Mono2Micro - From a Monolith to Microservices: Metrics Refinement

João Francisco Martins dos Santos Almeida

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. João António Madeiras Pereira
Supervisor: Prof. António Manuel Ferreira Rito da Silva
Member of the Committee: Prof. Luís Eduardo Teixeira Rodrigues

January 2021

Acknowledgments

First I would like to thank all my family, my parents, sisters, grandparents and aunt for their care, understanding and support throughout all these years.

I would also like to acknowledge my dissertation supervisor, Prof. Antonio Rito Silva for his insight, support and sharing of knowledge that has made this thesis possible.

Last but not least, to all my friends and colleagues that helped me grow as a better person.

To each and every one of you – Thank you.

Abstract

The microservices architecture has become mainstream for the development of business applications because it supports the adaptation of scalability to the type of demand, but, most importantly, allows a faster development because it fosters an agile development process based on small teams focused on the product. Consequently, a trend in migrating the existing monolith systems to a microservices architecture was born. Current approaches to the identification of candidate microservices in a monolith neglect the cost of redesigning the monolith functionality due to the impact of the CAP theorem. In this thesis we propose a redesign process, guided by a set of complexity metrics, that allows the software architect to analyse and redesign the monolith functionality given a candidate decomposition. We address two new research questions: (1) What set of operations can be provided to the architect such that the functionalities can be redesigned by applying microservices pattern? (2) Is it possible to refine the complexity value associated with the monolith migration when there is additional information about the functionalities redesign? - Regarding the first question we propose a set of operations that the architect can use in the redesign process. To answer the second question we define new metrics that provide a more precise value on the cost of the migration. Both questions are evaluated in the context of candidate decompositions of two monolith systems, LdoD e Blended Workflow.

Keywords

Microservices; Monolith Application; CAP theorem; Metrics; Design Patterns.

Resumo

A arquitetura de microsserviços tornou-se a tendência para o desenvolvimento de aplicações porque suporta a adaptação da escalabilidade ao tipo de procura, mas, mais importante, permite um desenvolvimento mais rápido porque promove um processo de desenvolvimento ágil baseado em pequenas equipas focadas no produto. Consequentemente, surgiu uma tendência de migração dos sistemas monolíticos existentes para uma arquitetura de microsserviços. As abordagens atuais para a identificação de microsserviços num sistema monólito negligenciam o custo de redesenhar a funcionalidade do monólito devido ao impacto do teorema CAP. Nesta tese propomos um processo de redesenho, guiado por um conjunto de métricas de complexidade, que permitam ao utilizador analisar e redesenhar a funcionalidade do monólito a partir de uma decomposição candidata. Abordamos duas novas questões de pesquisa: (1) Que conjunto de operações pode ser fornecido ao arquiteto para que as funcionalidades possam ser redesenhadas aplicando padrões de microserviços? (2) É possível refinar o valor de complexidade associado à migração do monólito quando há informações adicionais sobre o redesenho das funcionalidades? - Em relação à primeira questão, propomos um conjunto de operações que o arquiteto pode utilizar no processo de redesenho. Para responder à segunda pergunta, definimos novas métricas que fornecem um valor mais preciso sobre o custo da migração. Ambas as perguntas são avaliadas no contexto de decomposições de dois sistemas monolíticos, LdoD e Blended Workflow.

Palavras Chave

Microsserviços; Aplicações Monoliticas; Teorema CAP; Métricas; Padrões de Desenho.

Contents

List of Figures

List of Tables

Listings

Acronyms

API	Application Program Interface
ACID	Atomicity, Consistency, Isolation, Durability
CAP	Consistency, Availability, Partition Tolerance
CUD	Create, Update and Delete
CQRS	Command-Query-Responsability-Segregation
FoSCI	Functionality-oriented Service Candidate Identification
SBS	Service-based System
MVC	Model-View-Controller
CCT	Calling Context Tree
DAG	Directed Acyclic Graph
SAC	System Added Complexity
FRC	Functionality Redesign Complexity
QIC	Query Inconsistency Complexity

1 Introduction

Contents

Microservices architecture [1] composes an application as a suite of independently deployable services and emerged from the need to have highly available and scalable systems that can be developed by multiple teams in an agile environment. On the other hand, the monolith architecture consists of one large application block that contains all the application business logic, that is implemented, deployed and scaled as a whole. As its size increases, it imposes several drawbacks such as the lack of agility, modifiability and deployability. Hence, for some applications, including for some well-known applications such as Netflix and Amazon [2], there is the need to migrate existing monolith systems to a microservices architecture.

The migration between architectures is a complex process that requires the identification of candidate microservices and the transition from an isolated transactional environment to a distributed transactional environment. Therefore, it is necessary to find the best decomposition for the domain model and for the functionalities execution flow that best suit the application, weigh the transition's complexity and compare it to the benefits to decide if the migration is worth it.

Our work is to improve the existing Mono2Micro application by developing new metrics that can better explain and analyse the complexity originated on an architectural transition such that the software architect responsible for assessing the migration can make the best decision possible.

1.1 Context

This thesis leverages on two previous works [3, 4] where a tool was developed that collects information from monolith systems and, based on similarity measures, suggests a microservices candidate decomposition. Its level of complexity can be assessed through a complexity metric. In [3], the work is separated in three stages:

1. Through static analysis of the monolith source code, a callgraph is generated that shows the business functionality call chain which can be used to retrieve the domain entities accessed by each method.

2. Provide a microservices candidate decomposition by identifying clusters based on a similarity measure. The clusters, or microservices, are defined using an Hierarchical Clustering algorithm[1] that takes as inputs the weight attributed to the relation between two domain entities. The weight from entity E1 to entity E2 is the quotient between the number of controllers that access both entities and the amount of controllers that access only E1.

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

[1] docs.scipy.org/doc/scipy/reference/cluster.html

Later, in [4] the initial work is extended. In this work the main relevant topic to our work is the developed complexity metric, that calculates the impact that the relaxing of atomic transactional behaviour has on the functionalities redesign and implementation. The others aspects introduced are:

- The extraction from the monolith source code of additional information. For each functionality, which domain entities are accessed and in what mode, read or write. By combining the different entities access an access sequence can be constructed for each controller.
- Defines additional similarity measures used in the identification of candidate microservices decompositions.
- Enhances the visualization tool to analyse the microservices decompositions by comparing several decompositions using the developed metric. Also, the architect is provided with a modeling tool that allows him/her to change a decomposition. Based on those changes the new complexity value if updated and presented.

1.2 Problem

The transition between architectures imposes a cost because the application cannot preserve the exact behaviour that existed in the monolith. This is due to the introduction of distributed transactions, as the monolith functionalities will be implemented through multiple independent microservices, as opposed to the single isolated unit in the monolith. Therefore, transaction management is more complex in a microservices architecture because transactions cannot be executed according to the Atomicity, Consistency, Isolation, Durability (ACID) properties, which introduces extra complexity for developers to handle.

This extra complexity is explained by the Consistency, Availability, Partition Tolerance (CAP) theorem [5], where given the microservices partitioned nature and its core requirement of high availability, the level of information consistency must be sacrificed. To solve this problem, the two-phase commit protocol [6] and sagas [7] were suggested to handle distributed transactions. However the two-phase commit protocol does not scale with many replicas, being the use of sagas, in the context of microservices architectures [8, 9], the main alternative to the two-phase commit protocol. On the other hand, the Application Program Interface (API) Gateway pattern has been proposed [8, 9] to implement queries in a distributed system.

Hence, in a microservices architectural transition the cost of adapting each functionality is based on three main factors:

- The business logic of each functionality must change due to the new partitioned environment. Since the two-phase commit protocol does not scale in large scale systems, with multiple replicas, each functionality must be adapted to be executed as a saga, with multiple ACID transactions, one in each microservice, instead of one single ACID transaction.

- One additional problem is the new levels of consistency that the application must deal with. As explained by the CAP theorem, in a partitioned environment with high availability, the consistency across the system can not be ensured and the application must now deal with a new consistency model. For instance, the eventual consistency model, that may cause temporary inconsistency within the system.

- As a consequence of applying the SAGA pattern, the four ACID properties guaranteed in the monolith when executing a transaction do not exist. A saga can not ensure the isolation between concurrent executions that access to the same entities. Therefore, each functionality must also be changed to accommodate the countermeasures necessary to ensure isolation.

Concluding, the transition to a distributed system and the introduction of sagas lead to a relaxed consistency that, some times, can be perceived by the end-user, which may affect the application requirements. Also, the functionality design and implementation becomes more complex because the use of sagas is more cumbersome, for instance there is the need to implement compensating transactions for each possible fault and the need to implement countermeasures that tackle the lack of isolation between sagas. All these problems generate a migration complexity that must be assessed.

1.3 Research Questions

In this thesis we leverage on the previous works and extended it by, given a decomposition and a complexity value, supporting the redesign of the functionalities and queries where microservices patterns such as the SAGA and the API Gateway patterns are applied, while the complexity value is tuned.

As explained, due to the high availability requirement in a microservices architecture, the transition from a monolith to a partitioned microservices system imposes a relaxed consistency model. This situation has been coined in [10] as the *Forgetting about the CAP Theorem* migration smell, which has been ignored by the literature, where its impact in the identification of candidate decompositions and in the migration design and implementation was ignored.

We will focus on the impact that the new consistency levels may have on the application functionalities, which, in the vast majority of the cases, may require their redesign and reimplementation. That impact, or complexity, will be captured by integrating new information provided during the functionality redesign, while integrating the microservices patterns.

In the previous thesis [4], a metric was composed that evaluates the functionality migration complexity based only on the number of other functionalities that access the same entities. This metric does not incorporate the use of microservices design patterns, recommended by the literature [8, 9] to obtain the optimal microservices architecture. With our approach we intend to improve the previous work and answer to the following research questions:

1. What set of operations can be provided to the architect such that the functionalities can be redesigned by applying microservices pattern?
2. Is it possible to refine the complexity value associated with the monolith migration when there is additional information about the functionalities redesign?

1.4 Contributions

By leveraging on the previous work, our main contributions will be:

- We define a set of operations to be used in the functionalities redesign to transform a monolith execution flow into a distributed microservices execution flow that follows the microservices design patterns.
- We refine the existing metric into a set of complexity metrics to measure the microservices system quality in terms of the cost associated with the relaxation of the transaction model.
- We enhance the visualization tool with modeling capabilities that supports the functionality redesign by applying the set of operations. During the redesign process the architect is informed in real-time about the functionality complexity value that is calculated after each change.
- We provide a visualization tool that allows the architect to compare different redesigns for the same functionality based on the metrics values and their execution flow.
- A research paper entitled Monolith Migration Complexity Tuning Through the Application of Microservices Patterns [11] presented at the European Conference on Software Architecture (ECSA 2020) where the initial version of this work is presented.

1.5 Organization of the Document

This thesis is organized as follows: Chapter 2 presents the state of the art in microservices design patterns, an analysis on metrics for microservices evaluation and also researches about visualization tools. In chapter 3 our approach to the specification of the set of operations is explained as well as our developed metrics to assess cost of migration. In section 4 we describe how we implemented our approach, in particular the data structures developed and how the proposed operations affect them. In 5 the features added to the visualization tool according to the solution architecture are shown. In chapter 6 we evaluate our set of operations and metrics with two monolith systems with expert decompositions. Chapter 7 concludes this thesis with final remarks and an overall summary of the work, and describes the possibilities for future work.

2
Background and Related Work

Contents

In this section we start by discussing the problems that arise with the architectural transition from monolith to microservices and how some design patterns and countermeasures can be applied to solve those problems. Secondly, we present existing research that focuses on the architectural transition and on microservices patterns: what patterns there are, how and when to use them. Additionally, we present research that has formulated metrics to evaluate their microservices systems, what is the rationale behind the metrics, their benefits and drawbacks. To conclude we show researches that implement and use a visualization tool for assessing microservices systems.

2.1 Background

2.1.1 Distributed Transactions in Microservices

As explained in section 1.2, when transitioning from a monolith to a microservices architecture, the vast majority of the systems must use the SAGA pattern to implement distributed transactions as opposed to the two-phase commit protocol [6], as suggested and explained in [8, 9]. The SAGA pattern can be applied to Create, Update and Delete (CUD) functionalities, and consist in dividing a transaction in multiple local transactions, where each local transaction is executed inside a single service following the ACID properties. A saga can have two different structures:

- Choreography - where the decision and sequencing is distributed through the saga participants.
- Orchestration - where the decision and sequencing is decided in one orchestrator class, inside a cluster.

If a functionality is executed as a saga, then each local transaction has a type depending on its location on the execution flow. One local transaction can be of three types:

- Pivot - a transaction that if succeeds then the saga is going to succeed.
- Retriable - transactions that occur after the pivot transactions, do not rollback.
- Compensatable - the transactions that may have to rollback.

In a saga there is at most one pivot transaction, and all transactions that are not retriable nor the pivot transaction, are compensatable.

Independently of the structure, the usage of sagas can guarantee the properties atomicity, consistency, and durability but cannot ensure the isolation property. The lack of isolation causes anomalies that consist in a transaction reading or writing information in a way that it would not if all the transactions were executed sequentially, not concurrently. These anomalies are:

- Lost updates - when a saga overwrites data without reading changes performed by others sagas.

- Dirty reads - when a saga reads data changed by others sagas that have not been committed.
- Nonrepeatable reads - a saga reads the same data twice and gets different results.

To handle the concurrency anomalies, the application business logic must implement countermeasures to fix the lack of isolation. These are countermeasures that can be used to prevent side effects from the lack of isolation, which must be accommodated within the business logic. They are:

- Semantic lock - Is an application level lock that sets some flag to indicate that some information is modified but not yet committed.
- Commutative updates - Design updates operations such that they can be commutative, i.e, if they can be executed in any order.
- Pessimist view - Reorders the steps of a transaction to minimize business risk due to a dirty read.
- Reread value - It prevents lost updates by rereading a value before updating it. If the value has changed then the transaction must abort.
- Version file - Turns non-commutative operations into commutative operations by recording the operations that are performed on a record so that it can reorder them.
- By value - Selects concurrency mechanism based on business risks. Dynamically evaluates between performing sagas or the tow-phase commit protocol.

Each one has their benefits and drawbacks. The Commutative Updates countermeasure forces the redesign of every functionality to become commutative with the other functionalities, which in some cases might be impossible due to the functionality business logic, leaving it restricted to the minority of functionalities that can be commutative. The Pessimist View is similar to the commutative updates solution. Since it forces the functionalities to change their execution flow, it may not be possible to use it in every functionality. The Reread Value can be useful to correct the Lost Updates or the Nonrepeatable Reads anomalies but it does not resolve the dirty reads anomaly, because, for instance, a controller might read a dirty value twice while the concurrent transaction executes, and it gets the same value in both readings, but latter the second transaction aborts, making the controller readings absolute. The Version File countermeasure can be applied only in some systems that do not require constant interaction with the end user, since it first records the submitted transactions into a file to later organize them into commutative operations and executes them. The By Value solution is difficult to implement because it forces the functionalities to be implemented using sagas and the Two-Phase Commit protocol. On top of that, the application must have a set of rules to decide when to execute each functionality using a saga or the Two-Phase Commit protocol. The rules must be statically assigned into the system by the developers or they must be dynamically calculated. In either case, the rules definition is complicated and imposes an additional overhead in the architectural decision.

On the other hand the Semantic lock is easier to implement and is more advantageous. It corrects all the isolation faults by creating intermediate states as application-level locks that indicate if an entity was written by one saga, alerting others concurrent sagas to these events. It can be integrated with the SAGA pattern typically indicating the current saga state.

2.1.2 Queries in Microservices

Queries also need to be modified in an architectural transition. Similar to what happens with distributed transactions, queries will also be divided in multiple local queries, each one executed inside a microservice. There are two possible approaches in the literature [8, 9] to realize queries in microservices:

- API Gateway - A Gateway is created and is responsible for performing the queries. To execute a query, the gateway contacts each microservice where the required information resides, then combines its results and returns the query value.

- Command-Query-Responsability-Segregation (CQRS) - It separates the read and write operations over an entity into a ”command” and a ”query” side. The ”command” site is responsible for the CUD operations while the ”query” side manages the reads operations, using different interfaces for each side.

