Automated Identification of Monolith Functionality Refactorings for Microservices Migrations

José Manuel Meneses Correia

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor: Prof. António Manuel Ferreira Rito da Silva

Examination Committee

Chairperson: Prof. Teresa Maria Sá Ferreira Vazão Vasques
Supervisor: Prof. António Manuel Ferreira Rito da Silva
Member of the Committee: Prof. Carlos Nuno da Cruz Ribeiro

September 2021

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

The journey that led to this work would not have been possible without the dedication and hard work of both my parents on showing me, in different ways, what it takes to excel in life. Their example is in my mind in all the biggest decisions.

To my best friend and number one fan, my girlfriend, who was by my side from the beginning till the end, always keen to hear me talk about my research problems and accomplishments. Thank you for giving me the peace of mind to finish this thesis.

Also, and I cannot stress this enough, I want to acknowledge the incredible work that my supervisor, Prof. António Rito Silva, does with his students. From the moment I tried to have him supervise me, to all the fun productive meetings we had along the year, he has shown a dedication and an availability that is truly inspiring and made my motivation grow along the way.

In a last note, to all my friends and colleagues at Técnico that made the years in this college truly worth it.

Abstract

The process of migrating a monolith to a microservices architecture has a cost due to the refactorization of its functionalities in an eventual consistent transactional context. On the other hand, the object-oriented approach commonly followed in the development of monolith systems promotes fine-grained interactions in the functionalities implementation, which further increases the migration cost due to the large number of remote invocations between microservices. This calls must be changed to more coarse-grained interactions that increase the fault tolerance and reduce the latency of the system. In this research, we propose the addition of a new tool to the Mono2Micro system to help the software architect identify the functionality refactorings that transform several fine-grainded interactions into coarse grained ones, to ease the migration of any functionality to a SAGA pattern. We address two specific research questions: (1) Is it possible to automatically recommend refactorizations that minimize the migration effort of functionalities as *SAGA* orchestrations? (2) Can we characterize the *SAGA* orchestrators that result in a higher reduction of the migration effort? - Regarding the first question, the heuristic accuracy, and efficiency of the tool are evaluated by executing it for a dataset of 78 codebases, doing a statistical analysis of the results in terms of complexity reduction and operations performed, and then manually validating the SAGAs proposed for a particular codebase and comparing them with refactorizations made by an expert. To answer the second question, we define a set of metrics that characterize each microservice in a candidate decomposition and evaluate its correlation with the microservices orchestrators proposed by the tool in order to see if a pattern emerges.

Keywords

Monolith migration; Microservices; *Saga* Pattern; Refactoring; Heuristics

Resumo

O processo de migração de um sistema monolítico para uma arquitetura de microserviços tem um custo associado devido à refatorização das funcionalidades para um contexto distribuido com transações consistentes. Para além disto, a implementação de sistemas monolíticos promove um grande número de interações entre os diferentes módulos usados em cada funcionalidade, o que aumenta ainda mais o custo da migração, uma vez que é necessário alterar este comportamento de modo a reduzir as invocações externas entre microserviços. Nesta tese, propõe-se a adição de uma nova ferramenta ao sistema de Mono2Micro para ajudar a identificar refatorizações que diminuem a granularidade das interações entre os módulos, facilitando a migração de um dado sistema para uma arquitetura de microserviços que aplica o padrão de Sagas. Abordam-se, então, duas questões de pesquisa: (1) É possível fazer uma recomendação automática de refatorizações que minimizam o custo de migração de uma funcionalidade para um padrão de Sagas? (2) É possível caracterizar os orquestradores das Sagas que resultam numa maior redução do custo de migração? Relativamente à primeira questão, a precisão dos métodos heurísticos e a eficiência da ferramenta são avaliados executando esta última para um dataset de 78 sistemas, e efetuando a análise estatística dos resultados quanto à redução da complexidade de migração e operações efetuadas. Além disto, realiza-se uma verificação manual das implementações propostas para um sistema específico e compara-se com as refatorizaçõs feitas por um especialista. Para responder à segunda questão, definiu-se um conjunto de métricas que caracterizam cada um dos microserviços no contexto de uma decomposição, antes das refatorizações, e avaliou-se se estas métricas estão relacionadas com os orquestradores propostos pela ferramenta.

