design which considers that all communication is event driven between different instances of the same feature. In the event-driven microservice architecture, the

edges between microservices don't exist, and therefore they add internal feature instances to satisfy the feature dependency graph. However, adding the internal feature instances results in data and code duplication, which in turn requires an increase in communication. Therefore, they try to minimize the number of internal feature instances by grouping dependent features in the same microservice. They also require the current microservice architecture and its workload information. They obtain the workload from the access logs, which contain information about which feature has been called, and by whom and when. This way they derive the usage of features. They also retrieve performance metrics from the access logs, such as memory usage or request time, which helps them to understand what are the most overloaded features. After they have the required workload and feature information, they use a genetic algorithm with a fitness function to define clusters where each cluster is defined as a set of features. They evaluate their approach with a case study from an ERP system. They test two different scenarios: low workload, between 5 and 20 requests per second, and high workload, between 50 and 200 requests per second. They compare the obtained decomposition with the original one, and, for each scenario, they compare the request time and the average of requests the system handled per second. For the workload information they use a simulation based on a set of predefined profiles. In the low workload scenario, they concluded that their proposed decomposition reduced the total time of the performance test by 20% and the throughput of the system increased by 23% on average. In this scenario their decomposition is a single microservice. In this case, the overhead of processing every request multiple times by different microservices is larger than the benefits gained by the increased parallel processing. In the high workload scenario, the performance could not be improved substantially, but they were able to reduce the duplication of internal feature instances by 30%. In this case combining several microservices that contain the same internal feature instances resulted in a substantial decrease of the duplication, without a negative impact on the performance.

Their approach has some differences compared with our approach. Fundamentally, our idea is to migrate a monolithic application to a microservices architecture, where they start from a microservices application and optimize it depending on the workload and the feature model. They obtain the workload information dynamically from the access logs while the system is in production. The feature model is obtained manually, which can be a problem because they need to define what are the features and it can be a cumbersome task, but they don't address it. In our case we require the source code and obtain the information through static analysis. In the clustering phase, we group domain classes where they group features and the algorithms used are different. They don't detail about the capabilities for the visualization but they mention that they have a web-based microservice architecture model viewer.

In [17] they present a data-driven approach to define a microservices architecture from an existing software system. They use static and dynamic techniques for information gathering. The static information is obtained from the Abstract Syntax Tree (AST) that is obtained from the source code. They

also require a ontology of concepts. They extract dynamic information from execution logs and traces. Their idea is to gather data from an existing system to reason about which services are actually being executed. The goal is to describe a microservice as a set of functional properties and non-functional capabilities. They use the dynamic information to infer the non-functional capabilities. They feed the dynamic and the static data to a process mining algorithm. The algorithm extracts information by matching the ontology concepts with the function definitions from the AST to identify functional elements. In the end the algorithm uses the provided data to identify the candidate microservices. They implement a proof of concept for their approach but they don't mention any evaluation. They only mention a validation process that checks whether the extracted architecture model is conformant with the specifications, but they don't detail it farther than that.

Comparing with our approach, we can say that both approaches gather information from the source code. They use dynamic analysis but we don't. They don't explain clearly their idea of what a microservice should be and what their algorithm does. They seem to have an algorithm to create a decomposition but they don't explain how they define the microservices and what metric they use. They don't perform any evaluation and don't mention visualization capabilities.

In [18] they identify microservices from the business processes point of view. They consider BPMN business processes with data object reads and writes. The most important in each business process are the activities, each representing a unit of work, the gateways and the set of objects accessed in each activity. Their strategy is to group activities to form a microservice, where each activity plays the role of an operation. They start by defining two relations for each pair of activities. One represents the structural dependency between activities and is calculated based on the following metric: if there is a direct edge between two activities or there is a path between them containing only gateways, then those two activities are inter-connected (assigned weight 1, otherwise 0). The second one shows the data dependency (read/write) based on the used data objects for each pair of activities. They assign weight 1 to the objects that are written by both activities, 0.5 to the objects that are written by one and read by the other activity, and 0.25 to the objects that are read by both activities. After the two relations are calculated, they sum them to define the final relation. This final relation represents the dependency weight between activities that will be used to group them. After the final relation is calculated, they use a genetic clustering algorithm and turbo-MQ fitness function. The algorithm randomly partitions activities into K clusters and use turbo-MQ to measure the fitness of the identified clusters. Then, in each iteration, the algorithm tries to increase the fitness by making changes in the previous partitions. This process continues until the fitness converges. They obtain in the end a set of clusters, each representing a microservice, and each cluster is composed as a set of business activities. They conducted four experiments to evaluate their approach. For each one, four parameters were considered: number of activities, number of gateways, number of objects, and number of processes. In each experiment they study the

effect of one parameter on the accuracy of the identification approaches. To calculate the accuracy they compare the number of activities that are clustered in the correct microservice. They asked a group of domain experts to obtain the correct microservice decomposition. In the first experiment they variate the number of activities and concluded that the accuracy of the proposed method is almost independent of the size of the process. In the second experiment they variate the number of gateways and concluded that the accuracy of the proposed method is almost independent of the complexity of processes. In the third experiment they variate the number of objects and concluded that the proposed method is not independent of the number of objects. They observed that increasing the number of objects involved improves the accuracy. In the fourth experiment they variate the number of processes and concluded that the accuracy of the proposed method is almost independent of the number of processes involved.

Their approach is similar to ours. We require the applications source code where they require the applications business processes containing the data objects accessed in each activity. Both start by attributing dependency weights between entities (activities to them, domain classes to us) and then grouping these entities by using a clustering algorithm. Regarding the metrics, they follow the idea that activities that access the same data and are in the same control flow should be part of the same microservice, which differ from our approach because we don't consider any predefined execution sequence between controllers. To do this they combine a structural and data dependency metric where we use the number of controllers in common. We use a hierarchical clustering algorithm where they use a genetic algorithm with a fitness function. They don't mention any visualization capabilities.

In [13] they present Service Cutter, a tool framework for service decomposition. They have four core concepts:

- Nanoentities - can be of 3 types: data, operation and artifact. Data represents the persistent entities, operation are business rules or calculation logic, and artifact is a collection of data or operation results.
- Coupling Criteria - captures the architecturally significant requirements and arguments why two nanoentities should or should not be owned and exposed by the same service. They propose 16 coupling criteria distilled from the literature and industry experience. These criteria are divided in four main categories: cohesiveness, compatibility, constraints and communication.
- User representation - the artifacts they require as input that they call System Specification Artifacts (SSA), such as use cases, Domain Driven Design entities, and Entity-Relationship models. SSA represents the analysis and design artifacts that contain information about coupling criteria.
- Priority Scoring - they attribute weights between nanoentities to express whether they should be in the same service or not. To calculate this weight they sum the weight of all the criteria where in each one they multiply the level of priority of that criteria with the score. The level of priority is

a user defined number that indicates the most important criteria for the decomposition. The score is a number between -10 and +10 and expresses whether the two nanoentities should be in the same service or not, according to a certain coupling criteria.

After they have the undirected weighted graph with the nanoentities, they use a clustering algorithm to obtain a decomposition. Each cluster will correspond to a microservice and each microservice is a group of nanoentities. They propose a deterministic and a non-deterministic clustering algorithm, where the deterministic requires the number of clusters to be created as input. In the end they provide a web application for input of the priority levels and output visualization of the decomposition.

For the evaluation, they compare the decomposition obtained by Service Cutter with one produced beforehand manually called baseline (obtained from their experience of the application), for a trading and a cargo application. They rate the candidate service cuts in three categories: excellent (the obtained service cut is different from the baseline, but it improves the expected baseline), expected (the obtained service cut is the same as the baseline), and unreasonable (mismatch between the service cuts, and no benefits from the obtained one). To assess the quality of the decomposition they use a four-level classification: excellent output contains zero unreasonable service cuts and at least one excellent service cut. A good output contains zero unreasonable service cuts. An acceptable output contains at most one unreasonable service cut. A bad output contains two or more unreasonable service cuts. They concluded that both clustering algorithms produced acceptable or good service cuts, for both the cargo and the trading system.

In comparison with our approach, their approach provides more variety because they can fine tune the priorities to guide the decomposition where we focus more on the data accesses and consistency criteria. They require a heavier input than ours because we only need the source code, where they require a model of the system in a predefined structure, such as an ERM. Both approaches create a weighted graph representation where their nodes are nanoentities and ours are domain classes, but ours is a directed graph where theirs is an undirected one. For the clustering metrics they created a scoring system where we considered the controllers in common. Both use a clustering algorithm to group the entities and form the microservices. Both provide visualization capabilities for the decomposition.

In [19] they propose a visualization tool for microservices decomposition starting from a monolith application. They start by generating a calling-context tree (CCT) from a profiler in a dry run of the application. This tree is similar to a call graph, where we have information about the function calls of the system. They use this information to estimate the amount of communication between components. After they have the CCT, they propose two different clustering techniques. The first is a semantic-based clustering that considers text features in source files. It composes a microservice with similar classes in terms of the content of each class. They use k-means++ with tf-idf-based similarity. The second is a CCT-based clustering that considers the amount of communication. Its main concerns are in reducing

the amount of communication and thereby providing high performance. In this algorithm they calculate similarity between two functions, and give high similarity score to two functions on the same call flow. After they apply the clustering, they show the initial decomposition in a visual interface. The decomposition is presented as a directed graph where the nodes represent classes and the edges represent function calls. The tool allows for developers to refine the decomposition by performing four actions: (1) move the class from a microservice into another microservice, (2) clone the class into multiple microservices, (3) create a new microservice for the class, and (4) leave the class as is. The tool also recommends class files that should be considered to significantly reduce the amount of communication by displaying a list that shows classes in descending order of total number of function calls to/from classes in other microservices. It also provides views of the source code, commit information, and communication amount. For the evaluation, they presented two case studies, where the applications provide implementation of monolithic and microservice model (were manually migrated before). They ran their tool against the monolithic implementation, and compared their decomposition with the one produced by the system developers. They concluded that compared to the official microservice designs of the application, the proposed tool can effectively design microservice applications.