The CQRS pattern offers high performance when multiple operations are performed over the same information. However, it is a very complex pattern to implement, forces the separation between functionalities that read and write an entity and is necessary to develop different interfaces to each side, making it not suitable to every system. On the other hand the API Gateway pattern is easier to implement, but imposes the overhead of creating the gateway and is more difficult to ensure transactional data consistency.

2.2 Related Work

In [9] a qualitative study of 35 practitioner descriptions of best practices and patterns on microservices data management architectures is performed based on the model-based qualitative research method described in [12], that can be used to study established practices in a field. The model-based qualitative research method starts with some research questions that are investigated based on a pattern mining technique which starts with the authors’ own experiences, searches systematically for other known uses in real-life systems, and then applies a series of feedback loops to improve the pattern.

Based on that model, the authors developed an architectural design decisions model that groups the microservices practices and patterns studied. It contains one decision category, Data Management Category, relating five top-level decisions that are necessary to manage data in microservices. The decisions are the realization of queries, microservices transaction management, data sharing between microservices, microservices database architecture and the structure of API presented to clients. For each decision an UML style diagram is composed where the patterns to use and the decisions that are necessary to make with each one are specified. Accompanied with every diagram the authors give an explanation about the patterns, their advantages and drawbacks. To evaluate their work, the authors measure the improvement yielded by their study compared to the individual sources, specifically *microservices.io* [8], the most complete and detailed source they have found. They studied the *microservices.io* texts in detail and after careful analysis, the authors conclude that their study has an improvement of 210% in completeness because it captures 325 elements against only 105 elements in *microservices.io*.

In [13] a framework called Functionality-oriented Service Candidate Identification (FoSCI) is developed with the main responsibility of identifying service decomposition candidates, including entity and interface identification. It identifies service candidates through extracting and processing execution traces. Its process is divided into three main steps:

1. Representative Execution Trace Extraction - Extraction of representative execution traces based on an execution log file.

2. Entity Identification - Based on the execution traces, the tool identifies functional atoms and uses a multi-objective optimization technique to group them as class entities.

3. Interface Class Identification - For each service candidate, its interface classes with its operations are identified.

Additionally, an evaluation tool is composed to assess the service candidates conformance with 8 metrics that quantify three quality criteria of service candidates. The metrics are derived from the revision history stored in the application version-control system and provides the evolution path a software system experienced. The quality criteria are:

- Independence of Functionality - a functionality should be a well-defined, independent, and coherent function provided by an application, which should be a business capability accessible by external clients.

- Modularity - Focuses on the coupling and cohesion of the system. Measures if internal entities within a service behave coherently, while entities across services are loosely coupled.

- Evolvability - Measures a service's ability to evolve independently.

For the Independence of Functionality the authors created three measures. The first is called *ifn* (interface number) and measures the number of published interfaces of a microservice. The rational behind it, is that the smaller the *ifn*, the more likely the service assumes a single responsibility. The second metric is called *chm* (cohesion at message level) and evaluates the cohesiveness of interfaces published by a service at the message level, where the higher the *chm*, the more cohesive the service is. The last metric is the *chd* (cohesion at domain level) metric and it measures the cohesiveness of interfaces provided by a service at the domain level. The higher the *chd*, the more functionally cohesive this service is. To evaluate the modularity of a decomposition two metrics were composed. The first metric is called *SMQ* (Structural Modularity Quality and measures the modularity quality from a structural perspective. The higher the *SMQ*, the better modularized the service is. The second metric is called *CMQ* (Conceptual Modularity Quality) and it measures the modularity quality from a conceptual point of view, where the higher *CMQ*, the better. To assess the decompositions' evolvability the authors created three measures, where the first is called *icf* (internal co-change frequency) and measures how often entities inside a service change together as recorded in the revision history. A higher *icf* indicates the entities are more likely to evolve together. The second metric is called *ecf* (external co-change frequency) and it evaluates how often entities in different clusters change together based on the revision history. A lower *ecf* is better and indicates that entities in different services are more likely to evolve independently. The last metric is *REI* (Ratio of ECF to ICF) and, as the name suggests, measures the quotient between the two previous metrics. If the ratio value is lower than one then is good because changes are more likely to occur only inside one cluster.

To evaluate their approach, the authors compare FoSCI with three existing methods to identify service candidates decompositions, using six open-source projects. The evaluation indicated that their method outperforms the other three baseline methods with respect to their composed metrics, but it lacks an evaluation with more reliable and accepted metrics.

Comparing their work to ours, while they have a similar pipeline for the microservices identification, their metrics focus on different microservices qualities. Our focus is on the functionalities redesign complexity through the application of design patterns while they focus on the three quality criteria presented.

In [14] a similar work is performed. The authors present three formal coupling strategies that rely on meta-information from monolith code bases to construct graph representations of the monoliths that are in turn processed by a clustering algorithm to generate recommendations for microservice decompositions. Their pipeline consists of three phases, the monolith stage, the graph stage and the microservices stage, with two transformations between the stages: the construction step transforms the monolith into a graph representation and the clustering step decomposes the graph representation of the monolith into microservices candidates. Their extraction strategies are:

- Logical Coupling Strategy which is formulated around the single responsibility principle that states that a software component should have only one reason to change.

- Semantic Coupling Strategy that assess if each service corresponds to one single defined bounded context from the problem domain.

- Contributor Coupling Strategy that based on the implementation history, incorporates team-based factors into a formal procedure to group class files according to the authors of changes in the version control system.

To evaluate their system they used an open-source Java project and focused on the system performance and quality. For the performance, each one of the extraction strategies execution time was measured and concluded that they show good performance levels. For the quality aspect they formulate two metrics, the team size reduction metric and the average domain redundancy metric. The team size metric is computed as the average team size across all microservices candidates divided by the team size of the original monolith, and the domain redundant metric indicates the amount of domain-specific redundancy between the proposed microservices. For both metrics, they concluded that from the three strategies the semantic coupling showed the best results.

This research is similar to the previous works [3, 4] in what concerns the identification of candidates decompositions but when focusing on the main aspects of our work, they do not address the complexity that a decomposition has due to the transactional model transition and how the use of design patterns can improve the complexity. With their metrics they focus on non-functional requirements like the team size or the redundancy in the code.

The idea of an implementation cost can be viewed as the modifiability cost in a software system, which is closely related with maintainability. In [15], a literature review is conducted in order to investigate metrics that can be used to calculate the maintainability of a Service-based System (SBS). They define maintainability as the degree of effectiveness and efficiency with which a software system can be modified to correct, improve, extend, or adapt it. This definition is very close to one of our goals, as we intend to determine the cost associated with a system migration, which can be seen as a case of system adaptation. As a result they present the metrics grouped in four different categories that influence the system maintainability:

- Size - they define size as the number of services, stating that it is easier to maintain a small system than a big one. The metric presented to evaluate the maintainability based on this category is called Weighted Service Interface Count and calculates the amount of exposed interfaces on a given service.

- Complexity - defined as the amount and variety of internal work carried out as well as the degree of interaction between its services necessary to achieve this. Three metrics were presented to assess the complexity level. The first is the Total Response for Service and for a given operation in a system interface, calculates the number of sequences of other operations and implementation elements that can be executed. The second is called Number of Versions per Service and calculates the amount of versions that are used in a service. And the last is the Service Support for Transactions which represents the percentage of services with support for transactions.

- Coupling - defined as the degree of the strength of interdependencies of a service with other services. Four metrics were presented and started with the Service Interdependence in the System which has the number of service pairs that are bi-directionally dependent on each other. The second is called Absolute Importance of a Service and calculates the number of clients that invoke at least one operation of a service. The third is the Absolute Dependence of the Service and determines the number of consumers that depend on a service. The last is the Absolute Importance of the Service and quantifies the number of other services that a service depends on.

- Cohesion - defined as the extent to which the operations of a service contribute to one and only one task or functionality. The authors researched three metrics and the first is the Service Interface Data Cohesion measures the cohesion of a given service S with respect to the similarity of parameter data types of the operations of its interface. The second is called Service Interface Usage Cohesion and determines the percentage of operations in a service interface that are called by the same clients. Lastly, there is the Total Service Interface Cohesion that is simply a normalized version of the two previous metrics. It adds both values and then divides it by two.

The authors also explore how these metrics can be adapted to a microservices system, considered as a fine-grained variant of SBS. For them, the main aspects that differentiate microservices from SBS are the large number of small services, technological heterogeneity and decentralization of control and lightweight communication. With only a few limitations, the authors consider that the majority of metrics presented can also be applied to microservices. Among the variety of metrics discussed, the one that is closer to our complexity metrics is the Service Support for Transactions (SST), that calculates the percentage of services involved in a distributed transaction. Because the transition to distributed transactions is cumbersome, a high SST indicates a high complexity.

In [16] they implement a system that retrieves dynamic information from a microservices system and develop a tool called MAAT that evaluates the collected information, formalises an architecture model and evaluates the architecture with a set of metrics. The MAAT approach uses the Goal Question Metric [17] to generate metrics that can evaluate the principles discovered during the previous analysis. The metrics have as base the metrics defined in [15] and try to assess the level of conformity that a microservices system has in relation to microservices design principles by using those metrics. Those principles are collected from the literature and from a group of interviews performed to a group of people responsible for implementing five different microservices systems. Such principles include, for example, the need to have microservices with small size, the need to have the mapping between one task and one microservice, the need to have loose coupling and high cohesion, and more. For each principle one or more metrics are defined and after the phase of data collection, the architecture model is evaluated with them. Some of these metrics addresses problems like:

- the size of a microservice - evaluated by the number of synchronous and asynchronous interfaces of a service.
- the responsibility of a microservice - a service should only have one responsibility - also evaluated by the number of synchronous and asynchronous interfaces of a service and by the distribution of synchronous calls.
- loose coupling and high cohesion - evaluated by the number of synchronous and asynchronous dependencies. Denotes the number of service calls, either via synchronous or asynchronous communication dependencies.

To present the results to the user the tool MAAT provides a visualization tool that presents the model evaluated with all the constructed metrics, representing different types of communication: asynchronous and synchronous. It allows to select a specific metric and the results for that metric is shown in the model, with the help of a color scheme that highlights the problematic zones in the architecture according to that metric. Their tool is incorporated into a real system and the tool is evaluated by the project team members.

What differentiates it from our work is that in [16] they focus on a large spectrum of principles, and try to formulate metrics that evaluate them. Whereas, in our approach we focus on an in-depth study of the decomposition complexity by analysing the individual complexity contribution of each business functionality. The visualization tool is similar, but our focus is in a task model provided to the architect that can aid him/her in the complexity analysis. Additionally, the extended part created during this research is more focused on a functionality level, offering the possibility of carrying out its redesign, which makes it also a modelling tool.

In [18] the authors assess the quality of a candidate decomposition resulting from an architectural transition by defining design constraints and in turn metrics that evaluate those constraints, based on microservices patterns from [8]. To verify if the architectural model is following the best practices described by the design patterns, they select a few of them and find out what are their characteristics. Then, develop a constraint for each pattern and based on that constrain construct a set of metrics to perform the evaluation. They define metrics for two general design constraints:

1. The need to, in a microservices system, all the services to be independently deployed.

2. The need to avoid sharing components or entities between services.

From these design metrics the more similar to our approach is the second, however they do not focus on the implementation cost that derives from an architectural change. To evaluate these metrics the authors perform a manual analysis on a group of systems where they investigate if the system follows a microservices architecture and the design constraints imposed by the patterns. Based on this manual evaluation a value is given to the system that is later compared with the value returned by the metric. To observe the differences between them, the cosine similarity is computed and based on these values an evaluation is performed. This work is similar to ours, but, once more, they focus on different aspects of a microservices system and do not address the problem that is originated in data consistency and explained by the CAP theorem.

In [19] they focus on recovering the microservices architecture from a pre-existing microservices system. They aim at constructing a model for the microservices architecture and evaluate if it follows the microservices qualities, like single responsibility in a service or high cohesion and low coupling. Their process of architectural recovery is divided into two different phases:

1. First they perform the architectural recovery phase, where they follow a process similar to the previous and current work on the Mono2Micro application, since they perform a static and dynamic analysis to recover information to extract the microservices architectural model of a system.

2. Secondly, they perform an architecture refinement phase where the architect refines the model from phase one to enhance it with measures that are more suitable for its needs.

This work differs from ours since it does not construct any metrics to evaluate the functionalities complexity in a given decomposition. However, their work offers the possibility to the user to change the captured model and adapt it to what he/she considers the best for the given system, which is similar to our approach in the functionality redesign process when the user can apply the proposed set of operations to modify the functionality execution trace.

In [20] they propose a visualisation tool to assess the decomposition from a monolith to a microservices system. Their process is decomposed in three phases.

1. They start by generating a Calling Context Tree (CCT) that is similar to a call graph that contains information about the estimated amount of communication between microservices. They filter the CCT such that calling contexts that are unnecessary are removed, for instance a call to a library.

2. Secondly they generate an initial microservices design by employing two different clustering algorithms. The first is a semantic based clustering algorithm that uses the k-means++ with ttf-idf-based similarity and considers text features in source files. It groups classes into a cluster depending on the content of each class. The second is a CCT-based clustering that forms clusters based on the amount of communication between components with the purpose of reducing

the amount of communication, providing high performance. They calculate the similarity between functions, giving high similarity scores to functions on the same call flow, that they consider to be highly correlated.

3. The last step is the presentation of the decompositions in the visualisation tool to be refined by the architect. A decomposition is presented as a directed graph where the nodes represent classes and the edges represent function calls. It presents the classes that belong to the same cluster by colouring them with the same colour.

To refine the decomposition the tool provides three actions to the users:

- Create a new microservice with the selected class.
- Move the selected class into another microservice.
- Clone the selected class into all the microservices that communicate with the class.

Additionally, the tool contains extra features like the display of a list containing the descending order of number of API calls of a class. This list is a system recommendation of a set of classes that should be addressed to reduce the amount of services communication.

The authors evaluate the tool by assessing it against two systems that have a monolith and a microservices model and implementation, i.e, systems that started to be a monolith and transited to a microservices architecture. They compared the decompositions generated by their clustering algorithms with the case studies models produced by the developers, and also performed an evaluation on the refined actions performed by the developers. They evaluate the refinements by formulating a pairwise similarity metric that evaluates how many relationships between two classes are consistent in the cases of the official model, provided by the systems developers, and the one proposed by the tool. Also, another evaluation was performed to assess the number of API calls between microservices before and after the refinements. With these evaluations they intend to verify if through the user's actions the decompositions provided by the tool is improved, i.e, if it is more similar to the oficial mode or if it has less communication between classes. The results from the decompositions comparison shows that their tool can effectively design microservices applications, and the results from the refinement evaluation shows that after the refinement the level of similarity between models are high and that the number of API calls was reduced.

Comparing it to our work, the decompositions generation based on clustering algorithms and an initial visualization tool are already performed in the Mono2Micro tool by previous contributions, however in this work we intend to implement the third step of the tool process in [20], where we give to the architect, or to the developers, the change to refine the microservices decomposition model based on the microservices design patterns and the recalculation of the complexity metric. It should be noted that in our case the goal is not to diminish the complexity, because the Mono2micro application should already present the decomposition with lower complexity, however is to provide to the architect a set of choices (the proposed set of operations) that the he/she can apply to incorporate the design patterns.

In [21] the authors present a tool to find microservices candidate decompositions based on static analysis of the source code and dynamic analysis of the system's runtime behavior. Additonality, a visualization tool showing the execution traces is presented. To test their approach the authors used a legacy lottery application and applied their research in three stages:

1. Familiarization - Used to gather information about the application. The authors used documentation provided by the developers and conducted some interviews with them in order to create a context diagram of the application.

2. Modeling - Used to clarify the application behaviour and derive an architectural overview of the system. Once more with the help of developers and domain experts the application use cases and domain contexts were identified. Later, a mapping between the use cases and the domain contexts was constructed that revealed the connection between actors and code structures.

3. Partitioning - Using the collected domain knowledge, the authors identity target boundaries in the context model. These boundaries represent scopes in which each domain term has a unique definition, i.e., bounded contexts that are used during the decomposition process.