Palavras Chave

Migração de Monólitos; Microserviços; Padrão *Saga*; Refatorização; Heurística

Contents

1 Introduction **1**

1.1 Context 2

1.2 Problem 3

1.3 Research Questions 4

1.4 Contributions 4

1.5 Outline 5

2 Background and Related Work **7**

2.1 Background 8

2.1.1 Mono2Micro system 8

2.1.2 The Saga Pattern 10

2.2 Related Work 13

2.3 Conclusion 22

3 Solution **23**

3.1 Strategy 24

3.2 Functionality Migration 26

3.3 Saga Refactorization Algorithm 29

3.3.1 Flowchart Analysis 29

3.3.2 Pseudocode Analysis 31

4 Implementation **35**

4.1 Functional Requirements 36

4.2 Web Service 37

4.2.1 Architecture Overview 37

4.2.2 Execution Flowchart 39

4.2.3 Business Logic Interfaces 40

4.3 API Specification 42

4.4 Mono2Micro Integration 44

4.4.1 Deployment 45

4.4.2 User Interface 46

5 Evaluation and Analysis **49**

5.1 Complexity Reduction Analysis 50

5.2 Refactorization Accuracy Analysis 52

5.3 Orchestrators Characterization 55

5.4 Performance Analysis 56

5.5 Summary 58

5.6 Threats to Validity 59

6 Conclusion **61**

List of Figures

2.1 Architecture overview of the Mono2Micro system. 9
2.2 Workflow of creating a codebase decomposition in the Mono2Micro system. 10
2.3 Orchestration approach for the *Saga* pattern. 11
2.4 Choreography approach for the *Saga* pattern. 12

3.1 Sequence Change operation applied to a decomposition callgraph. 25
3.2 Local Transaction Merge operation applied to a decomposition callgraph. 25
3.3 Flowchart for the algorithm that computes the list of possible Saga orchestrations for a functionality. 30
3.4 Operation of adding pivot orchestrator invocations 32
3.5 Operation of merging and pruning invocations without data dependence 33
3.6 Operation of merging and pruning invocations with data dependencies, according to data dependence threshold 34

4.1 Use case diagram for the Refactorization Service. 36
4.2 Workflow from the moment a codebase is analyzed in Mono2Micro until it is refactored by the service. 37
4.3 Flowchart of the high level logic of the refactorization tool. 39
4.4 Containerized infrastructure of the Mono2Micro system integrated with the Refactorization Service. 45
4.5 Mono2Micro dashboard that is used to interact with the Refactorization Tool. 46
4.6 Dynamic table of results were each row corresponds to a functionality being refactored. . 47
4.7 Refactorization dropdown which demonstrates the proposed Saga. 48

5.1 Percentage reduction of the Functionality Migration Complexity in function of the number of merge operations made. 51
5.2 Behavior of the static analysis step when encountering conditional branching. 54

5.3 Probability of the orchestrator cluster having a read access in the initial monolithic functionality, in function of the final FMC reduction. 56

List of Tables

2.1 Benefits and drawbacks of the *Saga* pattern. 10
2.2 Benefits and drawbacks of the orchestration approach. 11
2.3 Benefits and drawbacks of the choreography approach. 13

4.1 Comparision between the behaviour of goroutines and OS threads. 38
4.2 Available endpoints of the Refactorization Service. 42
4.3 Description of the JSON fields in the body of the RefactorDecomposition request. 43

5.1 Average values for running the recommendation heuristic for 652 functionalities, considering $\delta = 1, 2, \infty$. 50
5.2 Functionality Migration Complexity reduction resulting from a refactoring using the tool with $\delta = 1$ and by an expert. 52
5.3 Correlation between the metrics for the orchestrator cluster and the reduction of the functionality migration complexity and system added complexity 55
5.4 Machines used to test the Refactorization Tool. 57
5.5 Execution times when refactoring the dataset in the **Local** machine. 57
5.6 Execution times when refactoring the dataset in the **AWS m5.xlarge** machine. 57
5.7 Execution times when refactoring the dataset in the **AWS c5.4xlarge** machine. 57
5.8 Cumulative bytes allocated for heap objects during the refactorization of the 78 codebases dataset. 58

Listings

3.1 *SAGA* estimator 31
3.2 Set Orchestrator for Partition 31
3.3 Refactor through merge of fine-grained invocations into coarse-grained. 32
4.1 Declaration of the RequestHandler interface 41
4.2 Declaration of the FilesHandler interface 41
4.3 Declaration of the MetricsHandler interface 41
4.4 Declaration of the RefactorizationHandler interface 41
4.5 Example cURL HTTP request for the ViewRefactorization endpoint 42
4.6 Example cURL HTTP request for the RefactorDecomposition endpoint 42
4.7 RefactorDecomposition successful HTTP response body 43
4.8 RefactorDecomposition error HTTP response body 44
4.9 Command used to deploy all the components of the Mono2Micro system. 46