Comparing with our approach, they follow a similar three step strategy: extraction, clustering, visualization. In the first one their information is obtained through dynamic analysis (profiler) where we use static analysis. They present two clustering techniques based on semantic similarity and on the call graph. The first one is similar to other approaches such as [12]. The second one is more similar to ours because its also based on call graph information. For the visualization, they also provide a view of the decomposition and the ability to change it, although their approach provides a more fine-grained way to change the decomposition, because they can transfer and clone classes between microservices.

To conclude the analysis on microservice identification, most of the articles follow a strategy similar to ours that starts by gathering information about the application entities, mainly by using analysis techniques such as static and dynamic analysis. A clustering algorithm, or some form of grouping methodology, is then used to create clusters of entities which correspond to microservices. Visualization capabilities for the microservice decomposition are occasionally found and often don't provide the ability to change the decomposition, as we do by cutting it and changing the granularity level. Only [19] provides a richer visualization, where it is possible to fine-tune the entities in the proposed microservice. In our approach we are interested in distinguishing clusters based on the transactional context by considering the data accesses. The idea is that the entities involved in a single transaction should be in the same microservice. From the approaches studied we found two that can relate with ours: the data dependency metric in [18] considers the data accesses in BPMN activities, and the consistency constraint coupling criteria in [13] considers that entities in the same database transaction should be grouped together, but they don't mention how that coupling criteria is measured. In [12] they compared

their approach against Service Cutter [13] and concluded that Service Cutter requires a detailed and exhaustive specification of the system, together with ad-hoc specification artifacts associated with coupling criteria, and the availability of that documentation is arguable, where they present their approach as only requiring lightweight input that can be automatically generated. Other approaches are based on system requirements and use cases [11], on semantic similarity [12] [14], or on workload information [16] [15].

2.2 Microservice Metrics

In this section we present a survey into the state of the art regarding software metrics in microservices architecture and service oriented architectures. Metrics for microservices are still relatively new and many descend from existing metrics for service oriented architectures. Assessing software quality through the creation of metrics exists for a long time, and new metrics need to be adapted as new architectures appear. Most articles present metrics of complexity, cohesion and coupling. The IEEE Standard Computer Dictionary defines complexity as the degree to which a system or component has a design or implementation that is difficult to understand and verify [20]. Cohesion measures the degree to which the elements of the system belong together [21]. Coupling is a measure of the extent to which interdependencies exist between software modules [22].

In [23], they propose a metrics suite for object-oriented methodologies, and present a solid base for the reasoning behind our metrics. Their metrics are proved according to the Bunge's ontology [24], and are explained below:

- Weighted Methods per Class - measure the complexity of a class as the average complexity of the methods in it. This definition comes from Bunge's definition of complexity of a thing, since methods are properties of object classes and complexity is determined by the cardinality of its set of properties.
- Depth of Inheritance Tree - measures the maximum length from the node to the root of the tree. This measure how many ancestor classes can potentially affect this class.
- Number of Children - number of immediate subclasses subordinated to a class in the class hierarchy. It is a measure of how many subclasses are going to inherit the methods of the parent class.
- Coupling between object classes - count of the number of other classes to which it is coupled. This relates to the notion that an object is coupled to another object if one of them acts on the other, i.e., methods of one use methods or instance variables of another. Since objects of the same class have the same properties, two classes are coupled when methods declared in one class use methods or instance variables defined by the other class.

- Response for a class - the response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class. Since it specifically includes methods called from outside the class, it is also a measure of the potential communication between the class and other classes.
- Lack of Cohesion in Methods - is the intersection of all sets of instance variables used by each method. The larger the number of similar methods, the more cohesive the class. Provides a measure of the relative disparate nature of methods in the class.

In [25], they propose a complexity metric based on interactions between components in a CBS (component based system). Their metric is applicable for black-box components because they only require the interfaces. They define complexity for a component as the number of incoming interactions plus the number of outgoing interactions. A component ”A” has a outgoing interaction to component ”B” if ”A” depends on ”B”. A component ”A” has a incoming interaction from component ”B” if ”B” depends on ”A”. The information regarding this interactions is obtained from the interfaces. The metric results shows a correlation between the complexity and the number of interactions, for a given component, i.e., the higher the number of interactions, the higher the complexity. They also propose that the average number of interactions per component should not be greater than 5.

In [26], they study the relationship between software complexity and maintenance costs in order to understand the factors responsible for software complexity and why maintenance costs increase with software complexity. They provide a background on software complexity and its causes, the factors that affect software maintenance costs and cost estimation for software maintenance. They used data from Windows, Debian and Linux to present their evaluation. The results show that as the size of the software (lines of code) increases, the software becomes more complex, and hence the cost of maintaining such software increases.

In [27], they propose an approach for service interfaces decomposition with the help of cohesion metrics. They only require the specifications of the service interfaces, therefore not needing access to the source code. They created a set of cohesion metrics that use the information from service interfaces and progressively decomposes a given interface into more cohesive interfaces. The metrics they propose are divided in three main groups: message, conversation and domain. Message-level cohesion assumes that two operations are related if their input (respectively, output) messages are similar. The notion of Conversation-level cohesion assumes that an operation is related with another if the former's input (respectively output) message is similar with the latter's output (respectively input) message. In Domain-level cohesion the names of the operations that are provided by a service reflect what these operations do. More specifically, the names of the operations comprise terms that correspond to certain actions (e.g., set, get) and terms that correspond to concepts of the domain that is targeted by the service (e.g., queue, attribute, message). Based on this intuition, two operations are considered as being

related if their names share domain-level terms. They validated their approach in 22 real-world services, concluding that the proposed approach is able to improve cohesion, and the number of interfaces produced by the approach linearly increases with the size of the decomposed interface.

In [22], they propose nine coupling metrics, where eight use static analysis and one uses dynamic analysis, for the purpose of predicting the quality attribute of maintainability. The metrics they propose, which cover all structural coupling relationships from a formal model of service-oriented design, were validated against the property-based software engineering measurement framework [28]. The metrics are described as follows, where a system implementation element represents a object-oriented class or interface, and a service is a group of system elements:

1. Weighted Intra-Service Coupling between Elements - is the weighted count of the number of other implementation elements of the same service to which it is coupled via incoming or outgoing relationships.

2. Weighted Extra-Service Incoming Coupling of an Element - is the weighted count of the number of system elements not belonging to the same service that couple to this element.

3. Weighted Extra-Service Outgoing Coupling of an Element - is the weighted count of the number of system elements not belonging to the same service that are used by this element.

4. Extra-Service Incoming Coupling of Service Interface - is a count of the number of system elements not belonging to the same service that couple to this service through its service interface.

5. Element to Extra Service Interface Outgoing Coupling - is a count of the number of external service interfaces that are used by this element. Multiple couples to a same interface are only counted once.

6. Service Interface to Intra Element Coupling - is a count of the direct relationships between its interface and its implementation elements.

7. System Partitioning Factor - measures the degree of partitioning of this system into services. More specifically it is the ratio of total elements in the system belonging to at least one service to the total number of the elements in the system.

8. System Purity Factor - measures the degree of purity of this system in terms of implementation elements belonging to one and only one service.

9. Response for Operation - is the cardinality of the sets of implementation elements and other service interfaces that can be executed in response to invocations of the operation with all possible parameters.

3

Solution Architecture

Contents

Considering the impact associated with the relaxing of the transactional behavior in monolith functionalities when they are migrated to a microservices architecture, we intend to estimate what is the cost of the migration, such that the architect can take informed decisions when analyzing candidate decompositions. Additionally, we intend to experiment with similarity measures to study which aspects of microservices, and microservices interactions, may impact on the cost of the implementation of the microservice architecture, like the difference between read and write accesses, and the sequence of invocations between microservices, that can be used to generate decompositions with different costs. Therefore, we expect that when applying these similarity measures to the clustering algorithm, we generate different candidate decompositons, which can be analyzed in terms of their quality.

A decomposition has an impact on the monolith functionalities, due to the relaxing of consistency, and the level of impact varies depending on the type of access. For instance, when a functionality writes an entity and this entity is read in the context of other functionalities, which in this decomposition has to be implemented as distributed transactions, it impacts on the business logic of these functionalities, and the cost of migrating this functionality increases with the number of other functionalities that read the entity because each one of them may need to be redefined to cope with a weaker model of consistency. Note that the functionalities that read the new written data need to consider the possibility that it can be outdated, and have to change their business logic accordingly. On the other hand, when a functionality reads an entity that is written by other functionalities, that have to be implemented as distributed transactions in this decomposition, it may impact on the business logic of the functionalities that write the entity, and, analogously to the previous case, this impact increases with the number of transactions that write the entity. In this case, the functionalities that write need to consider what level of consistency they want to provide.

Note that the cost associated with the impact of reads and writes can have different types of implementation. For instance, one solution would be to replicate the domain entity, where the cost corresponds to the implementation of the replication and the synchronization between replicas, which has an implementation cost associated with the number of replicas and microservices involved, as well as operation costs, which correspond to the frequency of the synchronization and the impact on the behavior perceived by the end-user in terms of data consistency. Another solution would be to implement explicit invocations between the microservices whenever the information is required, which also has implementation and operation costs, for example, due to the handling of failures in the distributed transaction.