After dividing the application into a conceptual domain model, the next step was, by using static analysis, to bound source code packages to the use cases defined in the modeling phase, that in turn are associated with bounded contexts, which allows to perform a direct bonding between code packages and bounded contexts. After this phase some ambiguities in the mappings were identified and resolved using dynamic analysis. The bounded contexts were refined in order to resolve the ambiguities and to discover additional microservices candidates. For instance, to resolve one mapping ambiguity that caused a mutual dependency between two services the authors performed a service redesign that in turn caused a functionality redesign, similar to our approach, that reconstructed the execution trace of that functionality.

In [22], Ntentos et al. assess how a set of microservices applications conform to coupling and cohesion patterns and practices. First, the authors form a qualitative study in microservices by gathering relevant literature and knowledge sources about established architecture practices and patterns. Based on this study a meta-model description of microservice architectures was built, verified and refined through iterative application in modelling a number of real world systems. Also, the authors propose three decisions that must be adopted in microservices systems to have high cohesion and low coupling:

- Inter-Service Coupling through Databases - address how each service employs data persistence such that microservices qualities are guaranteed. The pattern or practice that ensures a lower coupling between services is the *No Persistent Data Storage* or the *Database per Service*. The first is only applicable for services whose functions are performed on transient data and the second states that each service has its own database and manages its own data independently. The pattern which affects more negatively loose coupling is the *Shared Database* where services write to and read data from a common database. This pattern can be applied in two levels, the *Data Shared via Shared Database* where multiple services share the same table, causing a strong coupled system or *Databased Shared but no Data Sharing* where each service writes to and reads from its own tables, which has a lower impact on coupling.

- Inter-Service Coupling through Synchronous Invocations - assess how each service is dependent on others services. Ideally communication between the microservices should be, as much as possible, asynchronous. This can be achieved through the use of various technologies such as the *Publish/Subscribe* pattern, the use of a *Messaging* middleware, the *Data Polling* pattern or the *Event Sourcing* patterns [8].
- Inter-Service Coupling through Shared Services - evaluates how services are shared with others services, i.e, what other services one service needs to implement some functionality. The need for other services leads to chains of transitive dependencies between them, which is problematic when a service is unaware of its transitive dependencies. The authors define three cases, the first being the *Directly Shared Service* which is the simplest case and occurs when one microservice is required by more than one other service. The second is the *Transitively Shared Service* and occurs when a microservice is linked to other services via at least one intermediary service. The last one is the *Cyclic Dependency* which is formed when there is a direct or transitive path that leads back to its origin.

For each of the decisions presented the authors formulate a ground truth through a manual evaluation by the authors of each system on the decisions that are present in the systems, in a scale that varies from Supported, Partially Supported or Not Supported. Then for each decision group two metrics are formulated and evaluated by comparing the results to the ground truth. Their metrics evaluate different aspects of each decision that assess how a system supports high cohesion and low coupling, for instance the percentage of services that use an individual database or a shared database, the percentage of services that communicates with other services via asynchronous relay or the total number of other services that one service depends on. The metrics evaluation showed that the results are quite close to the manual ground truth. Comparing it to our work, the authors aim at evaluating already implemented microservices systems and do not focus on the architectural transition. Also, the authors focus on coupling and cohesion qualities that must be present in microservices applications while we try to evaluate the complexity of each functionality redesign.

3 Solution Architecture

Contents

Considering the use of microservices design patterns in the functionality redesign that are transited from the monolith due to the relaxation of the transactional model, we intend to improve the existing complexity metric by incorporating new information provided by the architect during the redesign process. However, firstly, to be able during the redesign process to design an execution flow correctly given the new microservices distributed environment a set of operations were constructed to guide the architect. Thus, we expect that by using this set of operations during the functionalities redesign process, that incorporate the use of microservices design patterns into the functionalities, we can extract new information that can be used to formulate new complexity metrics that can express more precisely the functionality complexity on a given decomposition.

Due to the functionality fragmentation across multiple distributed microservices, the functionality has to be changed and adapted to the new model. Since we no longer have a single ACID transaction but multiple ACID transactions, one per microservice that participates in the functionality execution, but still want to ensure the same behaviour as in the monolith there is the need to use the design patterns that allow us to maintain it. For instance, in the monolith since there is only one ACID transaction if a fault occurs the whole transaction is aborted and its effects are not visible to the outside system. However, in a microservices system if a fault occurs in a microservice and its transaction is aborted, then it has to warn all the other microservices that participate in the functionality execution to undo any possible modifications that may have been made as a result of these execution, otherwise an inconsistent state is created.

Additionally, in the monolith the transactions guarantee the properties of atomicity, consistency, isolation and durability, however with distributed transactions that use the SAGA patterns there is no guarantee of isolation. For instance, if two concurrent transactions access the same entity, if the first transaction modifies one value without committing it and that value is read by the second transaction, then a dirty read occurs and the second transaction might be executed based on an incorrect value if the first transaction aborts.

As a result the integration of microservices design patterns with compensating measures, as the semantic lock, becomes essential. Nonetheless their use is not a silver bullet and impose a cost. To start, there is a cost in each functionality for each semantic lock introduced by it, executed as a saga, because its business logic needs to be changed. Secondly, a functionality needs to be modified to adapt to the semantic locks introduced by other functionalities in entities it reads.

Beyond the impact that a functionality redesign has on it, we also consider the impact it has on the other functionalities. When a functionality creates a semantic lock in some entities it updates then all the other functionalities that read that same entity must be modified to accommodate the existence of the semantic locks.

These new metrics provide different information and their joint use allows to locate the focus of complexity introduced in the redesign of a functionality.

In this section we describe our approach to the generation of a set of operations to be used on the functionalities redesign in order to incorporate the microservices design pattern such that the new complexity metrics can be applied. We start by explaining the set of operations and how they focus on characteristics of the SAGA pattern allowing its incorporation into the functionalities. Then, we present the new set of complexity metrics and their explanation, concluding with the visualization tool that serves as a proof of concept, where, given a decomposition and a functionality from its application, the operations can be applied and the new metrics calculated.

3.1 Functionality Redesign

We start by presenting a set of formal definitions that will help to specify and understand the operations and metrics.

Definition: Monolith. A monolith is a pair (F, E), where F represents its set of functionalities, the functionalities are represented with lower case f, and E represents its set of domain entities, which are accessed by the functionalities, the domain entities are represented with lower case e.

The entities are accessed by the functionalities in two modes, read and write. Therefore, $M = \{r, w\}$ represents the access modes in a monolith, and an access is a pair domain entity access mode, represented by (e, m).

The accesses of a functionality f are represented as a sequence of accesses s, where S represents all the sequences of accesses done in the monolith by its functionalities to the domain entities, $f.sequence$ denotes the sequence of access of functionality f, $s.entities$ denotes the entities accessed in sequence s.

It is also defined the auxiliary function $entities(s : S, m : M) : 2^E$, which returns the entities accessed in s in mode m:

$$entities(s, m) = \{e \in E : (e, m) \in s\} \tag{3.1}$$

When a monolith is decomposed into a set of candidate microservices, each candidate microservice is a cluster of domain entities.

Definition: Monolith Decomposition. A monolith decomposition into a set of candidate microservices is defined by a set of clusters C of the monolith domain entities, where each cluster c represents a candidate microservice and $c.entities$ denote the domain entities in cluster c, such that all domain entities are in a cluster (3.2), and in a single one (3.3).

$$\bigcup_{c \in C} c.entities = E \tag{3.2}$$

$$\forall_{c_i \neq c_j \in C} c_i.entities \cap c_j.entities = \emptyset \tag{3.3}$$

Given $e \in E$ and a decomposition C, $e.cluster$ denotes the entity's cluster, and given a set of entities $E' \subseteq E$, $E'.cluster$ denotes its set of entities clusters:

$$E'.cluster = \{c \in C : \exists_{e \in E'} e \in c.entities\} \tag{3.4}$$

Given a monolith candidate decomposition, the monolith functionalities are decomposed into a set of local transactions, where each local transaction corresponds to the ACID execution of part of the functionality domain entity accesses in the context of a candidate microservice.

Definition: Functionality Decomposition. A monolith functionality f is decomposed, in the context of a candidate decomposition C, by a sequence of sequences of access to domain entities, denoted by $f.subsequences$, where all domain entities in a subsequence are in the same cluster (3.5), and two consecutive subsequences occur in different clusters (3.6). In order to have a consistent subsequence associated with a functionality f in a decomposition, the condition in (3.7) must hold:

$$\forall_{s \in f.subsequences} \exists_{c \in C} : s.entities \subseteq c.entities \tag{3.5}$$

$$\forall_{0 \leq i < f.subsequences.size-1} f.subsequences[i].entities.cluster \neq f.subsequences[i+1].entities.cluster \tag{3.6}$$

$$concat_{i=0..f.subsequences.size-1}(f.subsequences[i]) = prune(f.sequence) \tag{3.7}$$

Where the $prune$ function removes, for each sequence of accesses inside each cluster $c \in C$, the accesses according to the following rules:

1. If a domain entity is read, all subsequent reads of that entity are removed.
2. If an domain entity is written, all subsequent accesses of that entity are removed.

A sequence of domain entity accesses where these two rules hold is pruned, it only contains the read and write accesses that are visible outside the cluster, the ones that are relevant for the semantic lock countermeasure.

In the redesign of a functionality in the context of a decomposition we define the set of local transactions participating in the saga that implements the functionality.

Definition: Local Transaction. A local transaction lt of a functionality f, is a pair (s, t), where s is a pruned sequence of access domain entities, all its accesses are to domain entities of the same cluster (3.8), and t is the transaction type, which can be $compensatable$, $pivot$, and $retriable$.

$$\exists_{c \in C} : s.sequence.entities \subseteq c.entities \tag{3.8}$$

T denotes the set of transaction types, LT denotes the set of local transactions in a decomposition, lt denotes a local transaction, $lt.cluster$ denotes the cluster where the lt occurs, $lt.sequence$ denotes the sequence of accesses, and $lt.type$ denotes the type of the local transaction.

A local transaction sequence should be pruned, for each domain entity in the sequence there is $0..1$ read accesses and $0..1$ write accesses, and when there is a read and a write access to the same domain entity, the read access has to occur first. These are the accesses that have impact outside the local transaction atomic execution.

The redesign of a functionality in the context of a decomposition corresponds to the application of a set of operations to a graph which represents the functionality execution, where the nodes represent the functionalities' local transactions and the edges the remote invocations between transactions.

Definition: Functionality Execution Graph. A functionality f redesign in the context of a monolith decomposition is represented by a graph g, where the nodes are local transactions, denoted by $g.lt$, the edges are remote invocations between local transactions, denoted by $g.ri$, and the data dependencies existing in the functionality business logic are denoted by $g.dd$:

- $g.lt$ is the set of local transactions, such that:
  1. $\bigcup_{lt \in g.lt} lt.sequence.entities = f.sequence.entities$
  2. $\#\{lt \in g.lt : lt.type = pivot)\} \leq 1$
- $g.ri$ is the set of local transactions pairs that represent the remote invocations:
  1. $\forall_{(lt_i, lt_j) \in g.ri} \{lt_i, lt_j\} \subseteq g.lt$
  2. $\forall_{(lt_i, lt_j) \in g.ri} \neg \exists_{lt_{k \neq i} \in g.lt} (lt_k, lt_j) \in g.ri$
  3. The remote invocations define a partial order between the local transactions, denoted by $<_g$, and build using the transitive closure of the following initial elements $\forall_{(lt_i, lt_j) \in g.lt} lt_i <_g lt_j$. Therefore, given $lt_i, lt_j \in g.lt$ if $lt_i <_g lt_j$ then lt_i executes before lt_j.

The redesign of a functionality in the context of a decomposition starts with its initial graph, which is generated from the functionality decomposition.

Definition: Initial Graph. The initial graph g_I of a functionality f has as vertices the local transactions lt associated to each one of the subsequences of f (3.9), and has as edges the pairs of local transactions associated with consecutive subsequences (3.10). It is trivial to observe that the initial graph g_I is a well-formed graph of f.

$$g_I.lt = \{lt \in LT : lt.sequence \in f.subsequences \wedge lt.type \text{ is not defined}\} \tag{3.9}$$

$$\begin{aligned}(lt_j, lt_k) \in g_I.ri \rightarrow & \exists_{0 \leq i < f.subsequences.size-1} : \\ & lt_j.sequence = f.subsequences[i] \wedge lt_k.sequence = f.subsequences[i+1]\end{aligned} \tag{3.10}$$

A semantic lock is an intermediate state set by a compensatable local transaction, a write access, that is visible by the other functionalities, and that may eventually be undone.

Definition: Local Transaction Semantic Lock. Given an execution graph g of a functionality f, and one of its local transactions lt, $lt.sl$ denotes the domain entities with a semantic lock in lt (3.11).

$$lt.sl = \bigcup_{(e,m) \in lt.sequence} (lt.type = compensatable \wedge m = w) \tag{3.11}$$

Definition: Functionality Semantic Lock. Given an execution graph g of a functionality f, $g.sl$ denotes the domain entities with a semantic lock in g:

$$g.sl = \bigcup_{lt \in g.lt} lt.sl \tag{3.12}$$

Definition: Final Graph. A final graph g_F of a functionality f is a graph of f where all transactions have a type and all the transactions that follow the pivot transaction are retriable (3.13). Additionally, it is not possible to have a remote invocation between local transactions belonging to the same cluster (3.14). Given that a graph has at most one pivot transaction, and in a final graph all transactions have a defined type, it is trivial to observe that all the transactions that do not occur after the pivot transaction should be compensatable.

$$lt_i \in g_F.lt : lt_i.type = pivot \implies \forall_{j t_j : lt_i <_{g_F} lt_j} lt_j.type = retriable \tag{3.13}$$

$$\forall_{(lt_i, lt_j) \in g_F.ri} \{lt_i.cluster \neq lt_j.cluster\} \tag{3.14}$$

Definition: Redesign Process. The redesign of a functionality f is a process that starts with its initial graph g_I and through the application of graph operations produces a final graph g_F, where, in a first step, the software architect will perform operations over the execution graph to redesign the execution flow of f, and, finally the architect will characterize the type of local transactions, such that the *SAGA* pattern is applied to the functionality f in the context of the monolith decomposition.

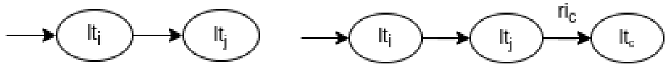
We propose three basic operations and a composed operation to support the redesign of a functionality. The basic operations are: *Sequence Change*, where the order by which the local transactions are invoked is changed; *Local Transaction Merge*, where two local transactions belonging to the same cluster are merged; and, *Add Compensating*, where a new local transaction is added when it is necessary to undo the changes done by local transactions. Additionally, we propose a composed operation, *Define Coarse-Grained Interactions*, where repetitive fine-grained interactions between two candidate microservices are synthesized into a single coarse-grained interaction.

By applying these operations, the software architect transforms the sequence of local transactions in the initial graph to a saga like interaction, either an orchestration or a choreography, where in the former case there is a cluster that coordinates the execution flow between the local transactions.

3.1.1 Operations

Definition: Add Compensating. Given a graph g of functionality f and two local transactions lt_c and lt_j, where:

- $lt_c \notin g.lt$ - The new local transaction added must be new.
- $lt_j \in g.lt$ - lt_j must be an existing local transaction.
- $lt_j.cluster \neq lt_c.cluster$ - The clusters of lt_c and lt_j must be different.
- $lt_c.sequence.entities = \bigcup_{lt_i \in g.lt} \{e \in entities(lt_i.sequence, w) : lt_i.cluster = lt_c.cluster \wedge lt_i.type = compensatable \wedge lt_i <_g ri_c[1]\}$ - The access sequence of lt_c must only contain entities that were previously accessed in write mode by a compensatable local transaction. This condition is particularly important because compensating transactions must only affect entities that have been updated before in the saga.
- $\forall_{(e,m) \in lt_c.sequence} m = w$ - The access mode for the entities in lt_c sequence must be write.