Acronyms

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Program Interface
DPD	Design Pattern Detection
DSL	Domain-specific Language
FMC	Functionality Migration Complexity
FoSCI	Functionality-oriented Service Candidate Identification
INESC-ID	Instituto de Engenharia de Sistemas e Computadores: Investigação e Desenvolvimento
MARPLE	Metrics and Architecture Reconstruction Plug-in for Eclipse
OOP	Object-Oriented Programming
PADL	Pattern and Abstract-level Description Language
POM	Primitives, Operators, Metrics
REST	Representational State Transfer
SAC	System Added Complexity
SVM	Support Vector Machines

1

Introduction

Contents

1.1 Context 2
1.2 Problem 3
1.3 Research Questions 4
1.4 Contributions 4
1.5 Outline 5

The microservices architecture [1] allows the split of a large software development project into several small agile cross-functional teams and facilitates independent scalability of the services that constitute the product [1]. On the other hand, it is a common practice to start developing a complex system as a monolith [2] due to the shorter time to market and the fact that it is difficult to find the correct modularization of a system without doing several refactorings. Therefore, many monolith systems have to go through a migration phase to a microservices architecture when scalability and development efficiency become a bigger priority.

This migration phase is a complex task for the developers. It requires timely planning of the correct decomposition of the monolith, dividing the business logic in a way that promotes high maintainability. In a later phase, it requires a complete rewrite of the monolith business logic to deal with the break of transactional behavior into eventual consistency. Therefore, it is relevant to prepare the monolith to reduce the migration effort.

The work in this thesis aims at improving the Mono2Micro system developed in previous work by integrating it with a new service implementing an automated refactorization algorithm. This algorithm reads the candidate decomposition of a monolith into clusters, and the monolith set of functionalities, an then proposes refactorizations that prepare the monolith to be migrated to a microservices architecture according to the SAGA pattern, while reducing the migration cost. Additionally, the service indicates what should be the orchestrator cluster for each functionality.

1.1 Context

We leverage on previous work [3–5] to support the architect with a recommendation mechanism for the refactoring of monolithical codebases in the context of a candidate decomposition.

The investigation in [3] resulted in the creation of a workflow that, by using static analysis of a monolith source code, can generate a callgraph showing the domain entities accessed by each functionality of the codebase. After this, given the call graph and a similarity measure based on the domain entities accessed by each functionality, a hierarchical clustering algorithm is applied to generate candidate decompositions. Each candidate decomposition is made of clusters of domain entities that are closer according to the similarity measure. Each cluster represents a candidate microservice. The architect is then able to visualize each one of the proposed clusters in a graphic user interface.

Later, in [4] the research is extended, this time by focusing on the definition of a migration complexity metric, and improving the data collection started in the previous work by also extracting the mode of the domain entities accesses, read or write, which allowed for the definition of new similarity measures that consider the type of access.

In [5] two new metrics are proposed to measure the complexity of migrating a monolith functionality

to a microservices architecture implemented by applying the Sagas pattern. These metrics extend the previous one for the Saga context and split it into two aspects: the complexity of migrating a functionality, and the complexity it adds to the migration of other functionalities. Additionally, it was observed a code smell for migrations with high complexity. An important conclusion in this research is that the high complexity associated with the migration of a functionality is due to the number of inter-candidate microservices invocations, which increases the number of the *Saga* intermediate states. The main reason for this situation is that monoliths are implemented using a significant number of fine-grained invocations, prevalent in the Object-Oriented Programming (OOP) paradigm.

1.2 Problem

Although our previous work allowed the software architect to be informed about the cost of migrating the functionality, given a candidate decomposition, he is not aware of the possible reductions that can exist if fine-grained inter-microservice invocations are refactored into a smaller number of coarse-grained interactions. This situation is worsened because a monolith can have hundreds, if not thousands, of functionalities, making it time-consuming and impractical for a manual inspection of each functionality structure. Engineering teams often dedicate months or even years to ultimately achieve a 100% microservice migration in the production environment [6], which results in less time allocation to develop new features and improve the system's core functionalities. This issue leads us to strongly believe that an automation system should make it easier to calculate the most efficient way to achieve the end goal.