Beyond the type of access, we also consider the sequence of 'hops' between the microservices that implement a functionality. At first sight, we can consider that a higher number of 'hops' the more complex the implementation of the microservice is. However, this may be tuned by the type of accesses, if, for example, the implementation of functionality accesses two microservices alternately several times and one of those microservices is always accessed to read the same domain entity.

In this section we describe our strategy to the generation of candidate microservice decompositions and our developed metrics to assess cost of migration to the decomposition. We start by explaining how we gather information from the monolith, and then we present our complexity metric. After that we explain what are the different similarity measures we used to group entities, and in the end we present our visualization tool which, given a decomposition, shows the above metrics, and serve as a proof of concept.

Our workflow to create microservices leverages on the process presented in section 1.1.

We start by collecting data from the monolith system using static analysis, where we, instead of collecting only the accesses, gather the reads and writes made to domain entities, and the sequence of those accesses. We also improve the accuracy of the data collected, by capturing method calls made inside Java streams.

In the clustering part we use the hierarchichal algorithm previously used, which requires as input a similarity matrix of entities and return as output a group of clusters, where each cluster is a group of entities. Each entry in the similarity matrix represents a value of similarity between two entities and is calculated using a combination of the new similarity measures that consider the type of access and the sequence of accesses.

In the visualization section we present different views depending on what we are analysing (decomposition, functionality or entities). We also show the complexity metrics and recalculate them as we change the decomposition. It is also possible to compare any two decompositions, where the main interest is comparing our generated decomposition with an expert one.

3.1 Data Collection

An Eclipse plugin was developed to introspect a project in the workspace using Eclipse[1]. The main goal of this plugin is to find all method call chains starting in methods of a controller. In our approach we analyse projects that use the Model View Controller architecture, and so, every method in a controller, that serves a certain endpoint, is considered a functionality. The information that is obtained from the method call chain (or call graph) is the accesses to classes of the Fénix Framework. Fénix Framework provides a transactional and persistent domain model. This way we can consider that an access to a Fénix Framework class corresponds to an access to persistent data, which is relevant with our approach of defining microservices based on the transactional context because the set of accesses to persistent entities in the context of the invocation of a controller method, occurs as an ACID transaction.

As a result, we obtain an intermediate data representation where we have, for each controller method, the sequence of domain classes accessed, and the type of each access, either a read or a

[1] vogella.com/tutorials/EclipseJDT/article.html

write.

The implementation consists on the following steps:

1. Iterate through all controller classes in the Eclipse project, and for each method in a controller that exposes an endpoint, obtain all method calls inside it.

2. Get all subsequent calls recursively, in a DFS, until we reach a Fénix Framework method call. If during this sequence of calls we reach a method that was already seen (cyclic method chain) we stop the recursion. We can do this because we keep the method stack.

3. When we reach a Fénix Framework method call, we add two accesses to the sequence of the current controller method being analysed: first, the name of the domain class called, and second, the return type, or argument type in case of a write, of the method (only considered if the type corresponds to a domain entity). Each entity accessed is associated with the type of access, in the form of a pair, which is given by the name of the method called (read if the method start with 'get', and write if the method start with 'set','add' or 'remove'). This can be assured because all Fénix Framework persistent data accesses are automatically generated and all follow this name pattern.

   As an example:

   company.getPerson() - corresponds to a read of Company and then Person

   ((Company,r),(Person,r))

   person.setCompany(company) - corresponds to a write of Person and then Company

   ((Person,w),(Company,w))

4. There is also the exception case of the calls to the 'getDomainObject' method. This is a special method from the Fénix Framework that receives an id and returns a domain object with that id. In this case, we are able to resolve the type binding of the object returned (most cases use a cast expression) and save this information, as a read of a domain entity.

3.2 Microservices Architectural Metrics

Before we introduce the specification of our metrics, we need to formally define a set of variables and operations. This will help to understand and better explain how the metrics are calculated.

- D represents the set of monolith decompositions, and by convention decompositions of D are represented using the lowercase letter d

- F represents the set of monolith functionalities, and by convention functionalities of F are represented using the lowercase letter f

- E represents the set of monolith entities (domain classes), and by convention entities of E are represented using the lowercase letter e
- M represents the access modes, which has two values, r for read and w for write, and by convention modes of M are represented using the lowercase letter m
- S represents the set of sequences, where the elements of a sequence are pairs (e, m), and by convention sequences of S are represented using the lowercase letter s
- $d.clusters$ represents the clusters of decomposition d
- $c.entities$ represents the entities of cluster c
- $s.entities$ represents the entities accessed in sequence s
- $f.sequence$ represents the sequence of accesses done by functionality f

We also need to define a set of functions that will facilitate the writing of the metrics.

- Given an entity e, we can obtain all the functionalities that access e.

$$functionalities(e) = \{f \in F : e \in f.sequence.entities\} \tag{3.1}$$

- Given an entity e and an access mode m, we can obtain the functionalities that access e according to that access mode.

$$functionalities(e, m) = \{f \in F : (e, m) \in f.sequence\} \tag{3.2}$$

- Given a cluster c, we can obtain the functionalities that access the cluster.

$$functionalities(c) = \cup_{e \in c.entities} functionalities(e) \tag{3.3}$$

- Given a sequence s and a decomposition d, we can obtain the clusters accessed in the context of that sequence.

$$clusters(s, d) = \{c \in d.clusters : s.entities \cap c.entities \neq \emptyset\} \tag{3.4}$$

- Given a sequence s and two accesses $(e1, m1)$ and $(e2, m2)$ of the sequence, we can know which occurs first in the sequence.

$$(e_1, m_1) <_s (e_2, m_2) = \begin{cases} true, & \text{if } (e_1, m_1) \text{ occurs before } (e_2, m_2) \text{ in } s \\ false, & \text{otherwise} \end{cases} \tag{3.5}$$

- Given a functionality f and a decomposition d, f is a distributed functionality in d if it accesses more than one cluster.

$$distributed(f, d) = \#clusters(f.sequence, d) > 1 \tag{3.6}$$

Lastly, we define the sequence of a functionality f in a decomposition d, denoted by $sequence(f, d)$. It is represented by a sequence of sequences, where the sequence represents the order by which the clusters are accessed according to the functionality, and each subsequence contains the relevant entity accesses that occur inside the cluster.

Formally, $sequence(f, d)$ can be defined as:

$$sequence(f, d) = split(f.sequence, d).map(s \Rightarrow remove(s)) \tag{3.7}$$

The $sequence(f, d)$ is formed by applying two operations: $split$ and $remove$, as observed in equation 3.7. Note that in the map function, s represents a subsequence of the original sequence $f.sequence$, such that all entities in s belong to the same cluster.

The $split$ operation is formally defined below. The result of this operation is a sequence of sequences, where each subsequence is denoted by s_i. The first condition assures that all entities in a subsequence s_i belong to the same cluster. The second condition assures that two consecutive subsequences are from different clusters.

The third condition assures that the original sequence can be obtained form the concatenization of the subsequences, such that no pairs are lost.

$$\begin{aligned} &split(s, d) = concat_{i=1..k}(s_i)\ such\ that \\ &\qquad \forall_{i=1..k} \exists c \in d.clusters : s_i.entities \subseteq c.entities \\ &\qquad \nexists_{i=1..k-1} : clusters(s_i, d) = clusters(s_{i+1}, d) \\ &\qquad flat(concat_{i=1..k}(s_i)) = s \end{aligned} \tag{3.8}$$

After the $split$ operation is performed, we apply a $remove$ operation to each subsequence s_i. This operation intent is to remove the entity pairs that are not relevant to calculate the cost. Entity pairs are removed according to the following rules:

- If an entity is read, all subsequent reads of that entity are removed
- If an entity is written, all subsequent accesses of that entity are removed

To have an accurate metric, we need to ignore repeated reads and writes that occur inside a microservice, because each microservice executes an atomic transaction. Therefore, in the context of the

execution of a functionality inside a microservice only the first-read, first-write, and first-read first-write are relevant. In the first case, where we have one or more reads of the entity and no writes, only the first one has a synchronization cost, because all subsequent reads use the same value. In the second case, there is at least one write and zero or more reads, but reads occur after the first write, and in this case the value read is written in the context of the transaction but the changes introduced by the first write should be visible to the other microservices. Finally, the last case is a combination of the the two previous cases, where there are several reads and several writes, and the first write is preceded by one or more reads, and so it is necessary to add the cost of the initial synchronization together with the cost of the entity state change introduced by the last write. Note that it is the last write, though it is only necessary to capture one write, for each entity, in the sequence to calculate the cost.

The $remove$ operation is formally described below:

$$
\begin{aligned}
&remove(s) = s_r \; belonging \; to \; sequence \; S \; such \; that \\
&\quad \forall_{(e,m) \in s_r} (e,m) \in s \\
&\quad \nexists (e',m') \neq (e,m) \in s_r : e' = e \wedge m' = m \\
&\quad (e,r) \in s_r \wedge (e,w) \in s_r \Rightarrow (e,r) <_s (e,w) \\
&\quad \forall_{(e,m),(e',m') \in s_r} (e,m) <_{s_r} (e',m') \Rightarrow (e,m) <_s (e',m')
\end{aligned}
\tag{3.9}
$$

The first condition assures that no new access pairs are introduced. The second condition assures that do not exist two access pairs for the same entity and mode. The third condition assures that if an entity is read and written, the read occurs first in the sequence. The fourth condition assures that we preserve the order from the initial sequence.