(a) Before the operation **(b)** After the operation

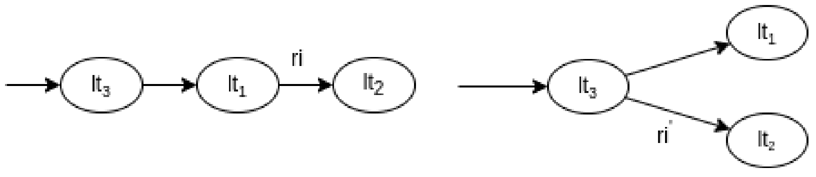
Figure 3.1: Result of applying the Add Compensating operation

Then a new graph $g' = (g.lt \cup \{lt_c\}, g.ri \cup \{ri_c\})$ is produced by creating a new remote invocation between lt_j and lt_c, $ri_c = (lt_j, lt_c)$, as explained in figure 3.1. This operation is used to create new local transactions that access some of the domain entities changed by other local transactions. It can be used to create the compensating transactions that are necessary for each compensatable transaction.

Definition: Sequence Change. Given a graph g of functionality f, three distinct local transactions, $lt_1, lt_2, lt_3 \in g.lt$ and a remote invocation $ri = (lt_1, lt_2) \in g.ri$ where:

- $lt_1 \neq lt_2 \neq lt_3 \neq lt_1$ - All local transactions are different for each others.
- $lt_3 <_g lt_2$ - lt_3 must be executed before lt_2.

It is possible to replace ri by $ri' = (lt_3, lt_2)$, such that g is transformed to $g' = (g.lt, g.ri \setminus \{ri\} \cup ri')$, a graph of f as exemplified in figure 3.2. It is trivial to observe that the transformed graph is a well-formed graph of f in the context of the decomposition, because lt_3 executes before lt_2 we can conclude that the resulting order continues to be a partial order and all local transactions are remotely invoked by at most one local transaction.



(a) Before the operation **(b)** After the operation

Figure 3.2: Result of applying the Sequence Change operation

The *change sequence* operation is used to change the flow of execution of the functionality in the context of the decomposition and it is possible to apply when no local transaction in the invocation chain between lt_3 and lt_2 requires data produced by lt_2. For instance, to change the local transaction (hence the cluster) that is responsible to trigger the execution of another particular local transaction, which may be useful to centralize the control of execution in a microservice that coordinates the execution of other local transactions, and so reduce the transactional complexity behaviour.

Definition: Local Transaction Merge. Given a graph g of functionality f and two local transaction $lt_1, lt_2 \in g.lt$, such that they belong to the same cluster, $lt_1.cluster = lt_2.cluster$, and they have adjacent executions, either:

1. Have consecutive executions and are connected by a remote invocation, such that:

$$(lt_1, lt_2) \in g.ri \tag{3.15}$$

2. Or they share the same caller that has one remote invocation for each local transaction, such that

$$\exists_{lt_i \in g.lt} : (lt_i, lt_1) \in g.ri \wedge (lt_i, lt_2) \in g.ri \tag{3.16}$$

A new graph g' of f is produced, where, considering the two cases:

1. For the first case, the g' local transactions set is formed by the g local transactions set excluding lt_1 and lt_2, plus the new local transaction lt_m, such that:

$$\begin{aligned} &g'.lt = g.lt \setminus \{lt_1, lt_2\} \cup lt_m \\ &\text{where } lt_m.sequence = prune(concat(lt_1.sequence, lt_2.sequence)) \end{aligned} \tag{3.17}$$

   The new g' remote invocations set is formed by the former g remote invocations set removing the remote invocation that lt_0 calls lt_1, the remote invocation between lt_1 and lt_2, all the remote invocations that start in lt_1 or lt_2 and adding the remote invocation between lt_0 and the new local transaction lt_m and one remote invocation between the lt_m and each local transaction that was called by lt_1 or lt_2, such that:

$$\begin{aligned} &g'.ri = g.ri \setminus \{(lt_1, lt_2)\} \setminus \{(lt_o, lt_1) : (lt_o, lt_1) \in g.ri\} \setminus \{(lt_k, lt_l) \in g.ri : lt_k = lt_1 \vee \\ &lt_k = lt_2\} \cup \{(lt_o, lt_m) : (lt_o, lt_1) \in g.ri\} \cup \{(lt_m, lt_i) : (lt_1, lt_i) \in g.ri \vee (lt_2, lt_i) \in g.ri\} \end{aligned} \tag{3.18}$$

2. For the second case, the g' local transactions set is equal to the first case, while the g' remote invocations set is formed by removing from g remote invocations set the remote invocations to lt_1 and lt_2 and all the remote invocations that start in lt_1, lt_2 or lt_i, and adding the remote invocation between lt_i and the new local transaction lt_m and one remote invocation between the lt_m and each local transaction that was called by lt_1, lt_2 or lt_i, such that:

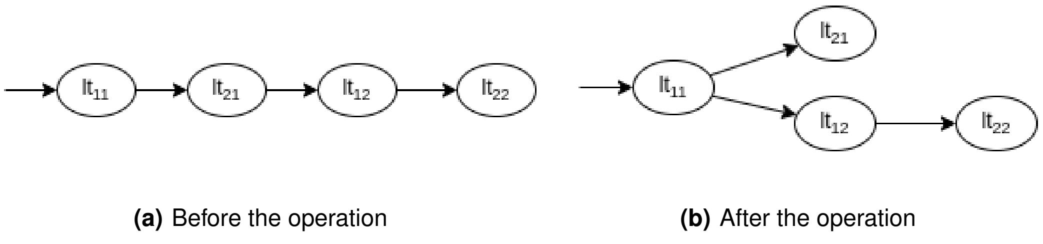
$$\begin{aligned} g'.ri =& g.ri \setminus \{(lt_k, lt_l) \in g.ri : lt_k = lt_1 \vee lt_k = lt_2 \vee lt_k = lt_i\} \cup \{(lt_i, lt_m)\} \\ &\cup \{(lt_m, lt_l) : (lt_1, tl_l) \in g.ri \vee (lt_2, tl_l) \in g.ri \vee (lt_i, tl_l) \in g.ri\} \end{aligned} \tag{3.19}$$

The *local transaction merge* operation is used when, in the redesign process, two local transactions become adjacent in the execution graph, and can be included into a single local transaction. From the transactional perspective, it is necessary to integrate their execution sequences, what is achieved with the prune function, and in the second case, is the software architect that decide the order by which the sequences are integrated. As result of applying this operation, the number of intermediate states in result of the distributed execution of the functionality is reduced.

Definition: Define Coarse-Grained Interactions: Given a graph g of functionality f, two candidate microservices, represented by the clusters $c_1 \neq c_2$ and two remote invocations $\{(lt_{11}, lt_{21}), (lt_{12}, lt_{22})\} \in g.ri$, such that:

- The remote invocations are between the given microservices, $c_1 = lt_{11}.cluster = lt_{12}.cluster \wedge c_2 = lt_{21}.cluster = lt_{22}.cluster$.
- lt_{11} executes before lt_{12}, $lt_{11} <_g lt_{12}$.

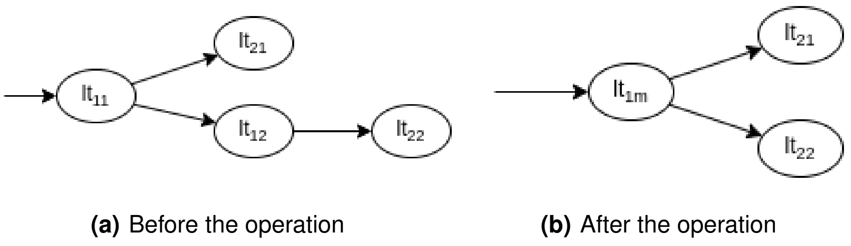
Then a new graph g' of f is produced by applying the basic operations *change sequence* and *local transaction merge*. First, *change sequence* operation is applied to lt_{11} and (lt_i, lt_{12}), to produce a new graph with remote invocation (lt_{11}, lt_{12}) as described in figure 3.3. Note that is possible to apply the operation, because $lt_{11} <_g lt_{12}$ and so there exists the remote invocation (lt_i, lt_{12}).



(a) Before the operation **(b)** After the operation

Figure 3.3: First step in the Define Coarse-Grained Interaction operation.

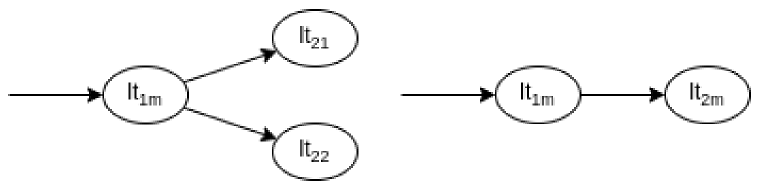
Then, *local transaction merge* operation is applied to lt_{11}, lt_{12} to produce a new local transaction lt_{1m} which has remote invocations to lt_{21} and lt_{22} as described in figure 3.4.



(a) Before the operation **(b)** After the operation

Figure 3.4: Second step in the Define Coarse-Grained Interaction operation.

Finally, *local transaction merge* operation is applied to lt_{21} and lt_{22} which results in the local transaction lt_{2m} and a coarser-grained remote invocation (lt_{1m}, lt_{2m}) as described in figure 3.5. Note that this operation can be applied to any number of remote invocations between two cluster, in the given conditions.



(a) Before the operation **(b)** After the operation

Figure 3.5: Last step in the Define Coarse-Grained Interaction operation.

This operation is used to create two coarse-grained local transactions, one in c_1 and another in c_2, by joining local transactions that are executed in those clusters, in order to reduce the number of remote invocations. It must be used when, after the automatically generated decomposition, the software architect realizes that there are several recurring fine-grained interactions between two candidate microservices, due to an object-oriented programming style in the monolith, which promotes the use of fine-grained invocations between the domain entities.

After the operations have been applied to the initial graph g_I of functionality f, the last step of the redesign is to produce a final graph g_F through the characterization of each one of the local transactions. Therefore, the software architect must select one transaction in the graph to be the $pivot$ transaction. Transactions that follow the pivot transaction are guaranteed to succeed are classified as $retriable$, and all other local transactions are classified as $compensatable$. The compensatable transactions that have semantic locks need to have at least one compensating transaction because some of the transactions that execute after it in the saga might fail.

3.2 Metrics

We start by explaining in detail the metric developed in the previous work [4] that serve as base for our work. Given a monolith decomposition, the base metric measures the complexity associated with the migration of a monolith system to a microservices architecture. It considers the complexity of each functionality redesign for the overall complexity of redesigning the monolith system, due to relaxing the functionality execution isolation, because the redesign of a functionality has to consider the intermediate states introduced by the execution of other functionalities.

Definition: Functionality Complexity in a Decomposition. Given a candidate decomposition C of a monolith, the complexity associated with the migration of a monolith functionality f is given by

$$\sum_{s_i \in f.subsequences} \# \bigcup_{(e,m) \in s_i} \{f_i \in F \setminus \{f\} : (e, m^{-1}) \in prune(f_i.sequence)\} \tag{3.20}$$

Where f_i is a distributed transaction, it executes in more than one cluster, and m^{-1} represents the inverse access mode, $r^{-1} = w$ and $w^{-1} = r$.

The overall idea behind the metric is to count, for each subsequence of a functionality, executing inside a cluster, the impact domain entities accesses have. The impact of a write depends on other functionalities that read it, and, therefore, they may have to consider this new intermediate state, while the impact of a read depends on how many other functionalities write it, and, therefore, introduce new intermediate states to be considered by the functionality. This metric reflects how many other functionalities need to be considered in the redesign of a functionality, thus, how the functionality redesign is intertwined with others functionalities business logic redesign.

However, during the redesign process, while the functionalities are redesigned, the concepts of local transaction and remote invocation are introduced, which allows a refinement of the previous metrics, such that, during the redesign process, the software architect can have more precise values about the complexity.

Therefore, and because the metric will be used to inform the functionality redesign activity, we distinguish between the complexity of redesigning the functionality from the complexity that the functionality redesign adds to the redesign of other functionalities.

Definition: Functionality Redesign Complexity. The complexity of redesigning a functionality f, executed as a graph g, is the sum of the complexity of each one of its local transaction:

$$complexity(f) = \sum_{lt \in g.lt} complexity(lt) \tag{3.21}$$

The complexity of one local transaction depends on the number of semantic locks that are introduced, because each semantic lock corresponds to an intermediate state for which may be necessary to write a compensating transaction, and it also depends on the intermediate states set by other functionalities that the local transaction may have to consider in its reads. Note that, during the redesign of a functionality, some of the functionalities that f interacts with may not have been redesigned yet, and so, the metric should take into account both situations.

Definition: Local Transaction Redesign Complexity. The complexity of lt is given by the number of semantic locks implemented in entities of $lt.sequence$, plus the number of other functionalities that write in entities read in lt, or which have semantic locks in those entities:

$$complexity(lt) = \#lt.sl + \sum_{(e,r) \in lt.sequence} \#\{f_i \in F \setminus \{f\} : (e,w) \in writes(f_i)\} \tag{3.22}$$

where

$$writes(f_i) = \begin{cases} \{(e,w) : (e,w) \in prune(f_i.sequence)\} & \text{if } f_i \text{ not redesigned} \\ \{(e,w) : (e,w) \in g_i.sl\} & \text{if } f_i \text{ redesigned as } g_i \end{cases} \tag{3.23}$$

Note that when a functionality is redesigned some writes may not be considered, because if they belong to pivot or retriable local transactions, they will not introduce intermediate states, and so, the metric will provide a more precise value.

The redesign of a functionality impacts on other functionalities redesign complexity. For instance, if a semantic lock is created in one entity e due to the execution of a functionality f_i then every other functionality f_j (where $i \neq j$) that read the same entity e must have to be changed to accommodate the existence of the semantic lock. Hence, the cost of redesigning f_j depends on the amount of semantic locks created by f_i in entities that f_j access.

Definition: System Added Complexity. Given the redesign of a functionality f executed as a graph g, the system added complexity introduced by redesign g, is given by:

$$addedComplexity(f, g) = \sum_{lt \in g.lt} \sum_{f_i \in F \setminus \{f\}} \#(reads(f_i).entities \cap lt.sl.entities) \tag{3.24}$$

where

$$reads(f_i) = \begin{cases} \{(e, r) : (e, r) \in prune(f_i.sequence)\} & \text{if } f_i \text{ not redesigned} \\ \{(e, r) : \exists_{lt \in g_i.lt}(e, r) \in lt.sequence\} & \text{if } f_i \text{ redesigned as } g_i \end{cases} \tag{3.25}$$

The redesign of functionality f may introduce inconsistent states in the application when it has two or more semantic locks. However, this situation only occurs when the entities belong to different clusters, because inside one cluster the entities are updated simultaneously by ACID transactions. Hence, we consider that a functionality changes a cluster when it introduces a semantic lock in one of its entities. If we consider that a functionality f writes in more than one cluster, this behaviour may introduce inconsistency views for any other functionality f_i that reads two or more of the changed clusters. Therefore, any functionality f_i that reads domain entities in different clusters, previously changed by f, might encounter inconsistent states.

From a redesign point of view, the inconsistency state complexity is particular relevant for functionalities that only read and have a single local transaction for each cluster they access. We call queries to this type of functionalities.

Definition: Query. A query q is functionality which graph g has the following properties:

1. Its local transactions are read only, $\forall_{lt \in g.lt} lt.sequence.mode = \{r\}$
2. They only access a cluster at most once, $\forall_{lt_i \neq lt_j \in g.lt} lt_i.cluster \neq lt_j.cluster$

Note that, if there is a functionality that only has read accesses, it is possible, by applying the redesign operations, to generate an execution graph that is a query. We define the cost of implementing a query as the inconsistency state it has to handle.

Definition: Query Inconsistency Complexity. Given a query q, its inconsistency complexity is the sum of all the other functionalities that write in at least two clusters that q also reads:

$$queryInconsistencyComplexity(q) =$$

$$\#\{f_i \in F \setminus \{q\} : \#clusters(entities(prune(q.sequence)) \cap writes(f_i).entities) > 1\} \tag{3.26}$$

where $writes(f_i)$ is defined as in the local transaction complexity metric.

4

Implementation Data Structures

Contents

In this section we present the data structures implemented in the Mono2Micro application backend and frontend systems that together form the functionality execution graph described in section 3.1. We leverage on the structure implemented in the previous work [4], that can be seen in figure 4.1, where we only show the classes and its attributes that are connected with our work. After presenting the data structures added to the existing system, we will show, for each of the proposed operations, what effects it has on each structure and how one operation changes the functionality execution graph.