Another important aspect is that developers' decision-making is undoubtedly flawed. Sometimes, the microservices design implemented might not be the one that results in a less complex, more scalable, and maintainable system and results in a faster time-to-market due to fewer changes in the logic. Planning for a microservices architecture requires an extensive discussion of the problem and answering important architectural questions such as:

- How will communications be made between multiple clusters?
- How will distributed transactions be handled?
- How will the system be monitored?
- How will it behave in a failure scenario, and how will it recover?
- How will dependencies be managed between the different teams maintaining each service?
- Are these dependencies tightly coupled?

If planning is not thorough enough, problems will arise during the migration phase, requiring going back to the whiteboard and completely overhauling the initial design.

1.3 Research Questions

This research leverages on previous work that suceeded to extract data from monolithic codebases, generate sequential callgraphs of functionalities and define a set of complexity metrics that measure the migration complexity in terms of the cost associated with the relaxation of the transaction model. With this valuable data extraction algorithms and complexity metrics we are able to focus on the automation of the process of refactoring a monolith by reducing the number of fine-grained accesses, by the means of reads or writes, to domain entities, so that, we reduce the overall cost of the migration by proposing a way to minimize te remote invocations between clusters.

However, this attempt at recommending the best functionality refactorizations needs to be validated regarding the accuracy of the results, since it is not enough to reduce the fine-grained invocations, but we need to make sure that this changes are implementable in the original source code and do not break data dependencies between sequential domain entity accesses.

With this in mind, we evaluate the performance of the tool on reducing the system complexity, by applying it to a dataset of 78 codebases, and analyze the accuracy of the results of one codebase by comparing them to a manual refactorization by a developer. Using this methodology, we will focus on answering the following research questions:

- Is it possible to recommend the refactoring of a functionality, by merging fine-grained inter-microservice interactions into coarse-grained ones, that minimizes its migration effort as an orchestrated *Saga* in the context of the decomposition?

- Can we characterize the *Saga* orchestrators that allow for a higher reduction of the migration effort?

1.4 Contributions

The main contributions of this work are:

- A recursive algorithm that, given a candidate decomposition of a monolith and the monolith set of functionalities, outputs a refactoring for each functionality that reduces the complexity of its migration using the Saga pattern. It presents a refactored callgraph, the identification of the best fitting orchestrator, and the overall impact that the reduction of fine-grained invocations had on the overall cost of the migration.

- An easily deployable and scalable Golang service, implementing a Representational State Transfer (REST) Application Program Interface (API) to expose the algorithm.

- An integration with the Mono2Micro system, through a graphical user interface and visualization tool which can be used to interact with the Refactorization Service by requesting codebase refactorizations and analyzing the results.

1.5 Outline

This thesis is organized as follows: Chapter 2 introduces some valuable concepts for the correct analysis of this research and then presents a discussion of the state-of-the-art on the field of monolith codebase migrations and identification of code-smells, both from a heuristic and from a machine learning perspective. Chapter 3 presents the discussion around the possible approaches to solve the problem and the decision of best suiting approach. Later its presented the core algorithm that was developed to refactor monolithic functionalities, given a candidate decomposition, such that the complexity of the migration is reduced. Chapter 4 describes the web service requirements, implementation, and how this service was integrated with the Mono2Micro system. In Chapter 5 the tool is evaluated by applying it to a large dataset of 78 codebases and then narrowing the analysis to a single codebase by comparing the refactorizations proposed to the ones made by an expert. Finally, Chapter 6 concludes this thesis with final remarks, an overall summary of the work done, and a brief description of possible future research topics that can extend these conclusions.

2

Background and Related Work

Contents

2.1 Background 8
2.2 Related Work 13
2.3 Conclusion 22

This section starts by presenting a set of critical concepts essential for a better analysis of the work done and the proposed solution. These concepts involve a description of the system where the Refactorization Service will be integrated and an introduction to the Saga pattern, which will be used to create microservice orchestrations.

Further on, we present the existing research on the field of monolith to microservice refactorizations and code-smell detection, both using heuristic and machine-learning solutions while comparing the benefits and drawbacks of each approach.

2.1 Background

The solution presented in this thesis will be integrated in the Mono2Micro software system, which we briefly describe below. The imlementation of the proposed solution involves executing an algorithm to compute refactorizations for the functionalities of a codebase, in the context of a decomposition and by applying the Saga Pattern, which is the next topic to be introduced in this subchapter.

2.1.1 Mono2Micro system

In the context of previous research [3–5] the Social Software Engineering research team[1] at the Lisbon's Instituto de Engenharia de Sistemas e Computadores: Investigação e Desenvolvimento (INESC-ID) developed Mono2Micro[2], a software system that implements tools to support software architects on the task of migrating codebases from a monolithic to a microservices architecture.