Example:

Given $e_1, e_2 \in c_1$,

$e_3 \in c_2$,

$c_1, c_2 \in d$,

$f.sequence =< (e_1,r),(e_2,r),(e_1,r),(e_3,r),(e_1,w),(e_1,r),(e_1,w) >$

$split(f.sequence, d) =<< (e_1,r),(e_2,r),(e_1,r) >,< (e_3,r) >,< (e_1,w),(e_1,r),(e_1,w) >>$

$split(f.sequence, d).map(s => remove(s)) =<< (e_1,r),(e_2,r) >,< (e_3,r) >,< (e_1,w) >>$

3.2.1 Complexity

To answer our first research question, we propose a complexity metric for each functionality, which in the context of the systems we are going to analyse is represented by a Spring-Boot controller, as described in the section 3.1, in the context of a particular decomposition, in order to provide a better insight into

the cost of the decomposition. This metric is concerned with the complexity of the functionalities when migrated to the new microservices architecture, due to the change in the level of consistency of the original monolith domain entities. The information used for this metric is the sequence of entities accessed during the transaction and the type of each access. For each microservice accessed (hop) in the execution of a functionality, we attribute a weight to it based on the entities accessed in that microservice. Therefore, we calculate for each entity the cost of reading or writing it, depending on whether the entity is read or written in the context of the functionality. According to our rational for the complexity of reading, the cost of reading an entity is the percentage of other distributed functionalities that write that entity. On the other hand, the cost of writing an entity is the percentage of other distributed functionalities that read that entity.

Complexity of a Functionality For each cluster accessed (sub) during the execution of a functionality f in a decomposition d, we sum the complexity of accessing each cluster, in the context of the sequence of accessed clusters by $sequence(f, d)$. If there is only one cluster (monolith situation), the complexity of the functionality is 0.

$$complexity(f, d) = \begin{cases} 0, & \text{if } sequence(f, d).length = 1 \\ \sum_{sub \in sequence(f,d)} complexity(f, d, sub), & \text{otherwise} \end{cases} \tag{3.10}$$

The complexity of accessing a cluster in the context of the sequence sub is represented by the size of the union of the complexities of each entity accessed by the functionality in that cluster.

$$complexity(f, d, sub) = \# \cup_{(e,m) \in sub} complexity(f, d, (e, m)) \tag{3.11}$$

The complexity of accessing an entity depends on whether the entity is being read or written.

$$complexity(f, d, (e, m)) = \begin{cases} complexityOfRead(f, d, e), & \text{if } m = r \\ complexityOfWrite(f, d, e), & \text{if } m = w \end{cases} \tag{3.12}$$

The complexity of reading an entity in the context of the functionality, is the set of other distributed functionalities that write it, where a distributed functionality is a functionality that accesses more than one cluster. If more distributed functionalities write the entity then the complexity of reading it will be high, because there will be more business logic to be redesigned, one for each functionality.

$$complexityOfRead(f, d, e) = \{f' \neq f : distributed(f', d) \wedge (e, w) \in flat(sequence(f', d))\} \tag{3.13}$$

On the other hand, the complexity of write is given by the set of other distributed functionalities that read it. Similar to the complexity of reads, the rational also applies to the complexity of writes, because

it measures the amount of business logic that needs to be redesigned due to the decomposition.

$$complexityOfWrite(f, d, e) = \{f' \neq f : distributed(f', d) \wedge (e, r) \in flat(sequence(f', d))\} \quad (3.14)$$

Complexity of a Cluster: Measures the complexity of a cluster c in $d.clusters$ using the complexity of the functionalities it takes part of. Therefore, the complexity of a cluster is the average complexity of the functionalities it participates.

$$complexity(c, d) = \frac{\sum_{f \in functionalities(c)} complexity(f, d)}{\#functionalities(c)} \quad (3.15)$$

Complexity of a Decomposition: Measures the complexity of a decomposition (d) as the average complexity of its functionalities.

$$complexity(d) = \frac{\sum_{f \in F} complexity(f, d)}{\#F} \quad (3.16)$$

3.3 Similarity Measures

A similarity measure represents the distance between two entities. It provides information about how coupled the entities are. In our case, a high value of similarity (value=1) means that two entities are highly correlated and therefore should be in the same microservice, while a value close to 0 means that the entities should be in different microservices. In our approach, this information is given as input to a hierarchichal clustering algorithm in the form of a similarity matrix, where each entry (x, y) represents the distance between two entities ($entities[x], entities[y]$). The similarity values are normalized (between 0 and 1).

Regarding our second research question, we propose the experimentation with four similarity measures, each based on a different type of information collected from the monolith. This measures represent the input parameters that are given to the clustering algorithm to generate candidate monolith decompositions. We intend to combine the four similarity measures and evaluate what are the weights for each element of the combination that generate a decomposition with the lower complexity cost.

Using definitions from section 3.2, we formally define the similarity measures below, for two entities, e_1 and e_2:

Access:

$$accessMeasure(e_1, e_2) = \frac{\#\{functionalities(e_1) \cap functionalities(e_2)\}}{\#functionalities(e_1)} \quad (3.17)$$

In this measure we attribute a similarity value to each pair of entities based on indistinct accesses. The

value of the similarity is the number of functionalities they have in common (that access both), divided by the functionalities that access the first one.

Read:

$$readMeasure(e_1, e_2) = \frac{\#\{functionalities(e_1, r) \cap functionalities(e_2, r)\}}{\#functionalities(e_1, r)} \tag{3.18}$$

In this measure, the value of the similarity is the number of functionalities that read both entities divided by the functionalities that read the first one.

Write:

$$writeMeasure(e_1, e_2) = \frac{\#\{functionalities(e_1, w) \cap functionalities(e_2, w)\}}{\#functionalities(e_1, w)} \tag{3.19}$$

In this measure, the value of the similarity is the number of functionalities that write both entities divided by the functionalities that write the first one.

Sequence:

$$sequenceMeasure(e_1, e_2) = \frac{\sum_{f \in F} countSequence(e_1, e_2, f)}{maxNumberOfConsecutives} \tag{3.20}$$

$$\begin{aligned} countSequence(e_1, e_2, f) = \#\{(i, i+1) : 0 < i < f.sequence.length - 1 \wedge \\ ((f.sequence[i].entity = e_1 \ \wedge \ f.sequence[i+1].entity = e_2) \vee \\ (f.sequence[i].entity = e_2 \ \wedge \ f.sequence[i+1].entity = e_1))\} \end{aligned} \tag{3.21}$$

In this measure, the value of the similarity is the number of times that e_1 followed by e_2 or the contrary appears in the functionality f sequence of accesses. To normalize all values, we divide each one with the biggest numerator, represented by $maxNumberOfConsecutives$. Unlike the previous measures, this one is symmetrical, meaning that the similarity between e_1 and e_2 is equal to e_2 and e_1.

3.4 Visualization and Modeling Tool

A tool was developed as a proof of concept, in the form of a web application, implementing the responsibilities for decomposition generation and analysis. The backend was developed with Spring-Boot and the frontend with ReactJS. The tool requires as input a data collection file that represents the codebase of the monolith. This file, which is obtained from the result of the data collection phase described in section 3.1, is an intermediate data representation and contains internal information about the invocation tree inside the system. Note that, each Spring-Boot controller corresponds to a functionality. After a codebase is created, we are able to perform three operations to it.

First, we are able to manipulate controller profiles, which means that we can choose which con-

trollers are being considered when calculating the similarity measures and creating the decomposition. For example, we can exclude setup and/or teardown controllers that may have a high impact on the decomposition but are only invoked during setup and shutdown of the system, and in that situation they can be invoked in sequence.

Second, we are able to generate a dendrogram that represents all the possible decompositions given a specific set of similarity measures. From a cut in the dendrogram results a decomposition. There are a number of parameters needed to create a dendrogram, which are the controller profiles that will be used, the linkage type for the hierarchichal algorithm, and the weight percentage for each of the four similarity measures available (access, read, write, sequence) which sum should be 100%.

The third operation is the creation of expert cuts. An expert cut is a decomposition that is created manually by the system owner/developer and represent what the expert intends to be the ideal decomposition of the system. This decomposition is used in the external evaluation to assess the quality of the tool-generated decomposition.

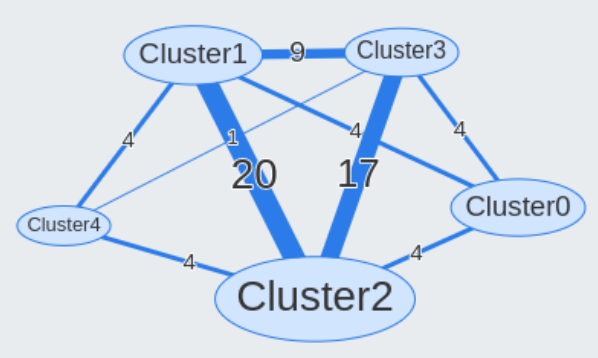


Figure 3.1: Cluster view presenting clusters and the relations between them, for a generated decomposition

After performing a cut in the dendrogram, a decomposition is shown, as seen in figure 3.1, in the form of a graph where each node is a microservice, and the edges show the functionalities that access both adjacent nodes. By clicking or hovering the nodes we are able to see what entities are inside the microservice.

Besides this view with the decomposition (Clusters View), there is also the Transaction View and the Entity View.

In the Transaction View, shown in figure 3.2, we can choose a functionality and observe the clusters it accesses and its complexity. By clicking or hovering the controller node we are able to see the entities that it accesses, the cluster nodes show the entities inside the cluster, and the edges show the entities the controller accesses inside that specific cluster. There is also the possibility to see the sequence of accesses between the clusters, either in the form of a graph or a table.