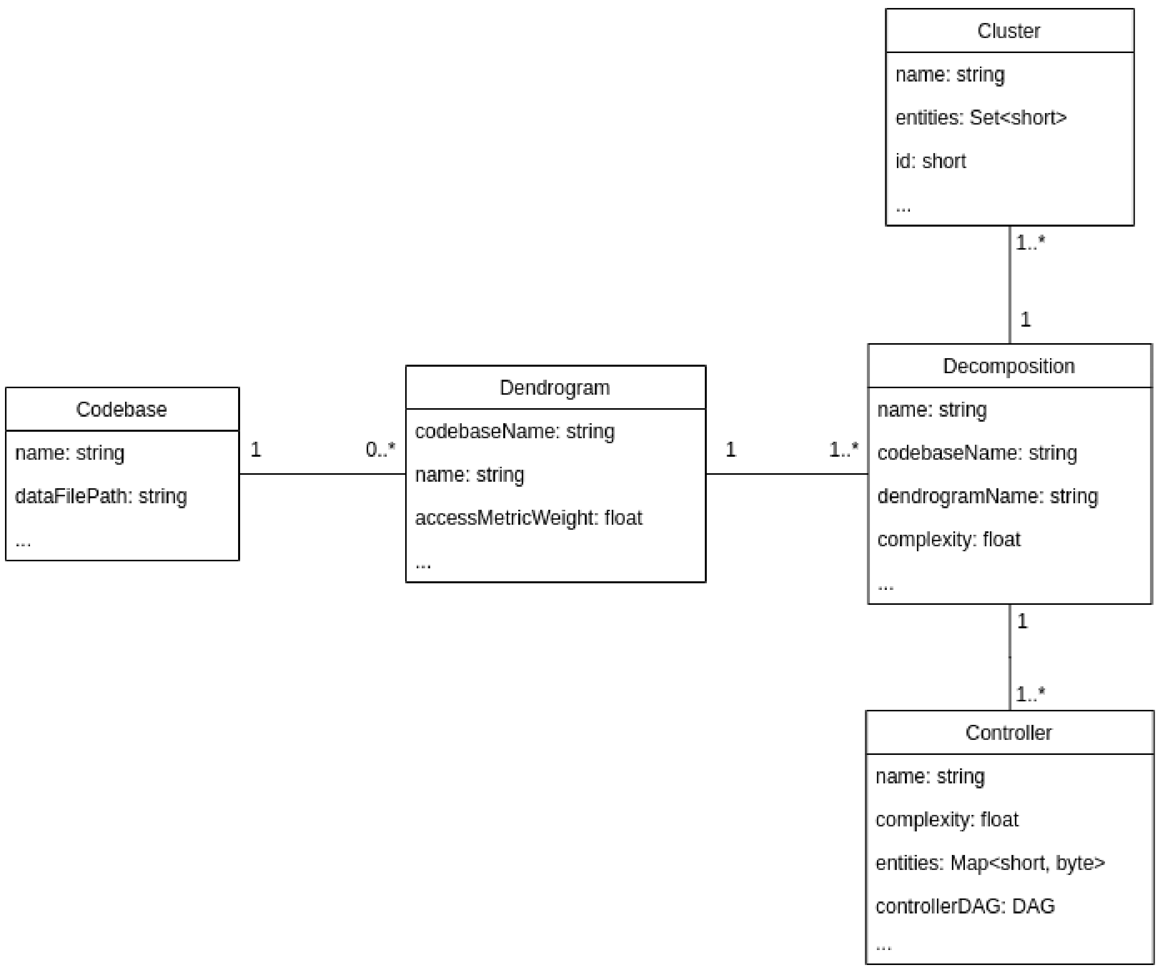


Figure 4.1: UML class diagram of the previous Mono2Micro system.

We have the java class *Codebase* that contains the information collected during the monolith analysis. Among other attributes, the *Codebase.java* contains a list of dendrograms. A dendrogram is represented by the *Dendrogram* class, where each dendrogram contains a list of all decompositions created for it. One decomposition is represented by the Java class *Decomposition.java* and is the representation of a candidate decomposition. Among other attributes, a decomposition is identified by a string with its name, it contains one float with the overall base complexity and contains a list of clusters, instances of the class *Cluster.java* and a list of controllers, where each controller is represented by the Java class

Controller.java. A cluster is identified by its name and its id and contains a set of shorts, where each short is the identifier of a an entity that belongs to the cluster.

Each controller corresponds to a business functionality and contains a name that identifies the controller, for example *ActivityModel.addActivity*, a float for the base complexity metric value, a map that associates the entities accessed by the controller to the access mode and a Directed Acyclic Graph (DAG) that constitutes the controller access sequence, named *controllerDAG*. The *controllerDAG* is formed, as explained in section 3.1, as a sequence of access sequences, where each access sequences is a node in the DAG and also a local transaction and the DAG edges are the remote invocations. The DAG can have a sequential trace or a three depending on the type of analysis made. If the code analysis made previously to the source code is a static analysis then the DAG will have a sequential trace will each node only having one remote invocation to other node. On the other hand if the analysis made was a dynamic analyses then the DAG trace is formed as a trace, where a node can have various remote invocations to others nodes.

4.1 Data Structures

The existing structure was not adequate for two reasons. First, it was to rigid and did not have enough flexibility that would allow us to have multiple executions traces for the same controller and to manipulate the execution trace accordingly to the operations performed by the architect. Secondly, we needed to store additional information on each structure, for instance the different complexities in each functionality redesign and the type of each local transaction. Therefore we improved the existing structure starting by the *Controller.java* class implemented in the previous work. A full preview of the updated structure is shown in figure 4.2.

We extended the controller class by introducing the *type* attribute that identifies if the controller is a saga or a query so we could calculate the corresponding metrics. Also, we added a map that contains for each cluster the entities that the given controller accesses inside it called *entitiesPerCluster* and a list of functionality redesigns. The map *entitiesPerCluster* associates a cluster to a set of entities and is necessary to the metrics calculation. Clusters and entities are identified by a unique numeric id and stored as a short to optimize the use of memory, for when the traces are very long.

The *FunctionalityRedesign* class corresponds to a possible execution flow that the user creates for the given controller. All controllers have an initial functionality redesign which is the monolith execution flow as described in the initial graph definition at section 3.1. This initial redesign is formed by translating the DAG created for each controller during the creation of their decomposition into the structure presented - the list of *LocalTransaction* objects connected through the remote invocations list that each stores. Since it is possible to have multiple redesigns for the same functionality, each controller can be

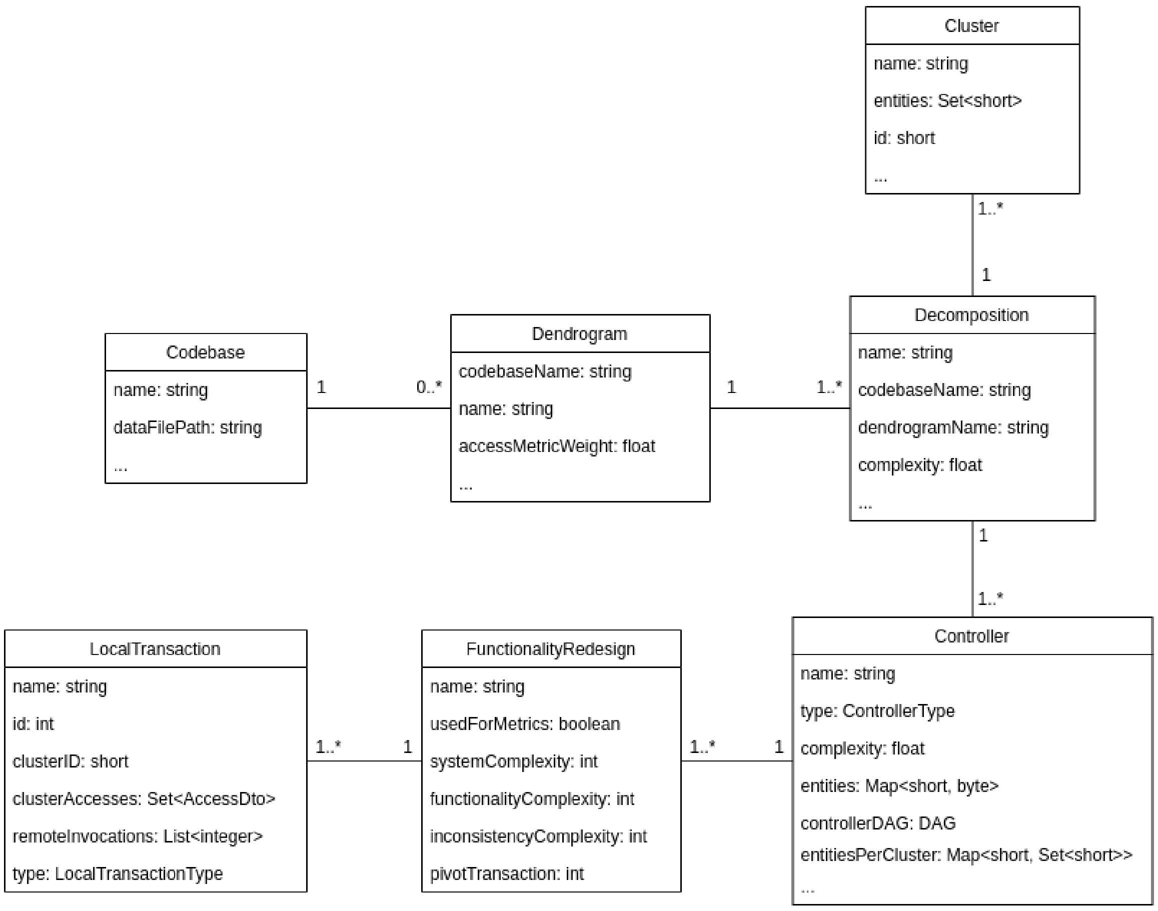


Figure 4.2: UML class diagram of the existing Mono2Micro system.

associated with various *FunctionalityRedesign* objects. On the other hand, a functionality redesign is primarily composed of several instances of the *LocalTransaction* class that contain its remote invocations.

Each functionality redesign constitutes a functionality execution graph, as defined in the functionality redesign section, and is represented by the *FunctionalityRedesign.java* class. For each functionality redesign we have:

- A string name that identifies the redesign.
- A boolean attribute that establishes if the redesign must be used in the metrics calculation when a controller has several different redesigns. It is necessary because one controller execution flow has impact on the others controllers metrics, so we need to know which redesign the user wants to use for the metrics calculation.
- A list of instances from the class *LocalTransaction* that together construct an execution graph.
- If the controller is a saga then we have two integers, one for each new metric, the Functionality

Complexity and the System Added Complexity. On the other hand if the controller is a query then we have an integer for the query metric, the Query Inconsistency Metric.

- A integer that identifies the pivot transaction within this redesign. The possible values are the local transactions' unique identifiers. This is only applicable if the controller is a saga, because there is no pivot transaction in a query.

To form the functionality redesign graph one functionality redesign is mainly composed by local transactions that store the other local transactions id with which has remote invocations. Each local transaction is composed by:

- The string *name* that the user can change through the change name operation.
- The integer *id* is a unique identifier that identifies one local transaction.
- The short cluster is the cluster where the local transaction is executed.
- The set *clusterAccesses* is the set of entities accessed by the local transaction, already pruned. Corresponds to a subsequence from the functionality access sequence and is stored as a set of *AccessDto* because that is the format that comes from the code analysis and is also the better format to send to the frontend application. An *AccessDto* stores the entity id as a short and the access mode.
- The list *remoteInvocations* is composed by other local transaction ids which are called by the local transaction, i.e., one id for each local transaction called in one remote invocation. The remote invocations dictate the local transactions execution order such that if a local transaction A has inside its *remoteInvocations* list the id of local transaction B then A is executed before B.
- The attribute type is the local transaction type and can be *compensatable*, *pivot* or *retriable* as previously described.

To resume, we have implemented the functionality execution graph described in section 3.1 basically as a list of *LocalTransaction* objects. Each graph is formed by iterating through the *redesign* attribute in the *FunctionalityRedesign* class that contains the set of local transactions described in the graph specification. Each *LocalTransaction* object is responsible for holding its remote invocations by storing in the list *remoteInvocations* the local transactions ids to which have a remote invocation. Therefore, the set of remote invocations described in the graph specification can be composed by concatenating the lists of remote invocations of each local transaction.

The backend is responsible for generating the initial graph, corresponding to the monolith trace, using the information collected during the monolith analysis. When requested by the frontend, the backend sends the graph but all changes performed during the redesign process are completed in the backend.

The frontend only needs to send the arguments necessary for each operation back to the backend, that executes the operation and returns the new graph, modified with the operations effects. Next, we will explain what each proposed operation does to the presented data structures. To help to understand, each operation has a listing with the pseudo-code of each operation.

4.2 Operations effects in the Data Structures

To perform the Add Compensating operation the user must start by selecting the local transaction that will call the new compensating local transaction and the set of entities that will be accessed. The frontend must send to the backend the caller id and the entities chosen by the user, and this will follow the algorithm in 4.1. It will search the local transaction with the specified id, create a new local transaction with the *retriable* type (because all compensating transactions execute after the pivot transaction) that accessess the entities sent and create a remote invocation between the caller and the new local transaction by adding the compensating id to the caller's list of remote invocations.

Listing 4.1: AddCompensation algorithm pseudo-code

```
public void addCompensating(String clusterName, String entities, String
    fromLTID) {

    integer newLTID = calculateLTID();
    LocalTransaction newLT = new LocalTransaction(newLTID, entities,
        clusterName);
    LocalTransaction fromLT = findLT(fromLTID);

    fromLT.addRemoteInvocation(newLTID);
    this.redesign.add(newLT);
}
```

To execute the Change Sequence operation the user must select the local transaction *lt* that wants to change and the local transaction that will be the new caller for the selected local transaction. The frontend performs an initial checking to verify if the caller local transaction is not in *lt* transitive closure, because if it is then a cyclic dependency would be created. After that initial check the frontend sends the two local transaction ids to the backend that follows the algorithm in 4.2. It starts by inserting a new remote invocation in the caller local transaction object by inserting the *lt* id, then searches for the previous *lt* caller and deletes the remote invocation to *lt*.

Listing 4.2: SequenceChange algorithm pseudo-code

```
public void sequence_change(String ltID, String newCallerID, String
    oldCallerID) {

    LocalTransaction oldCallerLT = findLT(oldCallerID);
    LocalTransaction newCallerLT = findLT(newCallerID);

    oldCallerLT.removeRemoteInvocation(ltID);
    newCallerLT.addRemoteInvocation(ltID);

}
```

The Define Coarse-Grained Interactions is the most complex operation to implement. The user can only select a set of local transactions that are executed inside two clusters because the frontend application filters out the local transactions that are not executed in the clusters selected by the user. After choosing the set of nodes, the frontend orders them and sends their ids to the backend that follows the algorithm in 4.3. It starts by creating two access sequences for the two new local transactions, the *from* and *to* local transaction. The access sequences of each are formed by selecting the nodes that are executed in each cluster, then the concatenation of access sequences of each node is performed followed by applying the prune function defined in section 3.1. The set of local transactions selected initially by the user is deleted and the two new local transactions are formed and added to the set of local transactions. One remote invocation is created between the two new nodes by inserting the *to* id in the set of remote invocations of the *from* local transaction. Every remote invocation that existed from a local transaction being merged is replaced by a remote invocation from the new *to* local transaction by adding the respective local transaction id to the *to* remote invocations set. To conclude the backend calculates the root node (the local transaction that calls the first local transaction called in the graph between the selected local transactions) and replaces the remote invocation that connects the node to the local transaction being removed by a remote invocation to the new *from* local transaction.