This system is composed of a frontend user interface built on top of the React Javascript library[3], which communicates directly with the backend Application Program Interface (API) service, built using the Spring-Boot Java framework[4], responsible for all the computations, script executions and persisting the data in the file system. Both the UI and the backend application are deployed in a containerized Docker environment, described in Fig. 2.1.

The main features implemented in Mono2Micro include:

1. Data extraction from the source code of codebases implemented using Spring-Boot, by applying static or dynamic analysis algorithms. This analysis generates a call-graph containing the order of entity accesses and type of access (read of write) for each functionality of the codebase.

2. Clusterization of the domain entities using a hierarchical clustering algorithm[5] implemented by Scipy Python library, which generates a dendrogram, as described in [3].

[1] https://socialsoftware.github.io/
[2] https://github.com/socialsoftware/mono2micro
[3] https://reactjs.org/
[4] https://spring.io/projects/spring-boot
[5] https://docs.scipy.org/doc/scipy/reference/cluster.hierarchy.html

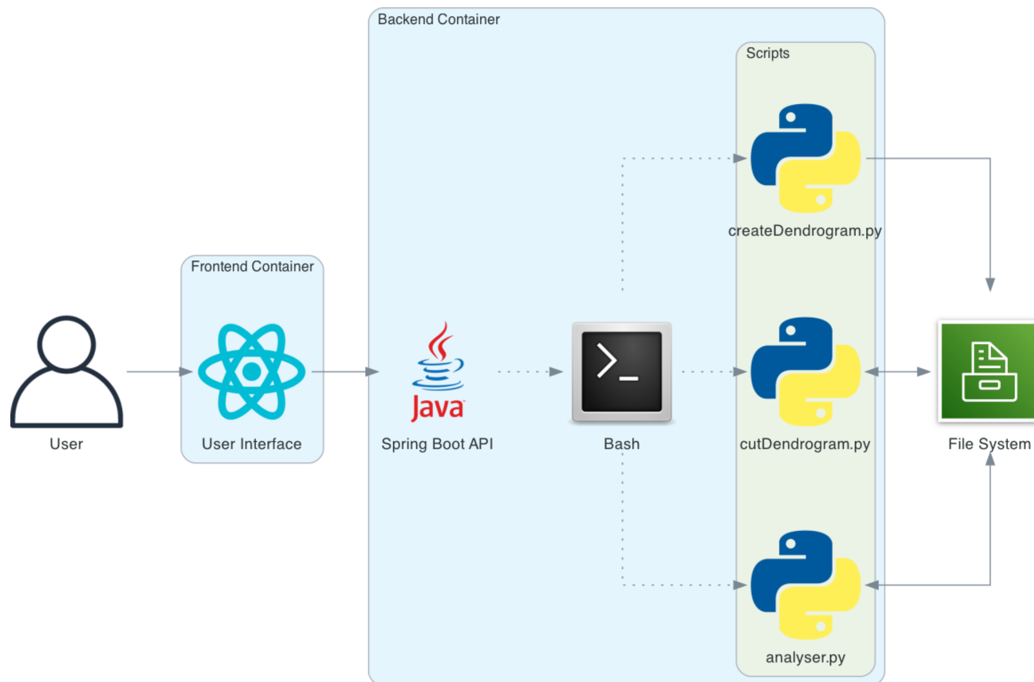

Figure 2.1: Architecture overview of the Mono2Micro system.

3. Manually cutting the dendrogram into a set of different clusters, to create a decomposition, as shown in Fig. 2.2. The architect can do this by choosing one of two distinct methods:
   - Choosing the total number of clusters that the cut must have.
   - Choosing the maximum distance between domain entities inside each cluster.
4. Visualizing complexity metrics that enable the architect to assess the quality of the decomposition created.
5. Experimenting with the clusters by allowing the architect to rename, merge and split clusters, as well as move data entities between them.
6. Refactoring any functionality of the codebase by applying a set of refactorization operations, including: Add Compensating, Sequence Change, Local Transaction Merge and Define Coarse-Grained Interactions, described in [5].

The solution presented in this research thesis has the purpose of being integrated into the Mono2Micro system in order to make it more feature complete, so that developers can have a recomendation system for refactoring the decompositions that they create, with a low effort, and while being able to analyse them in a rich user interface.