In the Entity View, shown in figure 3.3 we can observe what are the clusters that take part in the functionalities the entity also participates. Clicking the entity shows the functionalities that access it,

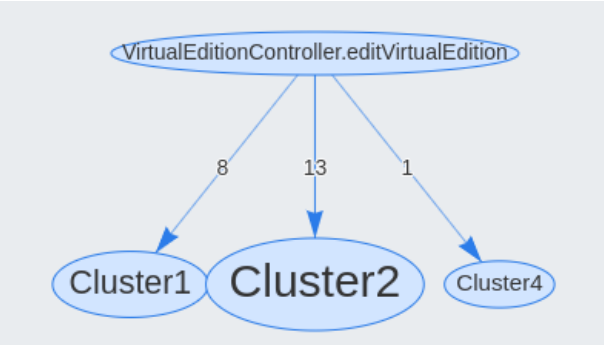


Figure 3.2: Transaction View with a functionality and the clusters it accesses

each cluster node show the entities inside it, and the edges show the functionalities that access both the cluster and the entity. Note that the cluster in which the entity is contained is not presented in the diagram, because we are interested in analysing the distributed functionalities the entity is involved in.

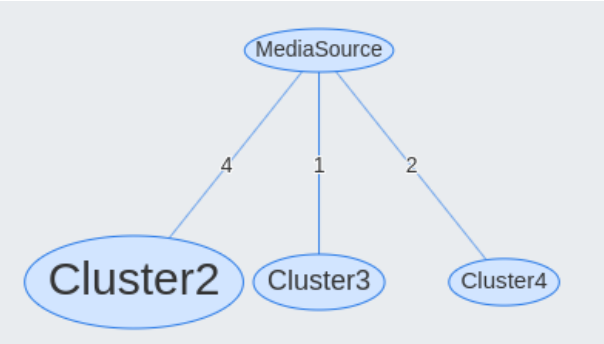


Figure 3.3: Entity View with an entity and the clusters that participate in the same functionalities

Regarding the metrics, we can see the complexity, cohesion and coupling metrics in the Clusters View for each cluster and the complexity in the Transaction View for each functionality. Additionally, we also have the complexity, cohesion and coupling values for the decompositions.

The tool also allows to compare any two decompositions, which is particularly helpful when we compare our generated decomposition with one created by an expert. The comparison is made using pairwise comparison, meaning that, given two entities, we compare if they are or not in the same cluster, for both decompositions.

This tool is essentially a visualization tool, but it also provides some modeling capabilities. The main ones are the manipulation of a candidate decomposition and the real-time recalculation of the metrics after each change is done to the model. In relation to the manipulation capabilities, shown in figure 3.4, there are four main operations to manipulate the graph: rename, merge, split and transfer.

We also developed an analyser that iterates through all the possible combination of similarity measures. As a result we obtain all possible decompositions and are able to find the similarity measures

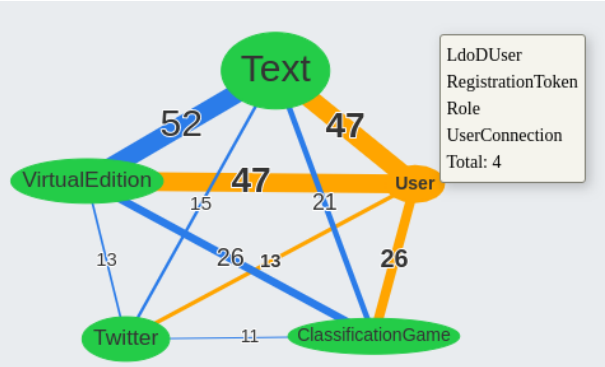


Figure 3.4: Manipulated decomposition

that provide better results, either in terms of comparing with the complexity metric or when compared with an expert decompositon, which allows to compare the qualities of the results of applying different similarity measures.

4

Evaluation

Contents

To validate our approach we used three monolith systems, LdoD[1] (122 controllers, 67 domain entities), Blended Workflow[2] (98 controllers, 49 domain entities) and FenixEdu Academic[3] (856 controllers, 451 domain entities).

All applications are layered and client server web applications, where Spring-Boot controllers implement services by accessing a rich object domain model implemented using Fénix Framework.

4.1 Complexity Metric

Answering our first research question, we evaluate our complexity metric through the analysis of its correlation to a cohesion and a coupling metric. These two metrics are similar to those encountered in other research such as [27] [22] [23], and therefore provide an evaluation of our metric against the research state of the art.

This set of validation metrics were created to answer the question: Does a decomposition that minimizes the transactional complexity also has good results in terms of modularity?

Cohesion: Cohesion refers to the internal grouping of entities, and the reasoning behind it can be understood as the following: everytime an entity is accessed in a functionality, all the other entities from the same microservice should also be acessed during the course of the functionality, showing that those entities implement the same responsibility and should be in the same microservice.

Cohesion of a Cluster: We measure the cohesion of a cluster c as the average percentage of entities accessed for each functionality that accesses the cluster. A cluster has maximum cohesion (cohesion = 1) if every functionality that accesses it does in fact access every entity inside the cluster.

$$cohesion(c) = \frac{\sum_{f \in functionalities(c)} \frac{\#\{e \in c.entities : e \in f.sequence.entities\}}{\#c.entities}}{\#functionalities(c)} \tag{4.1}$$

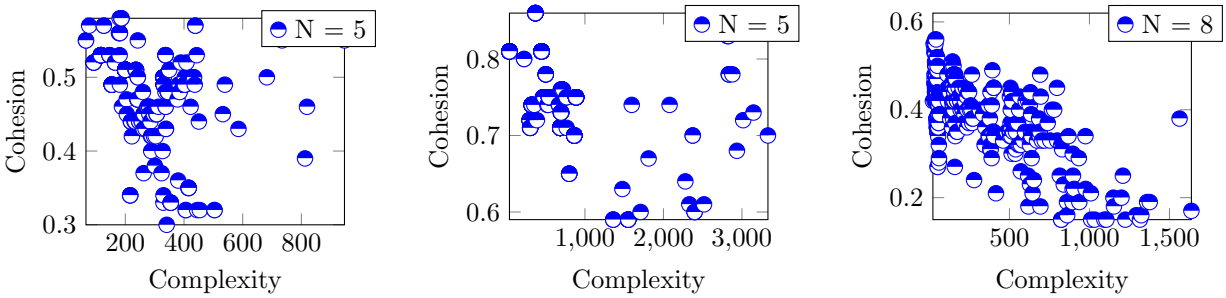
Our cohesion metric is related with the Lack of Cohesion in Methods metric presented in [23], where the methods represent our functionalities, and the instance variables our entities.

Cohesion of a Decomposition: Measures the cohesion of a decomposition d as the average cohesion of its clusters.

$$cohesion(d) = \frac{\sum_{c \in d.clusters} cohesion(c)}{\#d.clusters} \tag{4.2}$$

Figure 4.1 shows the correlation between the complexity and cohesion values for the three systems, where each dot corresponds to a decomposition generated for a particular combination of similarity measures in a total of 286 combinations. All the combinations of values have an interval which is a multiple of 10% and the same number of clusters, 5, for LdoD and Blended Workdflow, and 8 for

[1] github.com/socialsoftware/edition
[2] github.com/socialsoftware/blended-workflow
[3] github.com/FenixEdu/fenixedu-academic



(a) LdoD **(b)** Blended Workflow **(c)** FenixEdu Academic

Figure 4.1: Correlation between the Complexity and Cohesion metric, for the three monolith systems

FenixEdu Academic. From these figures we can denote a slight relation between the values, specially in FenixEdu Academic, where a higher complexity is associated with a lower cohesion. On the other hand, we can conclude, from the analysis of the graphs, that low complexity seems to imply high cohesion but that the inverse is not necessarily true, because there are some decompositions with high complexity and high cohesion, even though they may be outliers. This is not unexpected because the complexity of migrating the functionalities is not directly correlated with the cohesion, a functionality may not need to access all entities of a cluster to have low complexity, for instance if the cluster has a large number of entities. Note that in the case of the monolith, which has complexity 0 and the cohesion is low, provides the insight that the study of the possible correlation between cohesion and complexity should be done in a context where there are several clusters. Overall, we can conclude that for two decompositions with the same complexity the value of cohesion may be used to choose the one that follows the single responsibility principle.

Coupling: Coupling refers to the external distance between groups of entities and it can be used to determine how isolated a microservice is.

Coupling of a Cluster: Measures the level of dependency between clusters. If a cluster c has a high value of coupling, it means that it accesses many entities from other clusters. It is defined as the average level of coupling to the other clusters. Our metric is only applicable if the decomposition has more than one cluster because if there is only one cluster (monolith situation) the coupling value is irrelevant.

$$coupling(c, d) = \frac{\sum_{c' \in d.clusters\ ,\ c' \neq c} coupling(c, c', d)}{d.clusters - 1} \tag{4.3}$$

The level of coupling between any two clusters c_1 and c_2 is the number of entities of c_2 accessed by c_1. Note that we define accesses between clusters from the sequence of a functionality, i.e., if a functionality accesses an entity from c_1 and then an entity from c_2, we consider that this is an access

from c_1 to c_2 (a hop between the clusters).