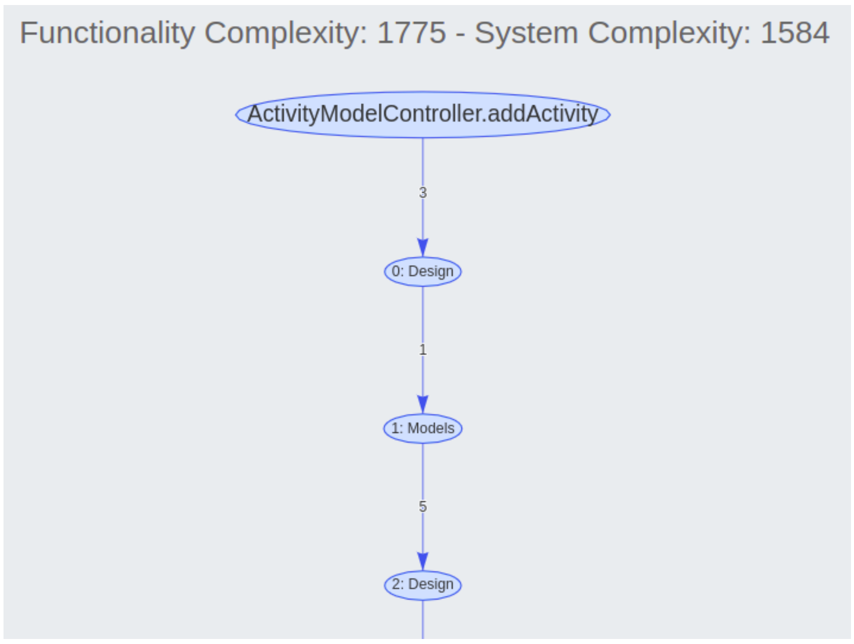
Listing 4.3: DCGI algorithm pseudo-code

```
public void dcgi(String fromCluster, String toCluster, List<String>
    localTransactions) {

    List<Strings> fromClusterLTs = filterLTs(localTransactions, fromCluster);
    List<Strings> toClusterLTs = filterLTs(localTransactions, toCluster);

    String fromLTAccessSequence = constructAccessSequenceWithPruneFunction(
        fromClusterLTs);
    String toLTAccessSequence = constructAccessSequenceWithPruneFunction(
        toClusterLTs);

    integer fromLTID = calculateLTID();
    integer toLTID = calculateLTID();

    List<integer> fromLTRemoteInvocations = new ArrayList<>();
    fromLTRemoteInvocations.add(toLTID);

    List<integer> toLTRemoteInvocations = new ArrayList<>();

    for(String ltID : localTransactions){
        LocalTransaction lt = findLT(ltID);
        toLTRemoteInvocations.addAll(lt.getRemoteInvocations());
    }

    this.redesign.removeAll(localTransactions);
    LocalTransaction root = findCaller(localTransactions[0]);
    root.remoteInvocations.remove(localTransactions[0]);
    root.remoteInvocations.add(fromLTID);

    LocalTransaction fromLT = new LocalTransaction(fromLTAccessSequence,
        fromCluster, fromLTAccessSequence, fromLTRemoteInvocations);
    LocalTransaction toLT = new LocalTransaction(toLTAccessSequence,
        toCluster, toLTAccessSequence, toLTRemoteInvocations);

    this.redesign.add(fromLT);
    this.redesign.add(toLT);

}
```

5

Visualization and Modeling Tool

The existing tool was improved as a proof of concept to incorporate the proposed operations and the metrics calculation that allow the architect to perform the functionalities redesign. The tool was implemented using the Java programming language with the Spring-boot framework for the backend, whilst the frontend was developed using ReactJS. Because we are adding new functionalities to the Mono2Micro application, we leverage on the existing work and to get to the functionalities redesign process, the tool needs as input a data collection file that represents the monolith's codebase. The codebase is obtained using static or dynamic analysis and contains the information about the invocation tree inside the system, After submitting it, a candidate decomposition can be created and the architect can see the functionalities execution flow on the given decomposition as well the new metrics values, as seen in figure 5.1.



Functionality Complexity: 1775 - System Complexity: 1584
ActivityModelController.addActivity
3
0: Design
1
1: Models
5
2: Design

Figure 5.1: Initial view presenting the monolith execution flow as well its metrics values.

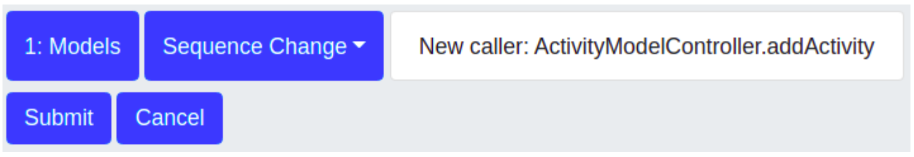
In figure 5.1 we can see at the top the values for the functionality and system complexities. The first node of an execution flow is always the controller, and each local transaction, or node, is identified by an unique id and by the cluster where it is executed. The edges correspond to a remote invocation and by hovering an edge the user can see the accessed entities in the next local transaction.

Now the architect can start the redesign process and create an execution flow he/she considers more adequate to execute in the microservices system using the design patterns. The architect has at his/her disposal five different operations:

- Sequence change - Corresponds to the homonym functionality defined in section 3.1.
- Add Compensating - Corresponds to the homonym functionality defined in section 3.1.

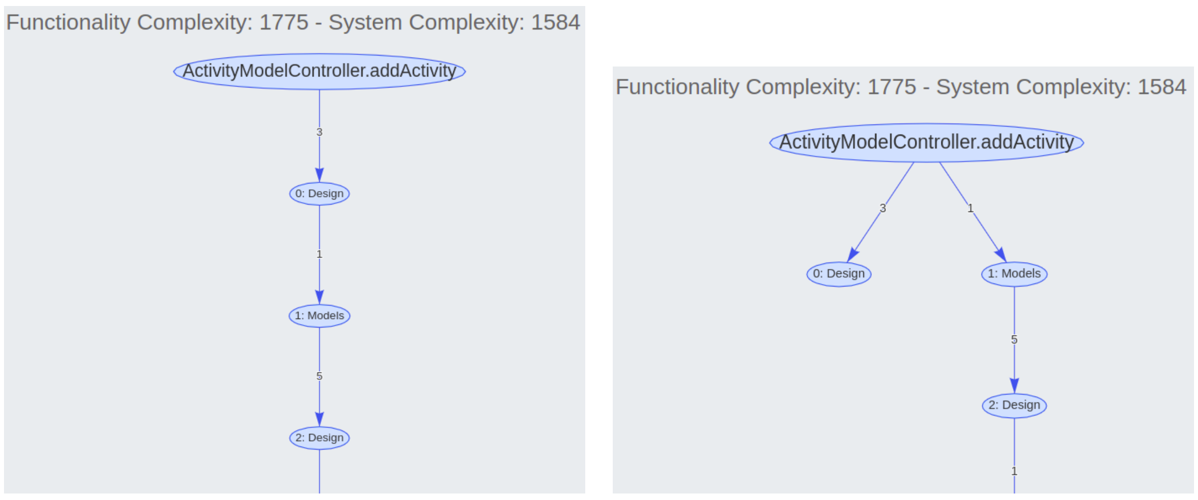
- Define Coarse-Grained Interactions - Corresponds to the homonym functionality defined in section 3.1.
- Select Pivot Transaction - This operation is applied at the end of the redesign process to incorporate the SAGA patterns into a functionality. By specifying the pivot transaction the tool is able to define what transactions are compensatable or retriable, and consequently define the semantic locks introduced by the functionality.
- Rename - This operations allows the user to change the name one local transaction has.

To execute any operation the architect must start by selecting a node from the graph that represents a local transaction. Choosing the Sequence Change operation the architect then must select another node that will be the new caller for the node selected in first place. After selecting the second node, the operation can be submitted, as seen in figure 5.2.



New caller: ActivityModelController.addActivity

Figure 5.2: Inputs necessary to execute the Sequence Change operation.


Functionality Complexity: 1775 - System Complexity: 1584
ActivityModelController.addActivity
3
0: Design
1
1: Models
5
2: Design
Functionality Complexity: 1775 - System Complexity: 1584
ActivityModelController.addActivity
3
1
0: Design
1: Models
5
2: Design
1

(a) Before the operation **(b)** After the operation

Figure 5.3: Result of applying the Sequence Change operation with the input as 5.2

After submitting the operation, the tool starts by checking if the new caller node is not executed after the first selected node. If it is, then an error message appears explaining that the operation is not possible since it would create a circular dependency in the graph. If the check is passed, the second selected node will now call the first selected node, and its previous remote invocation from its caller will be deleted. In the case presented in figure 5.2, the node *1: Models* will now be called by the node *ActivityModelController.addActivity*. The result is shown in figure 5.3, where 5.3(a) and 5.3(b) are the graph before and after the operation respectively.

To execute the Add Compensating operation, after selecting the first node that will be the caller for the new local transaction, the tool asks which cluster the new compensating local transaction will be executed. Then, the user selects the entities to be updated in the new local transaction from a set of entities, where only the entities touched in write mode by this functionality are present. After it, the user can submit the operation as presented in 5.4.

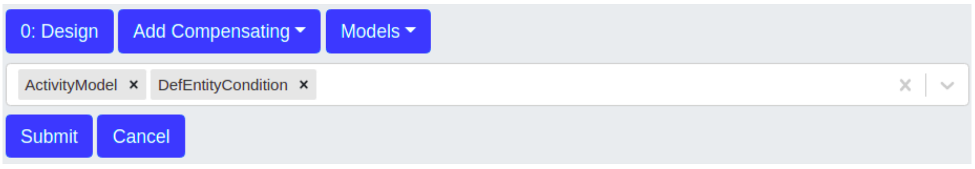
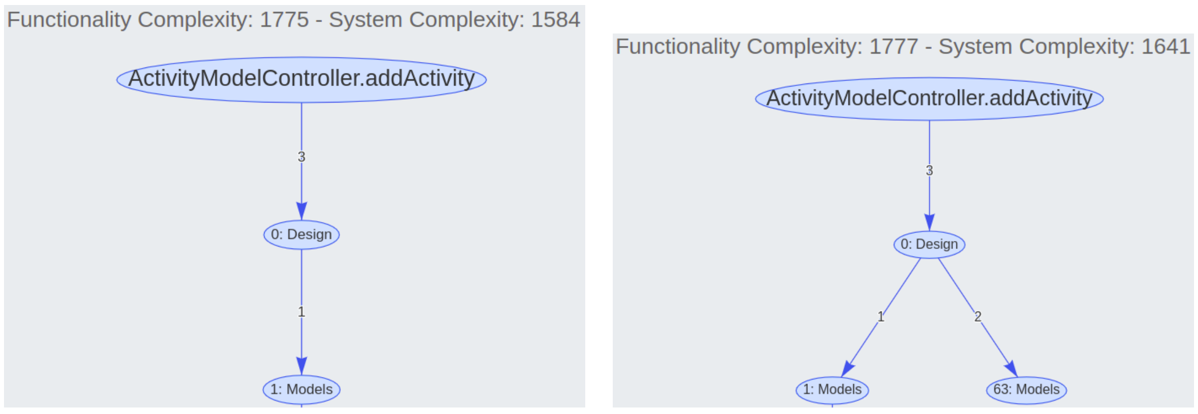


Figure 5.4: Inputs necessary to execute the Add Compensating operation.

As a result a new local transaction and a remote invocation from the first selected node to the new node are created. The result of applying the operation described in 5.4 is shown in 5.5.


Functionality Complexity: 1775 - System Complexity: 1584
ActivityModelController.addActivity
3
0: Design
1
1: Models
Functionality Complexity: 1777 - System Complexity: 1641
ActivityModelController.addActivity
3
0: Design
1
2
1: Models
63: Models

(a) Before the operation

(b) After the operation

Figure 5.5: Result of applying the Add Compensating operation with the input as 5.4

When executing the Define Coarse-Grained Interactions, two clusters must be selected. The first cluster is the cluster where the first selected node executes, whilst the second node is selected by the user from a list. After selecting the two clusters, a user can select the local transactions it wants to merge from a list which contains only local transactions executed in one of the selected clusters, or simply by clicking in the nodes. After selecting the local transactions the operation is ready to be submitted as shown in 5.6.

As a result, all the local transactions that are executed in one cluster are merged forming two new local transactions, one per selected cluster, and one remote invocation between them. The result of

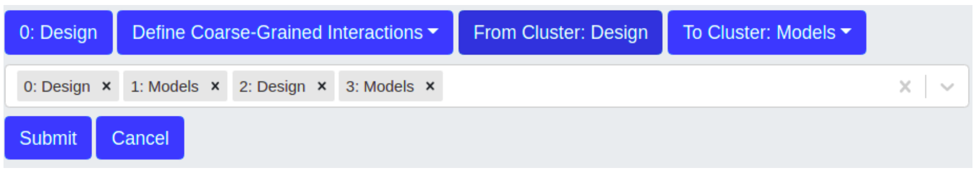
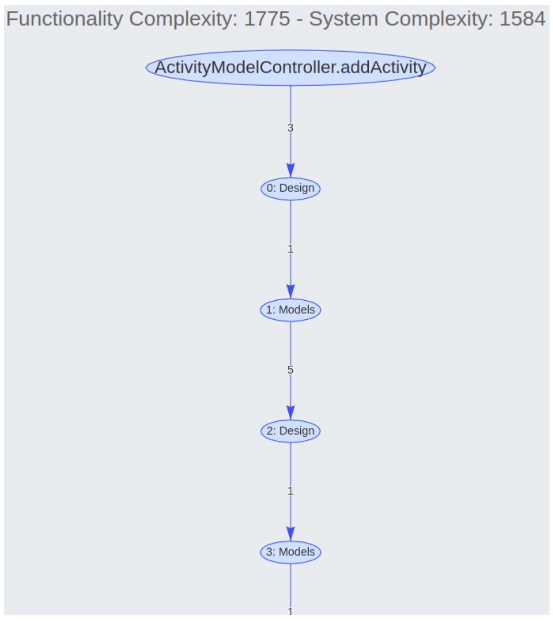
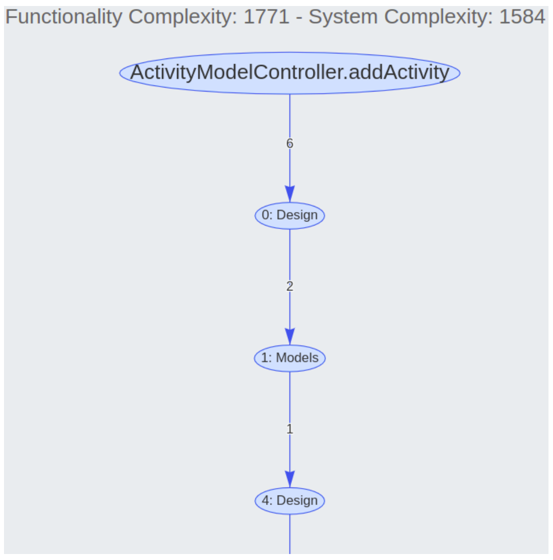


Figure 5.6: Inputs necessary to execute the Define Coarse-Grained Interaction operation.

applying the operation described in 5.6 is presented in 5.7.


Functionality Complexity: 1775 - System Complexity: 1584
ActivityModelController.addActivity
3
0: Design
1
1: Models
5
2: Design
1
3: Models
1

(a) Before the operation


Functionality Complexity: 1771 - System Complexity: 1584
ActivityModelController.addActivity
6
0: Design
2
1: Models
1
4: Design

(b) After the operation

Figure 5.7: Result of applying the Define Coarse-Grained Interaction operation with the input as 5.6

The last operation from the redesign process that the user can select is the Select Pivot Transaction operation. When selecting this operation the user is submitting to the system the transaction that will be the pivot transaction and the system calculates which transactions are compensatable or retriable. This should be the final operation to apply in the redesign process and after submitting the operation the user is presented with a menu of redesigns that the system contains for the selected functionality. Before submitting it the user is requested to write the name that will identify the redesign. In figure 5.8 we can see an example of this operation being used.

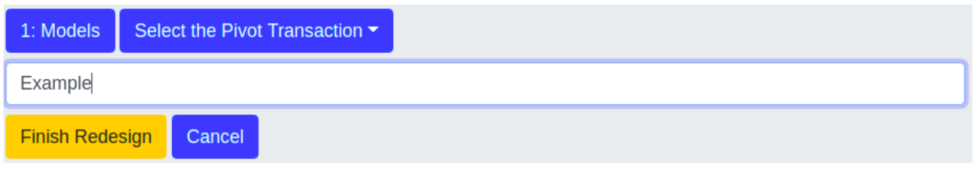


Figure 5.8: Select the Pivot Transaction operation example.