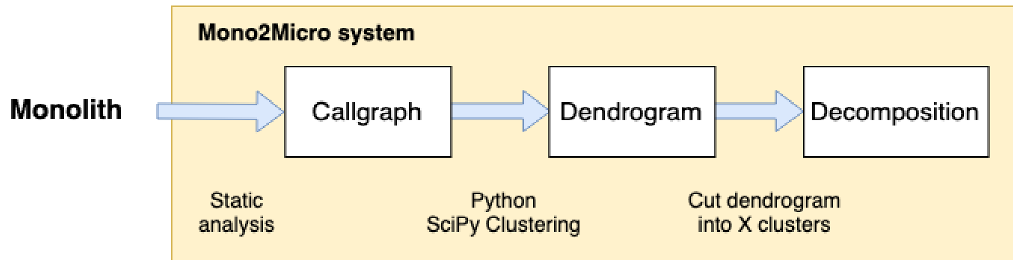

Figure 2.2: Workflow of creating a codebase decomposition in the Mono2Micro system.

2.1.2 The Saga Pattern

The traditional solution for handling data operations in a distributed system spanning multiple services is known as distributed transactions. In this approach, the services provide data-changing resources, and a manager is responsible for invoking each one of the services and managing the global state of the transaction, making sure that the Atomicity, Consistency, Isolation and Durability properties are used. This approach has significant drawbacks when it comes to maintaining data consistency across all services. For instance, if the local transaction in one of the services fails, the manager must make sure that this is reflected in the data state of all the other services. Another drawback is visible when accounting for service availability since for a distributed transaction to be successful, all the services must be available, which creates tight coupling between all the participants.

The so-called gray literature proposes the use of the *Saga* pattern to implement business functionalities in the microservices architecture [7] in order to maintain data consistency across the multiple services and without using distributed transactions. This pattern is based on the seminal work by Hector-Molina and Kenneth [8] and addresses the lack of isolation between the code modules due to the creation of intermediate transactional states that are visible outside the scope of the functionalities execution.

Each local transaction updates data within a single service using the familiar Atomicity, Consistency, Isolation, Durability (ACID) transaction frameworks and then publishes a message or event to trigger the next local transaction in the Saga. If a local transaction fails because it violates a business rule, then the *Saga* executes a series of compensating transactions that undo the changes that were made. In Table 2.1 we present the main benefits and drawbacks of using the *Saga* pattern in a distributed system.

Table 2.1: Benefits and drawbacks of the *Saga* pattern.

Benefits	Drawbacks
Maintains data consistency across services	The programming model is more complex
Promotes loose coupling	More difficult to debug
Supports long-lived transactions	Developers must design compensating transactions which undo changes made

There are two main approaches for designing a *Saga* pattern when it comes to the service's roles and protocols of communication between them: orchestrations and choreography. In an orchestration, which is presented in Fig. 2.3, an orchestrator informs the participants about which local transactions should be executed. The orchestrated cluster performs the local transaction and reports back to the orchestrator, informing if it was completed or not. This communication can be done by any regular HTTP implementation or by an asynchronous event-producer and event-consumer design.

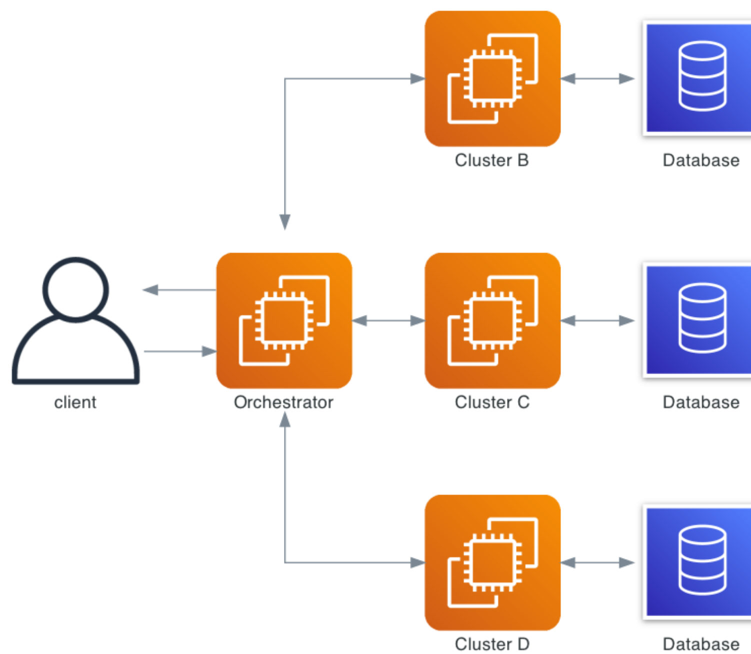

Figure 2.3: Orchestration approach for the *Saga* pattern.