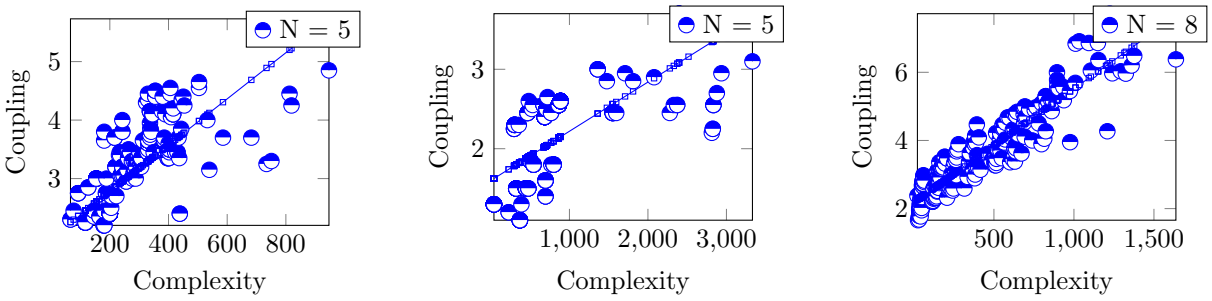
$$coupling(c_1, c_2, d) = \#\cup_{f \in F} clusterDependencies(c_1, c_2, d, f) \tag{4.4}$$

$$\begin{aligned} clusterDependencies(c_1, c_2, d, f) = \{e \in c_2.entities : \exists_{e' \in c_1.entities\ ,\ 0<i<sequence(f,d).length-1} : \\ sequence(f, d)[i].last.entity = e' \wedge \\ sequence(f, d)[i+1].first.entity = e\} \end{aligned} \tag{4.5}$$

Our coupling metric is very similar to the Weighted Extra-Service Outgoing Coupling of an Element metric showed in [22], defined as the weighted count of the number of system elements not belonging to the same service that are used by this element, where instead of implementation elements we consider clusters. It is also similar to the Coupling Between Object Classes metric in [23], defined as the count of the number of other classes to which it is coupled. This relates to the notion that an object is coupled to another object if one of them acts on the other, i.e., methods of one use methods or instance variables of another. This can be correlated to our metric where a method corresponds to a cluster, and an instance variable to an entity.

Coupling of a Decomposition: Measures the coupling of a decomposition d as the average coupling of its clusters.

$$coupling(d) = \frac{\sum_{c \in d.clusters} coupling(c, d)}{\#d.clusters} \tag{4.6}$$



(a) LdoD **(b)** Blended Workflow **(c)** FenixEdu Academic

Figure 4.2: Correlation between the Complexity and Coupling metric, for the three monolith systems

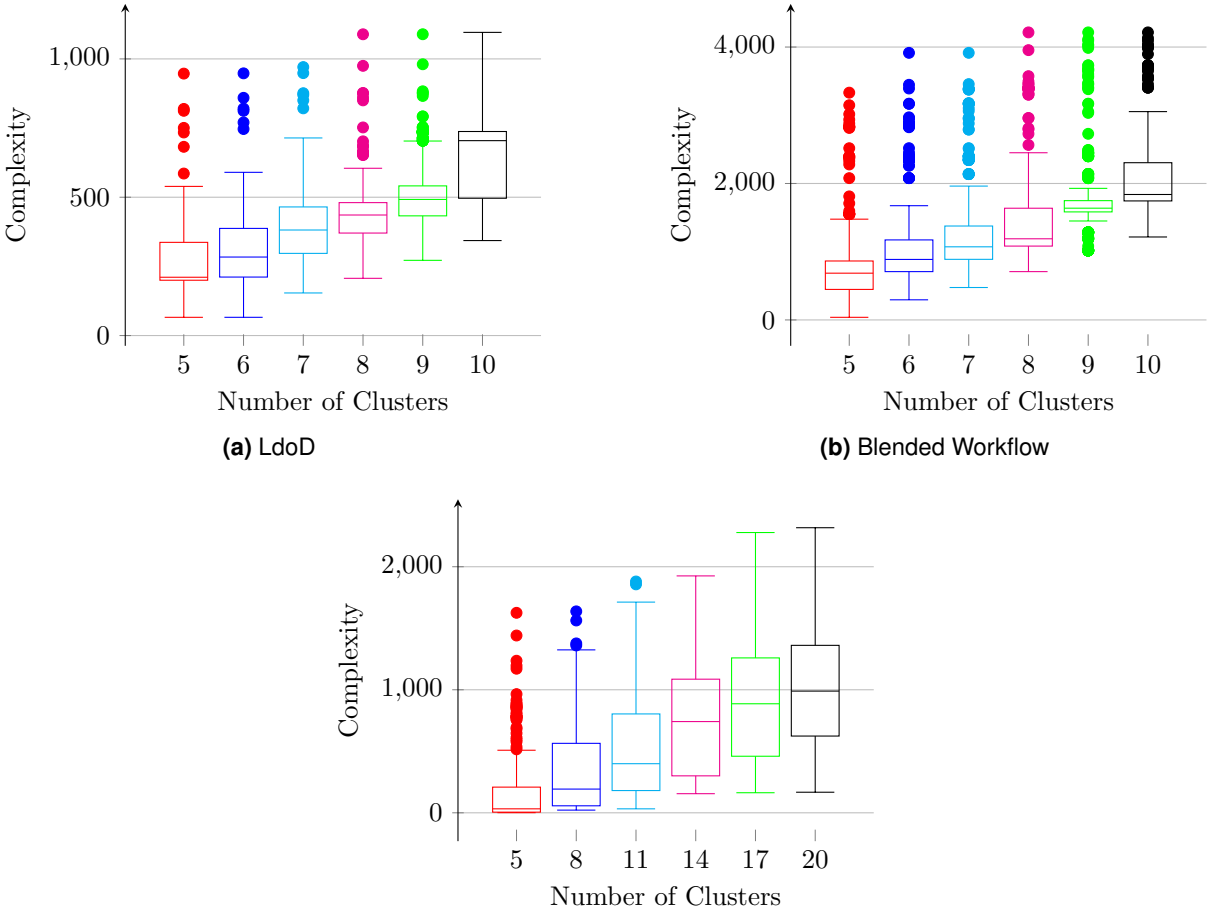
Figure 4.2 show the correlation between the complexity and the coupling metrics for the three systems, where each point is a decomposition generated by a combination of similarity measures, composed of 5 clusters for LdoD and Blended Workflow, and 8 clusters in FenixEdu Academic. The figures also contain a linear regression line to better illustrate the correlation. In this figures we can clearly establish a relation between the metrics, which is the higher the complexity the higher the coupling. Decompositions with a high complexity tend to have a high number of distributed transactions for each

functionality and this leads to more interactions between clusters, which is what the coupling criteria measures.

From these two comparison (cohesion and coupling) we can conclude that our complexity metric behaves as expected when compared with other metrics.

4.2 Decomposition Complexity

Answering our second research question, we evaluate our similarity measures against the complexity obtained from each combination.



(a) LdoD

(b) Blended Workflow

(c) FenixEdu Academic

Figure 4.3: Complexity of Decompositions, for the three monolith systems

Figure 4.3 shows the complexities of all decompositions generated for the 286 combinations of similarity measures and different number of clusters for the three monolith systems, from 5 to 10 in LdoD and Blended Workflow, and 5 to 20 with intervals of 3 for FenixEdu Academic. Each point represents a decomposition created by a specific combination of the similarity measures. The box in each graphic

contains half of the points (two quartiles) and the line in the middle represents the median. Outside the box there's also the upper and lower bound, and the outlier points. From this figures we can conclude that the complexity metric grows as the number of clusters increases, which is understandable because our metric takes into account the size of the clusters access sequence by functionalities, and when the number of clusters increases the implementation of functionalities gets split into more clusters. We also observe, from the size of the boxes, that many combination of similarity values lead to very similar values of complexity, which indicates that there isn't a unique solution to the values of similarity, that leads to the best decomposition in terms of our complexity metric.

	Similarity Measures			
Complexity	Access	Write	Read	Sequence
946.87	0	0	0	100
819.22	10	0	10	80
812.4	0	0	10	90
750.7	0	10	0	90
734.06	10	0	0	90
682.25	20	0	0	80
585.61	0	10	10	80