The tool also allows to compare two different redesigns for the same functionality, which is helpful because it allows the user to have a close look at the differences between two redesigns and consequently, to make a better decision about the chosen redesign. In the comparing view, presented in figure 5.9, the selected redesigns are shown side-by-side with their functionality and system complexities values if the functionality is a saga or with the query inconsistency complexity if it is a query. The user can navigate the execution flow of each one and compare them.

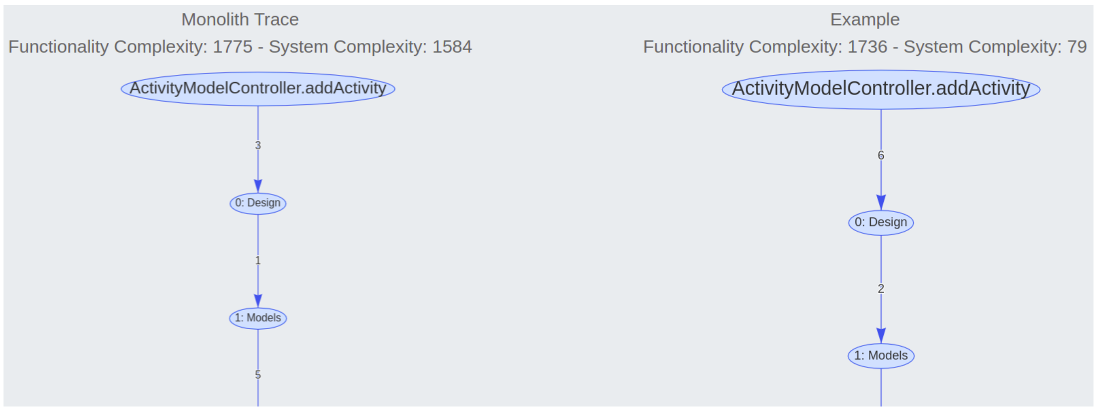

Monolith Trace
Functionality Complexity: 1775 - System Complexity: 1584
ActivityModelController.addActivity
3
0: Design
1
1: Models
5
Example
Functionality Complexity: 1736 - System Complexity: 79
ActivityModelController.addActivity
6
0: Design
2
1: Models

Figure 5.9: Comparing redesigns view.

To calculate the metrics values, the tool needs to know which redesign to use. When there is only one redesign for a functionality there is no problem because the tool uses it, the monolith execution trace, to calculate the metrics. But when there are multiple redesigns the tool must know which redesign to use to calculate the functionalities new metrics values. Therefore, if there are multiple redesigns for a functionality, the tool allows the user to select the redesign it must use in the metrics calculation. Additionally, the tool also allows the user to delete a redesign.

6 Evaluation

Contents

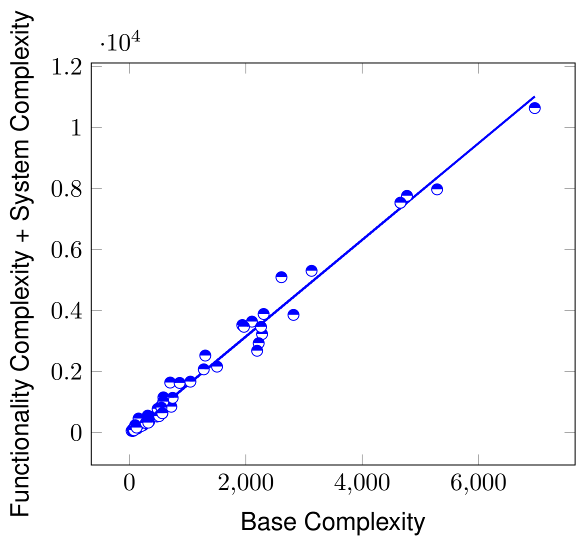
To evaluate the operations and metrics presented we analyzed two systems: LdoD (122 controllers, 67 domain entities) and Blended Workflow[1] (98 controllers, 49 domain entities)[2]. Both systems are a client-server application that use the Model-View-Controller (MVC) architecture, constructed using the Java programming language and the Spring-Boot framework, where each controller correspond to a functionality.

The set of redesign operations were defined, and formalized, after an extensive experimentation that identified which changes have to be applied to the decomposition of a functionality to create a suitable SAGA implementation, while preserving its semantics. Note that the operations were constructed to implement the SAGA pattern but they can also be applied to use the API Gateway pattern since it is only necessary to create a local transaction in each service that the query access.

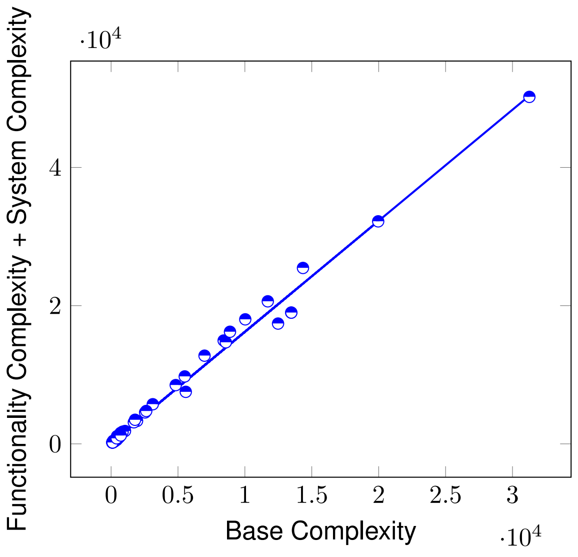
Since the operations and metrics are applied in the context of a candidate decomposition, any decomposition would serve. For our analysis we used the expert decompositions of these systems. As the main goal of this work is to refine the existing complexity metric we start by showing that the base metric and the new metrics are correlated, when applied for the initial graph where every local transaction is typed as compensatable. If the new metrics and the base metric are correlated then we can leverage on the previous work and in the analysis performed on the base metric. In figure 6.1 we can observe the correlation graphs, for the sagas functionalities, where each point represent for one functionality its values according to the base metric and to the sum of the refined metrics, Functionality Redesign Complexity (FRC) and System Added Complexity (SAC). It can be observed that the metrics are correlated.

To validate the proposed operations and the new set of metrics we need to analyse a set of functionalities. However, applying the operations to compose a new execution trace is a manual time intensive process that needs to be done by the systems experts. Therefore we started by filtering the functionalities in each system. The goal is to have two sets of functionalities, one with the functionalities that perform some create, update or delete operation (CUD operations), i.e, functionalities that write domain entities and that will be implemented using the *SAGA* pattern, and another with the functionalities that only read entities, i.e, functionalities that are queries and which implementation is done using other type of patterns, e.g. *API Gateway* pattern. Then, for the CUD system, we performed a quartile analysis over the complexities where we got 4 distinct groups of functionalities, grouped by their complexity. We randomly picked three functionalities from each group and after careful analysis of the source code we applied the operations to redesign the functionality for the given decomposition. This process allow us to analyse different functionalities with different complexities that are a representation of the entire application. The redesign goal was done to achieve a saga orchestration style as recommended in [8], to minimise the remote invocations between services and reduce the network latency effect. As to the read set, because it is a simpler process where the new query metric can be calculated automatically by the

[1] https://github.com/socialsoftware/blended-workflow
[2] https://github.com/socialsoftware/edition



(a) LdoD



(b) Blended Workflow

Figure 6.1: Correlation between the base metric and the sum of Functionality complexity and System complexity metrics.

tool we evaluate every query, compare the new values to the base metric and how it is related to the number of accessed clusters.

6.1 Operations Evaluation

Our main goal is to be able to apply the new metrics to the functionalities' traces. Therefore we created the set of operations that allow us to modify the monolith execution trace into a microservices distributed execution trace, such that after it, the metrics can be applied. Hence, we need to assess whether the operations are adequate to carry out this transformation.

In tables 6.1 and 6.2 are the operations evaluation performed in 12 functionalities for the LdoD and Blended Workflow systems respectively. For each of the selected functionalities the number of transactions of each type, the total number of local transactions and the total number of accessed clusters are presented for the final execution graph. Additionally, it presents the sum of the two complexity metrics, for the initial and final graph. Firstly, we can observe that all the final complexity values are lower than its initial values, which show us that the operations can be applied to every functionality and permits the complexity refinement. Secondly, we can observe that to preserve the data dependencies in the functionality, it is not possible to apply the operations until the number of local transactions is equal to the number of accessed clusters.

Would be expected that more complex functionalities needed more local transactions. While in gen-

	Local Transactions					Metrics	
Functionality	C	P	R	Total	# Access Clusters	Sum for the g_I	Sum for the g_F
Q1: confirmRegistration	1	1	1	3	2	173	64
Q1: approveParticipant	3	1	0	4	3	213	147
Q1: submitParticipation	2	1	1	4	2	321	87
Q2: createTopicModelling	0	1	4	5	3	343	143
Q2: removeTweets	0	1	3	4	4	442	82
Q2: getTaxonomy	0	1	4	5	3	529	192
Q3: deleteTaxonomy	0	1	2	3	4	1014	164
Q3: mergeCategories	0	1	7	8	3	1671	253
Q3: dissociate	1	1	12	14	4	2075	489
Q4: createLinearVirtualEdition	5	1	2	8	3	3467	609
Q4: associateCategory	7	1	4	12	4	3470	1067
Q4: signUp	0	1	4	5	4	3861	376

Table 6.1: Local transactions types in the functionalities of LdoD. C - Compensatable; P - Pivot; R - Retriable; # Accessed clusters - number of accessed clusters.

	Local Transactions					Metrics	
Functionality	C	P	R	Total	# Access Clusters	Sum for the g_I	Sum for the g_F
Q1: createActivity	0	1	2	3	3	302	130
Q1: createEntityDependence Condition	0	1	1	2	2	410	195
Q1: updateView	2	1	0	3	3	415	257
Q2: associateDefPathCondition ToGoalAct	2	1	0	3	2	750	400
Q2: createSpec	0	1	1	2	2	801	79
Q2: removeSequenceCondition ToActivity	2	1	3	6	2	1110	455
Q3: cleanGoalModel	1	1	3	5	4	1759	534
Q3: addSequenceConditionTo Activity	1	1	1	3	2	1860	489
Q3: addActivity	6	1	2	9	3	3323	1493
Q4: mergeGoals	16	1	2	19	4	12764	2781
Q4: extractProductGoals	9	1	2	12	3	14699	2269
Q4: extractActivity	25	1	3	29	4	20628	5636

Table 6.2: Local transactions types in the functionalities of Blended Workflow. C - Compensatable; P - Pivot; R - Retriable; # Accessed clusters - number of accessed clusters.

eral the presented values that may suggest it, there are a few cases like the *signUp* and the *createLinearVirtualEdition* functionalities that can be considered outliers. These cases show us that the correlation between initial complexity and amount of local transactions is not linear and that the number of local transactions is also dependent on the business logic of each functionality.

In what concerns the local transactions types, one clear and obvious conclusion, since the complexity depends on the number of local transactions, is that the sum of the refined metrics increases with the number of transactions. We can also observe that the number of compensatable transactions impacts on the complexity. This is due to the fact that the existence of compensatable transactions involves the creation of semantic locks (if the access mode is write) and also the creation of more transactions to implement the compensating transactions logic needed in case of a transaction abort.

6.2 Sagas Complexity Metrics Evaluation

Tables 6.3 and 6.4 show the complexities for each functionality analyzed in the systems LdoD e Blended Workflow respectively. We can observe that for both systems the reduction in the functionality complexity surpasses, in average, 40%. This shows the advantage of the proposed redesign operations and the refined metrics. Additionally, we also observe the relation between the functionality *associateCategory* and *signUp*. Before redesign the *associateCategory* has a lower complexity value than *signUp*. However after the redesigning, and the application of the *SAGA* pattern, that relation is reversed and the complexity value for the *associateCategory* is higher than for *signUp*. This shows that the impact of the redesign operations is not equal and depends on the functionality business logic. Additionally, it shows an advantage of allowing the software architect to redesign the trace that results from the automatic decomposition of the monolith, in particular the verification of whether the most complex functionalities, according to the base metric, can or not be significantly reduced.

Functionality	Initial FRC	Final FRC	% Reduction	Inital SAC	Final SAC	% Reduction
Q1: confirmRegistration	67	64	5 %	106	0	100 %
Q1: approveParticipant	190	147	23 %	23	0	100 %
Q1: submitParticipation	169	87	49 %	152	0	100 %
Q2: createTopicModelling	157	143	9 %	186	0	100 %
Q2: removeTweets	134	82	39 %	308	0	100 %
Q2: getTaxonomy	317	192	39 %	212	0	100 %
Q3: deleteTaxonomy	253	164	35 %	761	0	100 %
Q3: mergeCategories	453	253	44 %	1218	0	100 %
Q3: dissociate	772	489	37 %	1303	0	100 %
Q4: createLinearVirtualEdition	1790	383	79 %	1677	226	87 %
Q4: associateCategory	1803	662	63 %	1667	405	76 %
Q4: signUp	1490	376	75 %	2371	0	100 %
		Average:	41 %			97 %

Table 6.3: Functionality complexity and System complexity for the functionalities in the LdoD system. FRC - Functionality Redesign Complexity; SAC - System Added Complexity.

By analysing the SAC values, in both system we got a significant reduction in the complexity values after the redesign, which allows us to provide the architect with more precise values on the impact the functionalities redesign has on the system. For instance, only the functionalities *associateCategory* and *createLinearVirtualEdition* have a non zero value in the LdoD system, which indicates that only these functionalities, of the functionalities analyzed, introduce complexity into others functionalities redesign

Functionality	Initial FRC	Final FRC	% Reduction	Inital SAC	Final SAC	% Reduction
Q1: createActivity	164	130	21 %	138	0	100 %
Q1: createEntityDependence Condition	260	195	25 %	150	0	100 %
Q1: updateView	204	134	34 %	211	123	42 %
Q2: associateDefPathCondition ToGoalAct	318	214	33 %	432	186	57 %
Q2: createSpec	632	79	88 %	169	0	100 %
Q2: removeSequenceCondition ToActivity	861	376	56 %	249	79	68 %
Q3: cleanGoalModel	1155	456	61 %	604	78	87 %
Q3: addSequenceConditionTo Activity	1324	279	79 %	536	210	61 %
Q3: addActivity	1775	721	59 %	1548	772	50 %
Q4: mergeGoals	8471	1474	83 %	4293	1307	70 %
Q4: extractProductGoals	9573	1072	89 %	5126	1197	77 %
Q4: extractActivity	13849	2930	79 %	6779	2706	60 %
		Average:	58 %			73 %

Table 6.4: Functionality complexity and System complexity for the functionalities in the Blended Workflow system. FRC - Functionality Redesign Complexity; SAC - System Added Complexity.

despite that in the initial monolith trace all functionalities had a non zero value. A strong example is the functionality *signUp*, which at the beginning was considered to have the most impact on the system, but ended up having no impact on the system since it does not create any semantic locks. We can conclude that with these new metrics we can offer to the user more information and more precisely about the affected areas in the system, since the refined metrics separate the base metric into two different concerns - functionality complexity that translates the complexity introduced in a business functionality and system added complexity that translates the complexity introduces to the system by the redesign of one business functionality.