In Table 2.2 we present the benefits and drawbacks of an orchestration, which were used to decide the approach to follow in the solution implementation.

Table 2.2: Benefits and drawbacks of the orchestration approach.

Benefits	Drawbacks
Suitable for complex workflows where new participants can be added anytime	Additional complexity, since it requires implementing coordination logic in the orchestrator
It does not introduce cyclical dependencies	Single point of failure since the orchestrator manages the whole workflow
The participants do not need to know about commands for other participants	
More separation of concerns, which simplifies the business logic	

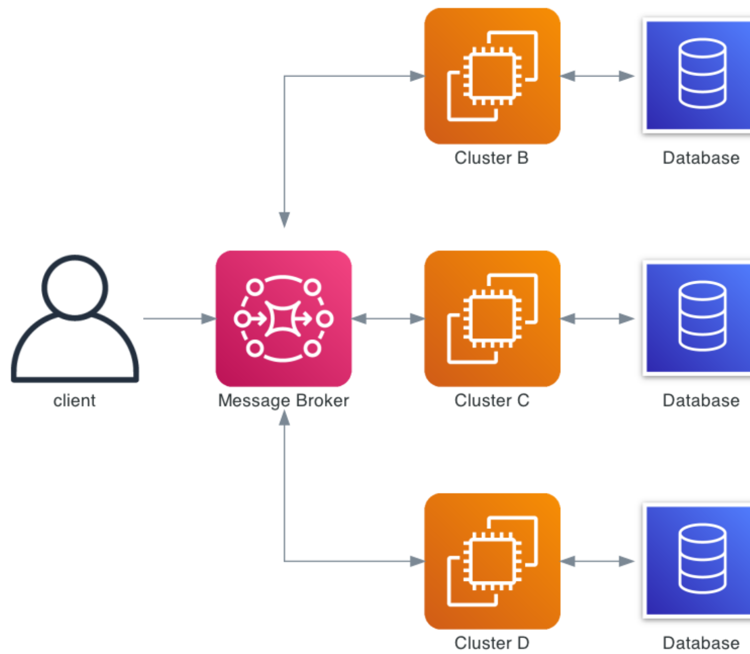

Figure 2.4: Choreography approach for the *Saga* pattern.

In a choreography, Fig. 2.4, in opposit to the orchestration, there is not a *Saga* manager cluster which triggers and coordinates the execution.

Instead of relying on an orchestrator cluster, the *Saga* is triggered by an event published to a message broker. Participants configured to consume that event will then trigger their local transactions, and upon completion, publish an event back to the message broker, which will trigger local transactions in other participants. This pattern goes on until all the required local transactions in the *Saga* are completed or until one fails and a compensating transaction is initiated.

In Table 2.3 we present the benefits and drawbacks of a choreography, which were used to decide the approach to follow in the solution implementation.

For this thesis we will consider the orchestration approach, since it is the most generic one that can be applied to more complex workflows. Additionally, we have datasets of refactorizations by human experts that converted functionalities to Saga orchestrations, which will be useful during the evaluation phase.

Table 2.3: Benefits and drawbacks of the choreography approach.

Benefits	Drawbacks
Suitable for simple workflows with few participants	Workflows can become confusing, are more challenging to track and debug
Does not require additional orchestrator service implementation and maintenance	Risk of cyclic dependencies between participants, since they consume each other's events
It does not introduce a single point of failure	All services must be running to simulate a transaction, making integration testing more difficult.
More separation of concerns, which simplifies the business logic	

2.2 Related Work

By looking at the research community, we can find significant contributions to the problem of migrating a monolithic codebase to a microservices architecture [3, 9–15]. Although they propose different techniques, they concur on the definition of three steps of the migration process:

1. Collection of data about the original codebase.
2. Generation of a decomposition.
3. Visualization of metrics and migration of the functionalities.

Data collection techniques like static and dynamic analysis are applied to extract data about the monolith behavior, either based on data models or the monolith's source code. This information is fed to the second step, which identifies candidate decompositions of the monolith into a set of microservices, each containing a set of the domain entities of the system, based on a group of metrics that maximize the expected quality of the decomposition. Finally, some of the approaches also support visualizing the algorithm's results, which the architect can use to interact with a representation of the candidate decomposition and even change the decomposition while continuously receiving feedback on the impact the changes have on the quality metrics. The authors in [12] developed a framework called Functionality-oriented Service Candidate Identification (FoSCI) to identify service decomposition candidates, including entity and interface identification. This framework identifies service candidates through the extraction and analysis of execution traces. The workflow of the solution proposed is divided into three main steps:

1. Execution Trace Extraction - extraction of representative execution traces based on a file containing execution logs.
2. Entity Identification - based on the execution traces, the tool identifies functional atoms, which represent units where all the entities are responsable for the same functional logic, and then uses a multi-objective optimization technique to group this functional atoms in candidate microservices.