Table 4.1: Decomposition Complexity Outliers, N = 5, LdoD

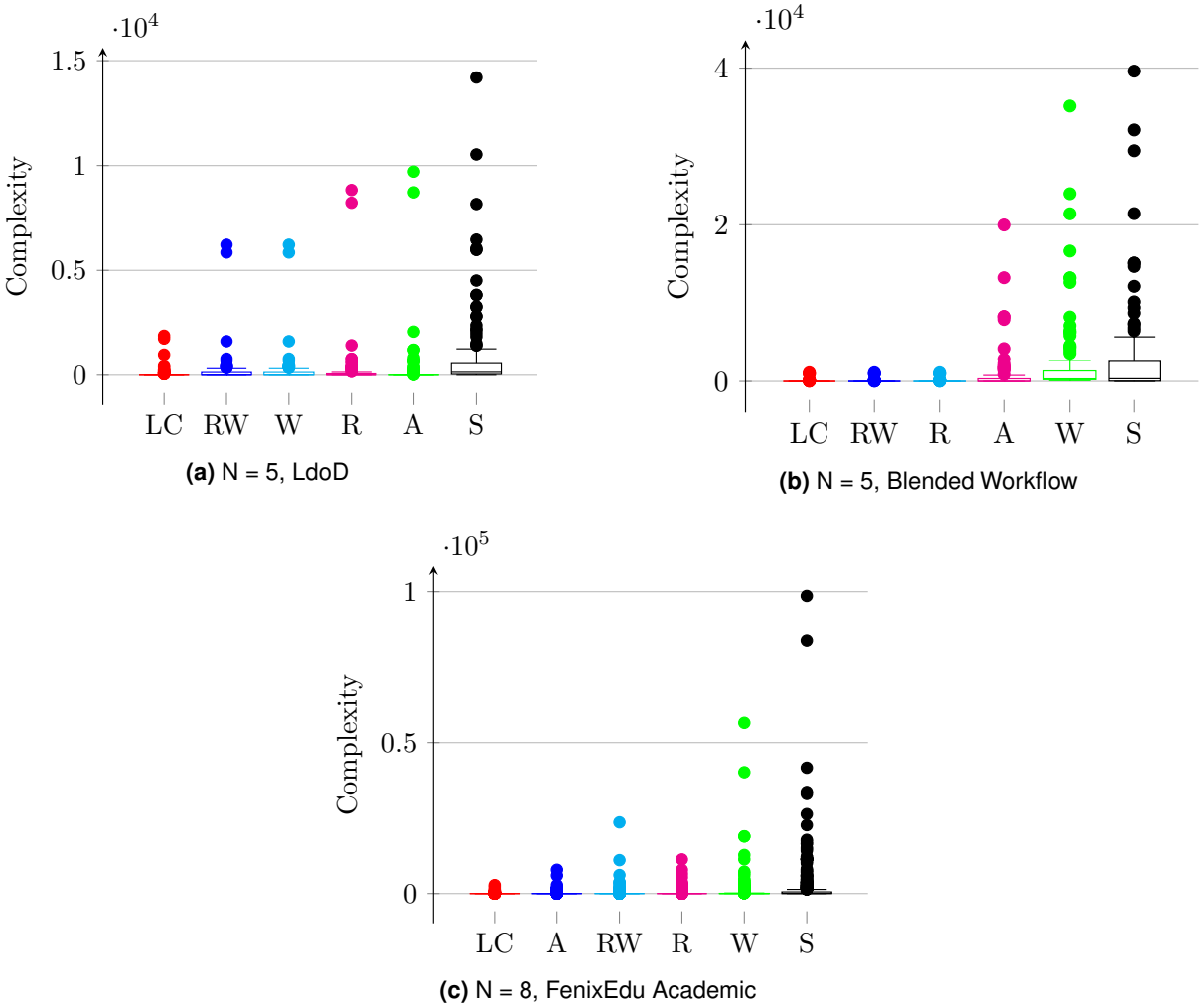
Table 4.1 shows the similarity combinations that led to the highest values of complexity, represented in figure 4.3a as the outlier points of the first plot, where the number of clusters is 5. We can immediately observe that the outliers share a high weight of the sequence measure, which indicate that the application of this similarity measure doesn't provide good results when used alone.

4.3 Functionality Complexity

For a certain decomposition, we analyse the disparity between the complexity of its functionalities. This information is particularly helpful to understand what are the most complex functionalities in the decomposition.

Figure 4.4 shows the distribution of the complexities of the functionalities, for some representative combinations of similarity measures, for the three monolith systems. The plots are ordered left to right, from lowest complexity to highest. The decompositions represented in each figure are the following:

- Lowest Complexity (LC) is the decomposition with the lowest value for the complexity metric, for any combination of the similarity measures
- Combination of Reads and Writes (RW) is the decomposition with the lowest value for the complexity metric, for any combination of the read and write similarity measures, while the other measures (access and sequence) are fixed at 0%



(a) N = 5, LdoD

(b) N = 5, Blended Workflow

(c) N = 8, FenixEdu Academic

Figure 4.4: Complexity of Functionalities for a set of particular decompositions, for the three monolith systems

- Only Writes (W) is the decomposition generated using only the write similarity measure
- Only Reads (R) is the decomposition generated using only the read similarity measure
- Only Accesses (A) is the decomposition generated using only the access similarity measure
- Only Sequence (S) is the decomposition generated using only the sequence similarity measure

We can observe that the majority of the functionalities have a similar value of complexity, with only a small amount of outliers increasing the overall complexity.

Table 4.2 shows the functionalities that have the higher value of complexity, for the LC decomposition, with 5 clusters in LdoD. We can observe that the first two functionalities are detached from the rest and they correspond to functionalities that are executed when booting the system, which can, actually, execute in a non-concurrent context. Therefore, our tool provides a feature that allow the definition of profiles of functionality, such that it is possible to exclude some functionalites which, for instance,

Complexity	Functionality
1886	AdminController.loadTEIFragmentsStepByStep
1753	AdminController.loadTEIFragmentsAtOnce
986	AdminController.generateCitations
422	SignupController.signup
370	VirtualEditionController.editVirtualEdition

Table 4.2: Functionality Complexity Outliers, for the Lowest Complexity Decomposition, N = 5, LdoD

have high complexity but are considered not to occur frequently, from input to the hierarchical clustering algorithm.

		Similarity Measures			
	Complexity	Access	Write	Read	Sequence
Lowest Complexity	65.89	30	30	40	0
Reads and Writes	199.95	0	50	50	0
Only Writes	199.95	0	100	0	0
Only Reads	219.34	0	0	100	0
Only Accesses	242.53	100	0	0	0
Only Sequence	946.87	0	0	0	100

Table 4.3: Analysis of similarity measures for a set of decompositions, with 5 Clusters, in LdoD

		Similarity Measures			
	Complexity	Access	Write	Read	Sequence
Lowest Complexity	38.32	10	10	70	10
Reads and Writes	38.32	0	20	80	0
Only Reads	38.32	0	0	100	0
Only Accesses	886.82	100	0	0	0
Only Writes	2397.12	0	100	0	0
Only Sequence	3015.85	0	0	0	100

Table 4.4: Analysis of similarity measures for a set of decompositions, with 5 Clusters, in Blended Workflow

Tables 4.3, 4.4 and 4.5 show the complexity metric for a set of combinations of similarity measures, with the number of clusters equal to 5 in LdoD and Blended Workflow and 8 in FenixEdu Academic. From this tables we can conclude that there is no unique combination that leads to the lowest complexity. Using only each measure, we are able to conclude that using only the sequence measure always provided the worst result in terms of complexity, and there isn't any supremacy of the other measures when each one is used alone. The results also show that using a combination of measures provided better results than using only one measure in terms of complexity, and that combination varies from case to case. Therefore, the complexity metric can drive the identification of the best decompositions, instead of trying to find a winner similarity measure.

		Similarity Measures			
	Complexity	Access	Write	Read	Sequence
Lowest Complexity	21.81	40	0	0	60
Only Accesses	57.48	100	0	0	0
Reads and Writes	112.9	0	40	60	0
Only Reads	158.68	0	0	100	0
Only Writes	380.02	0	100	0	0
Only Sequence	1239.65	0	0	0	100

Table 4.5: Analysis of similarity measures for a set of decompositions, with 8 Clusters, in FenixEdu Academic

4.4 Expert Decomposition

As part of the external evaluation, we compare a certain decomposition generated by us, with one created by an expert of the monolith system. Considering the expert decomposition as a source of truth help us to assess the relevance of our approach. For this part we were able to obtain an expert decomposition for the LdoD and Blended Workflow systems.

We do a pairwise comparison between the two decompositions, meaning that, for each pair of monolith entities, we count the following:

- True Positive (tp) if both entities belong to the same cluster in both decompositions (expert and ours)
- True Negative (tn) if the two entities belong to different clusters in both decompositions
- False Positive (fp) if both entities belong to the same cluster in our decomposition, and in different clusters in the expert
- False Negative (fn) if the two entities belong to different clusters in our decomposition, and belong to the same in the expert

With this information we calculate the measures of accuracy, precision, recall, specificity and f-score, defined below, which indicate the level of similarity between the two decompositions.

Accuracy:

$$accuracy = \frac{tp + tn}{tp + fp + fn + tn} \tag{4.7}$$

Precision:

$$precision = \frac{tp}{tp + fp} \tag{4.8}$$

Recall:

$$recall = \frac{tp}{tp + fn} \tag{4.9}$$

Specificity:

$$specificity = \frac{tn}{tn + fp} \tag{4.10}$$

F-score:

$$f - score = 2 * \frac{precision * recall}{precision + recall} \tag{4.11}$$

		Similarity Measures				
	Complexity	Access	Write	Read	Sequence	F-Score
Lowest Complexity	65.89	30	30	40	0	0.47
Reads and Writes	199.95	0	50	50	0	0.53
Only Writes	199.95	0	100	0	0	0.53
Only Reads	219.34	0	0	100	0	0.37
Only Accesses	242.53	100	0	0	0	0.38
Highest F-Score	434.29	20	0	30	50	0.73
Expert	617.35	–	–	–	–	1
Only Sequence	946.87	0	0	0	100	0.53

Table 4.6: Analysis with the expert for a set of decompositions, with 5 Clusters, in LdoD

		Similarity Measures				
	Complexity	Access	Write	Read	Sequence	F-Score
Lowest Complexity	38.32	10	10	70	10	0.36
Reads and Writes	38.32	0	20	80	0	0.36
Only Reads	38.32	0	0	100	0	0.36
Highest F-Score	685.31	0	20	50	30	0.47
Only Accesses	886.82	100	0	0	0	0.4
Expert	2129.61	–	–	–	–	1
Only Writes	2397.12	0	100	0	0	0.37
Only Sequence	3015.85	0	0	0	100	0.42

Table 4.7: Analysis with the expert for a set of decompositions, with 5 Clusters, in Blended Workflow

Tables 4.6 and 4.7 shows the results for the comparison with the expert decomposition, for the LdoD and Blended Workflow systems, for a specific set of decompositions already defined in section 4.3. We can observe that the expert decomposition is not the one with the lowest complexity. We also analysed the other decompositions that had lower complexity than the expert, and concluded that most of them have one large cluster and other smaller clusters when the number of clusters is relatively small. Although this decompositions are not fit for a complete migration to microservices, they provide a crucial help for an incremental migration, because they have one large cluster, which we can consider as the monolith core, and a number of smaller clusters, that indicate the first services to be splitted from the monolith. When we increase the number of clusters generated, the clustering algorithm starts to break the single large cluster meaning that we can also provide a complete migration to microservices if we increase the number of clusters high enough. Increasing the number of clusters is particularly helpful when executing an incremental migration, because we can propose new services as the number of clusters increases. Interestingly, this is the approach recommended by microservices experts on the migration of a monolith to a microservices architecture [5, Chapter 13]:

> Fortunately, there are strategies you can use to escape from monolithic hell without having to rewrite your application from scratch. You incrementally convert your monolith into microservices by developing what's known as a strangler application. The idea of a strangler application comes from strangler vines, which grow in rain forests by enveloping and sometimes killing trees. A strangler application is a new application consisting of microservices that you develop by implementing new functionality as services and extracting services from the monolith. Over time, as the strangler application implements more and more functionality, it shrinks and ultimately kills the monolith. An important benefit of developing a strangler application is that, unlike a big bang rewrite, it delivers value to the business early and often.
>
> Chris Richardson

4.5 Summary

As a result of our evaluation, we summarize our conclusions as follows:

- Our complexity metric has a correct correlation with other metrics of cohesion and coupling defined in other research;
- Regarding the similarity measures, we are able to conclude that the sequence measure provides the worst results in terms of complexity, but there is no single combination that outperforms the other, which is confirmed by the existing research that, frequently using different similarity measures, report good results in their validation;
- The complexity metric correctly identifies what are the most complex functionalities in a decomposition;
- The expert decomposition is not the one with the lowest complexity, because the lower complex decompositions tend to identify small microservices while preserving a large cluster that continues to behave as a monolith. This leads to the idea that the tool can help on an incremental migration to microservices approach which is actually defended by the industry experts.