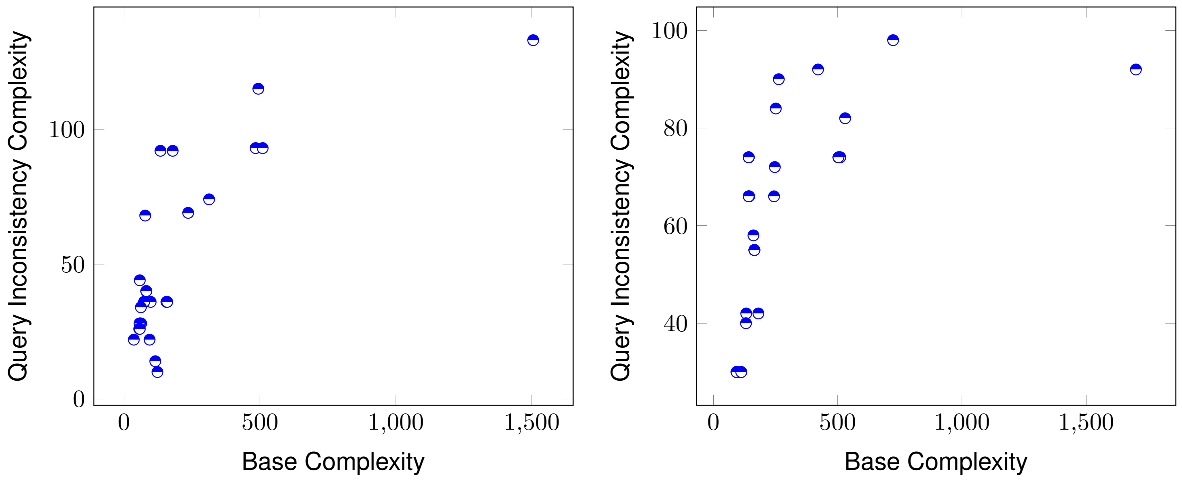
We can analyse the different results in the SAC metric we got in the two different systems. In the LdoD system only two functionalities have a non zero SAC value, while in the Blended Workflow system only three functionalities have a zero value. This difference can be explained by the source code complexity of each application. The Blended Workflow system has a more complex code, where various conditions must be met at different stages of each business functionality, that imposes the need to create various compensating and compensatable transactions with semantic locks to safeguard against any error that might occur while executing one saga. On the other hand, the LdoD system has a lower source code complexity where, in most cases, to execute a functionality no conditions must be verified, or when there are conditions that must be guaranteed they can be confirmed at the beginning of the saga. This lower source code complexity makes possible to implement a saga with no, or very few, compensatable transactions and semantic locks, that in turns results in a low SAC value. We can conclude from this analysis that is important to have different systems with different code complexities because their functionalities will also have different complexities which can be captured by the new presented metrics. Hence, by comparing the end values for both metrics for a set of functionalities from different applications we can also verify that the new metrics can show, with some certainty, what system have a lower transition cost.

When analyzing together all tables from the operations and metrics analysis, it is visible the relation between the existence of compensatable transaction and a positive value for the SAC. There are some exceptions, like the functionality *approveParticipant* in the LdoD system that contains 3 compensatable transactions and 0 system complexity. After analysing the final redesign graph we noted that the 3 compensatable transactions were read only transactions. They are considered compensatable because, by definition, all the local transaction that do not occur after the pivot transaction are compensatable, but in this case they do not need a compensating transaction in case of a transaction abort. However, as previously noted in the relation between the number of compensating transactions and complexity, we can conclude that most of the compensating transactions require semantic locks, which increases the functionality complexity. The Blended Workflow system is one clear example of this correlation, where we can see that the only functionalities that have a zero final system added complexity are also the functionalities that do not have compensatable transactions.

By analysing these 24 controllers and the values obtained we can conclude that our metrics clearly offer more precise information about the complexity in general and about the systems parts that are affected during one functionality redesign. While before the redesign the new metrics do not offer new additional information comparing with the base metric, by capturing new information during the redesign process by using the proposed set of operations we can clearly reduce the functionalities complexity and also offer distinct information about the complexity location.

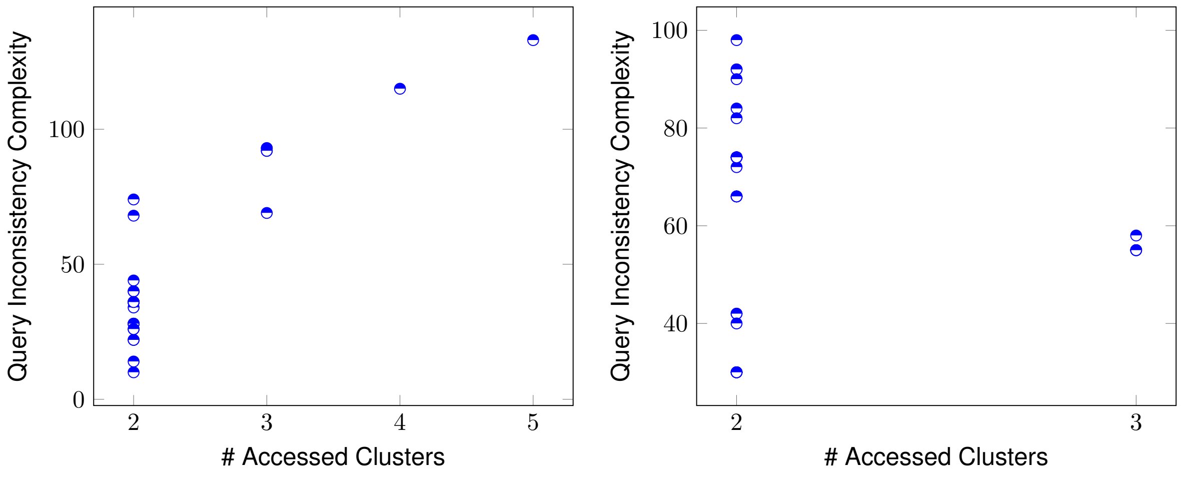
6.3 Queries Complexity Metric Evaluation

The new Query Inconsistency Complexity (QIC) metric is not derived from the base metric, however we evaluate how both are correlated for both systems and the results can be seen in figure 6.2. It is visible that the new metric is not highly correlated to the base metric as are the sagas' new metrics. This was expected because QIC is a brand new metric that was developed to captured the specific complexity of applying the pattern API Gateway. Nevertheless, by observing both graphs, it seems to exist a saturation line that limits how QIC can grow in comparation with the base metric. This is explained because the amount of accessed clusters by each query limits the QIC values while in the base metric its values are dependent on the amount of read and writes performed in a entity, which can grow more linearly. For instance, if one query only accesses two clusters then for each saga that accesses the same two clusters it will only count that amount of functionalities, while in the base metric it counts the number of writes performed in the entities, which is in almost all cases bigger.



(a) LdoD **(b)** Blended Workflow

Figure 6.2: Correlation between the base metric and Query Inconsistency Metric.



(a) LdoD **(b)** Blended Workflow

Figure 6.3: Correlation between the number of accessed clusters and Query Inconsistency Metric.

Because the number of clusters might limit the Query Inconsistency Metric we also analyse how both correlate and the results are visible in figure 6.3. We have got two different results, one for each system. On one hand, in the LdoD system there is some correlation between the number of accessed clusters and the QIC values. On the other hand, in the Blended Workflow system there is no correlation between the values. One aspect that is important to note is that the LdoD system is partitioned in five services and that are queries that accesses different amount of clusters, while the Blended Workflow system is only partitioned in four services and there is only queries accessing two or three clusters. This limitation in the variety of accessed clusters in the Blended Workflow application might be affecting the results since we do not have a wide spectrum of values. Nevertheless, if we focus on the LdoD application then as the number of accessed clusters grow the QIC values also grow, which shows that the new metric provides coherent values since as the number of accessed clusters increase the probability of encountering inconsistent states in different services also increase. In real life applications that can contain hundreds of microservices it is expected that the QIC metric provides coherent value for the inconsistency complexity in queries.

6.4 Summary

In summary, our analysis has shown that:

- The new presented metrics are highly correlated with the base metric, which allows us to leverage on the previous work.
- Our proposed set of operations can be applied to create a distributed transactions following the SAGA pattern.
- Our proposed set of operations help refine and reduce the complexity of each functionality.
- After the redesign, the new metrics can better identify functionalities that have a high complexity.
- When analysing the whole set of functionalities from different systems we can analyse and conclude what systems are more complex to transit.
- The new proposed metric for queries reflect with some certain the probability that each query might encounter when accessing different services.

Going back to the research questions raised:

1. What set of operations can be provided to the architect such that the functionalities can be redesigned by applying microservices pattern?
2. Is it possible to refine the complexity value associated with the monolith migration when there is additional information about the functionalities redesign?

To answer them, first we have defined a suitable set of operations that the architect can use in the design stage in order to design functionalities in a microservices architecture. Secondly, by separating the base metric in two distinct metrics we can target different affected areas during the functionalities design and implementation, and we obtained more precise values for the functionality migration cost.

6.5 Threats to Validity

In terms of internal limitations, the use of the expert decomposition has no impact on the validation conclusions, actually, to evaluate the metrics refinement and the operations, any candidate decomposition could be used. The validation of the operations was done to a small subset of functionalities, but a systematic method to select them was chosen and functionalities with different levels of complexity were also chosen. Also, the redesign was done to follow an orchestration style for the functionalities sagas. However, considering that: (1) we are evaluating the applicability of the redesign operation; (2) evaluating whether the new metrics can provide a more precise value, this is not biased by following a orchestration style, though the complexity values could change for depending on the redesign chosen, affecting the complexity reduction. Nevertheless this is not a defect but instead an advantage of the new metrics, that can clearly differentiate the complexity of two redesigns, as it can be seen by the different values obtained before and after the redesign process.

In terms of external validity, we believe that our conclusion can be generalized to the monolith systems that were implemented using a rich domain model, which is the case of the two analyzed systems, that were implemented using fine-grained object-oriented interactions.

7

Conclusion

Contents

In this thesis we addressed the problem that arises from an architecture transition from a monolith to a microservices architecture and present a set of complexity metrics that are based on the previous developed metric to assess the migration cost. We focused on the complexity of redesign the existing business functionalities to the new microservices environment, due to the lack of isolation, and started by presenting a set of operations for the redesign process. Without applying this set of operations our new metrics could not be applied because it was possible to apply the microservices design patterns that the metrics rely on. The SAGA pattern is applied to the CUD functionalities and the number of semantic locks is used to calculate the complexity. On the other hand a new metric was developed to assess the inconsistency complexity that the queries implemented using the API Gateway pattern might encounter. By dividing the existing complexity metric into two distinct metrics, it becomes possible to distinguish between the complexity inherent to each functionality redesign, and the complexity added in the redesign of other functionalities. As a proof of concept we improved the existing tool to incorporate the functionality redesign process with the proposed set of operations and the new metrics. As result of the evaluation, we observed that through the application of the operations a suitable execution flow of the functionality, following the SAGA pattern for the CUD transactions and the API Gateway pattern for the queries, is obtained. Regarding our sagas complexity metrics evaluation, they allow us to reduce the initial complexity value and properly identify and distinguish the business functionalities that are more complex to implement in a candidate microservices decomposition. Part of this work is already published in the European Conference on Software Architecture (ECSA 2020) [11]. The code is publicly available in a GitHub repository[1].

7.1 Future Work

Since some of the initial monolith execution traces are very cumbersome and complex, reaching hundreds of nodes and edges, one possible extension to be made would be an automatic tool that gives suggestions to the user of where and what operations to apply. These tools would analyse the execution traces and detect where a specific operation can be applied making the existing tool more powerful and alleviating the user work.

Another possible extension would be the detection of dependencies in the execution flow. As it is currently, the static and dynamic analysis are not capturing the data dependencies that may exist, it is only capturing the entities accessed and the order in which they are accessed. We consider a data dependency as a link between two local transactions where the first local transaction produces some information that the second requires in order to execute. During the redesign process we had to make this analysis manually by looking at the source code and detecting whether a local transaction could

[1]github.com/socialsoftware/mono2micro

be changed by any operation such that any existing data dependence would not be broken. These restrictions should be incorporated in the system such that the user during the redesign process does not break any dependency that would alter the business functionality.

One possible solution that we could already have implemented in this work would be to have a more restricted dependencies model, where we would define a model that in the worst case would be more severe than the real system. This model would be obtained by assuming that if a local transaction writes some entity which a future local transaction reads then a data dependency exists between them. Note that in reality this dependency might not exist. Consider that the first local transaction updates argument a of entity X and the second local transaction reads the same entity X but the argument b. In this possible model a data dependency would be created when in reality there is no dependency between the local transactions. However, there are other types of dependencies that we would need to consider. In this restricted dependencies model we are only considering dependencies between local transactions that execute in the same cluster. Since in microservices different clusters manage different entities, for two different local transactions to access the same entity they must be executed in the same cluster. So we must also evaluate the dependencies that might exist between local transactions that are executed in different clusters. But the current available data given by the static and dynamic analysis does not provide enough information to detect those dependencies because we do not know what information is passed between local transactions. As we can not detect or theorize about the second types of dependencies, the more restricted dependencies model that we start considering does not cover all possible dependencies.

Having this theory as a base, a future work could be developed to improve the current system by restricting how the proposed set of operations can be applied during the redesign process having in mind the dependencies that exist in the business functionality.

Bibliography

[1] J. Lewis and M. Fowler. Microservices a definition of this new architectural term. Accessed: 15-01-2020. [Online]. Available: https://martinfowler.com/articles/microservices.html

[2] A. Kwiecień. 10 companies that implemented the microservice architecture and paved the way for others. Accessed: 15-01-2020. [Online]. Available: https://divante.com/blog/10-companies-that-implemented-the-microservice-architecture-and-paved-the-way-for-others/

[3] 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.

[4] N. Santos and A. Rito Silva, “A complexity metric for microservices architecture migration,” in *Proceedings of the IEEE 17th International Conference on Software Architecture (ICSA 2020)*. IEEE, 20202, pp. 169–178.

[5] 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

[6] Y. Al-houmaily and G. Samaras, *Two-Phase Commit*, 01 2009, pp. 3204–3209.

[7] H. Garcia-Molina and K. Salem, “Sagas,” *ACM Sigmod Record*, vol. 16, no. 3, pp. 249–259, 1987.

[8] C. Richardson, *Microservices Patterns*. Manning Publications Co., 2019.

[9] E. Ntentos, U. Zdun, K. Plakidas, D. Schall, F. Li, and S. Meixner, “Supporting architectural decision making on data management in microservice architectures,” in *European Conference on Software Architecture*. Springer, 2019, pp. 20–36.

[10] 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: Association for Computing Machinery, 2018, p. 1–6. [Online]. Available: https://doi.org/10.1145/3242163.3242164

[11] J. F. Almeida and A. R. Silva, “Monolith migration complexity tuning through the application of microservices patterns,” in *European Conference on Software Architecture*. Springer, 2020, pp. 39–54.

[12] U. Zdun, M. Stocker, O. Zimmermann, C. Pautasso, and D. Lübke, “Guiding architectural decision making on quality aspects in microservice apis,” in *International Conference on Service-Oriented Computing*. Springer, 2018, pp. 73–89.

[13] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng, “Service candidate identification from monolithic systems based on execution traces,” *IEEE Transactions on Software Engineering*, 2019.

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

[15] J. Bogner, S. Wagner, and A. Zimmermann, “Automatically measuring the maintainability of service- and microservice-based systems: a literature review,” in *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*. ACM, 2017, pp. 107–115.

[16] T. Engel, M. Langermeier, B. Bauer, and A. Hofmann, “Evaluation of microservice architectures: A metric and tool-based approach,” in *International Conference on Advanced Information Systems Engineering*. Springer, 2018, pp. 74–89.

[17] V. R. Basili and H. D. Rombach, “The tame project: Towards improvement-oriented software environments,” *IEEE Transactions on software engineering*, vol. 14, no. 6, pp. 758–773, 1988.

[18] U. Zdun, E. Navarro, and F. Leymann, “Ensuring and assessing architecture conformance to microservice decomposition patterns,” in *International Conference on Service-Oriented Computing. ICSOC. Lecture Notes in Computer Scienceg*. Springer, 2017, pp. 411–429.

[19] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, “Towards recovering the software architecture of microservice-based systems,” in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 2017, pp. 46–53.

[20] 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)*. IEEE, 2018, pp. 32–42.

[21] A. Krause, C. Zirkelbach, W. Hasselbring, S. Lenga, and D. Kröger, “Microservice decomposition via static and dynamic analysis of the monolith,” in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2020, pp. 9–16.

[22] E. Ntentos, U. Zdun, K. Plakidas, S. Meixner, and S. Geiger, “Assessing architecture conformance to coupling-related patterns and practices in microservices,” in *Software Architecture*, A. Jansen, I. Malavolta, H. Muccini, I. Ozkaya, and O. Zimmermann, Eds. Cham: Springer International Publishing, 2020, pp. 3–20.