Going back to the research questions we defined in section 1.3:

1. Is it possible to calculate the cost associated with the migration to a microservices architecture due to the introduction of relaxed consistency into the business behavior?
2. Which similarity measures are more effective in the generation of a candidate microservices decomposition in terms of the cost of migration?

Answering the first research question, we were able to calculate the cost associated with the migration to a microservices architecture, with the introduction of the complexity metric. Answering the second research question, we concluded that there isn't a unique combination of similarity measures that are more effective in the generation of candidate microservices decomposition in terms of the cost of migration.

4.6 Limitations

Currently, there are some limitations of our approach as described below:

- Static analysis is used to collect data from the monolith. This analysis may not be the most accurate when gathering the entities accessed, as for example, dynamic analysis can be, because only during runtime is possible to determine what entities are actually accessed. Therefore, it is an open research question whether by collecting data through dynamic analysis we can generate decompositions with lower complexity;
- Only a narrow spectrum of systems are considered (Spring-Boot applications that use the Fénix Framework). It would be interesting to verify our results on other systems that don't use the same technology.

5

Conclusion

Contents

In this work we discussed the problem of microservice architecture migration and presented a complexity metric to assess the cost of the migration. We identified some of the existing research articles related the subject of microservices migration and metrics in service oriented systems, assessed each one and presented a conclusion of the analysed articles. This topic is still relatively recent and there is new research emerging everyday. To tackle this challenge, we leveraged on the strategy created in the previous thesis, where we use static analysis to obtain information on method calls and a clustering algorithm to group the domain entities based on that information. In our work we enriched the previous one with information about the read and write sets, and the sequence of accesses of each functionality. As a proof of concept we developed a visualization tool with different views for the decomposition, functionalities and entities. In this tool we introduced our metric for complexity, cohesion and coupling. The complexity metric correctly identified the most complex decompositions, and the functionalities of the monolith. Regarding the similarity measures, we were able to conclude that the sequence measure provides the worst results in terms of complexity, but there is no single combination that outperforms the other, which is confirmed by the existing research that, frequently using different similarity measures, report good results in their validation. We were also able to conclude that the expert decomposition is not the one with the lowest complexity, because the lower complex decompositions tend to identify small microservices while preserving a large cluster that continues to behave as a monolith. This lead to the idea that the tool can help on an incremental migration to microservices approach which is actually defended by the industry experts. Part of this research, in what concerns the visualization tool, is already published in the European Conference on Software Architecture (ECSA 2019) [29]. The tool code is publicly available in a GitHub repository[1].

5.1 Future Work

- Explore new similarity measures and microservices architectural metrics. Define metrics for the different qualities associated with a microservices architecture.
- Collect new information on the monolithic system using dynamic analysis. Currently only static analysis is used.
- Analyse monolith applications from code repositories in GitHub. With this we can collect new information, such as commits, etc..
- Generalize the data collection tool to any type of monolith applications. This way we can increase significantly the range of applications and provide a better evaluation.

[1] github.com/socialsoftware/mono2micro

- Extend the tool to provide modeller capabilities. The modeller should guide the architecture through the application of microservices patterns and update information on the impact of the modelling process in the metrics provided. Currently there is only a small part of the tool related to modeling, such as the decomposition graph manipulation and the real-time recalculation of the metrics.

Bibliography

[1] M. Fowler and J. Lewis, "Microservices," 2014. [Online]. Available: http://martinfowler.com/articles/microservices.html

[2] K. Ismail, "7 tech giants embracing microservices," https://www.cmswire.com/information-management/7-tech-giants-embracing-microservices, 8 2018, accessed: 2018-11-29.

[3] L. Nunes, "From a monolithic to a microservices architecture," Master's thesis, Instituto Superior Técnico, 9 2018.

[4] S. Gilbert and N. Lynch, "Perspectives on the cap theorem," *Computer*, vol. 45, no. 2, pp. 30–36, Feb. 2012. [Online]. Available: https://doi.org/10.1109/MC.2011.389

[5] C. Richardson, *Microservices Patterns: With Examples in Java*. Manning Publications, 2019. [Online]. Available: https://books.google.pt/books?id=UeK1swEACAAJ

[6] A. Carrasco, B. v. Bladel, and S. Demeyer, "Migrating towards microservices: Migration and architecture smells," in *Proceedings of the 2Nd International Workshop on Refactoring*, ser. IWoR 2018. New York, NY, USA: ACM, 2018, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/3242163.3242164

[7] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '00. New York, NY, USA: ACM, 2000, pp. 7–. [Online]. Available: http://doi.acm.org/10.1145/343477.343502

[8] 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.

[9] D. Escobar, D. Cárdenas, R. Amarillo, E. Castro, K. Garcés, C. Parra, and R. Casallas, "Towards the understanding and evolution of monolithic applications as microservices," in *2016 XLII Latin American Computing Conference (CLEI)*, Oct 2016, pp. 1–11.

[10] A. Levcovitz, R. Terra, and M. T. Valente, “Towards a technique for extracting microservices from monolithic enterprise systems,” *CoRR*, vol. abs/1605.03175, 2016. [Online]. Available: http://arxiv.org/abs/1605.03175

[11] M. Ahmadvand and A. Ibrahim, “Requirements reconciliation for scalable and secure microservice (de)composition,” in *2016 IEEE 24th International Requirements Engineering Conference Workshops (REW)*, Sep. 2016, pp. 68–73.

[12] L. Baresi, M. Garriga, and A. De Renzis, “Microservices identification through interface analysis,” in *Service-Oriented and Cloud Computing*, F. De Paoli, S. Schulte, and E. Broch Johnsen, Eds. Cham: Springer International Publishing, 2017, pp. 19–33.

[13] 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.

[14] G. Mazlami, J. Cito, and P. Leitner, “Extraction of microservices from monolithic software architectures,” in *2017 IEEE International Conference on Web Services (ICWS)*, June 2017, pp. 524–531.

[15] O. Mustafa and J. Marx Gomez, “Optimizing economics of microservices by planning for granularity level,” 04 2017.

[16] 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)*, April 2017, pp. 11–20.

[17] Z. L. UU, M. Korpershoek, and A. O. VU, “Towards a microservices architecture for clouds.”

[18] M. J. Amiri, “Object-aware identification of microservices,” in *2018 IEEE International Conference on Services Computing (SCC)*, July 2018, pp. 253–256.

[19] R. Nakazawa, T. Ueda, M. Enoki, and H. Horii, “Visualization tool for designing microservices with the monolith-first approach,” in *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, Sep. 2018, pp. 32–42.

[20] A. Geraci, *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*, F. Katki, L. McMonegal, B. Meyer, J. Lane, P. Wilson, J. Radatz, M. Yee, H. Porteous, and F. Springsteel, Eds. Piscataway, NJ, USA: IEEE Press, 1991.

[21] W. P. Stevens, G. J. Myers, and L. L. Constantine, “Structured design,” *IBM Syst. J.*, vol. 13, no. 2, pp. 115–139, Jun. 1974. [Online]. Available: http://dx.doi.org/10.1147/sj.132.0115

[22] M. Perepletchikov, C. Ryan, K. Frampton, and Z. Tari, “Coupling metrics for predicting maintainability in service-oriented designs,” in *2007 Australian Software Engineering Conference (ASWEC’07)*, April 2007, pp. 329–340.

[23] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994. [Online]. Available: http://dx.doi.org/10.1109/32.295895

[24] Y. Wand and R. Weber, “Mario bunge’s ontology as a formal foundation for information systems concepts,” 1990.

[25] U. Kumari and S. Upadhyaya, “An interface complexity measure for component-based software systems,” *International Journal of Computer Applications*, vol. 36, 01 2011.

[26] E. Ogheneovo, “On the relationship between software complexity and maintenance costs,” *Journal of Computer and Communications*, vol. 02, pp. 1–16, 01 2014.

[27] D. Athanasopoulos, A. Zarras, G. Miskos, V. Issarny, and P. Vassiliadis, “Cohesion-driven decomposition of service interfaces without access to source code,” *IEEE Transactions on Services Computing*, vol. 8, pp. 1–1, 01 2014.

[28] L. C. Briand, S. Morasca, and V. R. Basili, “Property-based software engineering measurement,” *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 68–86, Jan 1996.

[29] 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.