3. Interface Class Identification - for each service candidate, the framework identifies its interface classes.

Moreover, the contributors develop an evaluation to assess the service candidates on top of eight metrics that quantify three quality criteria for service candidates. These metrics are derived from the revision history stored in the application version-control system and provides an evolution map of the codebase. These quality criteria for the service candidates 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.

This research is comparable to the work developed in the context of the Mono2Micro system since it provides a similar pipeline for the identification of microservices. However, their metrics focus on different microservices qualities. Additionally, while our research focuses on the functionalities migration complexity by applying design patterns, the author's research focuses on the quality of the decompositon according to the three quality criteria described above.

The values returned by the quality metrics are strongly dependent on the monolith functionalities structure, but this aspect is absent in the literature, except in [5], where it is analyzed how the refactoring of a functionality can significantly impact the value of complexity metrics for the monolith decomposition.

Most of the approaches, but [3, 11], do not consider the interactive capability of experimenting with different decompositions, containing different numbers of clusters.

In [11] the authors proposed a visualization tool that allows developers to migrate an existing monolithic application as microservices interactively. This task is achieved by initially constructing a callgraph using a codebase execution profiler, then generating two different microservice designs by applying two clustering techniques, one using semantic-based clustering, which takes into account semantic similarities between the classes, and another using call-context-based clustering, which creates clusters based on the communication between the multiple modules of the codebase. After this decomposition strategy, the architect is then presented with a user interface containing clusters of classes, where he can freely change the candidate decomposition of the system by applying a class operation from the list available:

- Create a new microservice with a selected class.
- Move a class into another microservice.

- Clone the selected class into all the microservices that communicate with it.

The results when applying the call-context-based clustering showed a considerable reduction of the number of calls between services, which is a critical metric when developing a microservices architecture since the communication between clusters must be very coarse-grained in order to limit the number of intermediate states and inter-cluster dependencies.

This approach, despite being similar to what is applied in Mono2Micro in [3] when it comes to creating a decomposition and allowing the developer to edit it, it is still, however, a manual task and does not apply a microservice-specific design when it comes to managing data consistency and database transactions in a distributed manner.

The current research on code-smell and design pattern detection follows two main trends: heuristic algorithms and machine learning classification models. In most heuristic classification approaches, a set of code metrics is computed and combined to create detection rules, which by employing thresholds, decides if the code follows a certain pattern [16–19]. However, some drawbacks have been identified in the heuristic approach due to the low agreement between different detectors and difficulties in finding suitable thresholds to be used for detection [20].

In [16] the researchers develop a well-defined methodology that summarizes and defines all the steps necessary for the specification and detection of code and design smells. This methodology, called *DECOR* (Detection & Correction), is composed of five steps:

1. *Description Analysis*: by doing an extensive analysis of the literature descriptions for each code-smell, key concepts are identified, which form a glossary of concepts to describe each one of the smells when it comes to specific code abstractions, e.g., Low Cohesion, No Inheritance, Private Field, Uses Global Variable.

2. *Specification*: The concepts are aggregate in order to specify smells concisely.

3. *Processing*: The specifications, created in step 2, are translated into detection algorithms that can be directly applied to a system.

4. *Detection*: The detection algorithm is applied to the system and returns the list of code abstractions (e.g., classes, methods) suspected of implementing a smell.

5. *Validation*: The outputted abstractions are manually analyzed to assert that they contain, in fact, the code-smell.

By applying DECOR, the scholars develop their own detection technique called DETEX (Detection Expert), allowing software engineers to specify smells at a high level of abstraction using a unified vocabulary and domain-specific language and automatically generate detection algorithms. This implementation is supported by a Domain-specific Language (DSL) called SmellDL (Smell Definition Language)

for specifying smells using high-level abstractions, which is used to execute steps 1 and 2. Then, in order to generate the detection algorithms, the authors resort to a framework developed in previous research, called SmellFW (Smell FrameWork), built on top of the Pattern and Abstract-level Description Language (PADL) meta-model and Primitives, Operators, Metrics (POM) framework. This framework exposes interfaces to build systems models, manipulate them, and generate other models using the Visitor design pattern. This framework is applied to the code-smell models built in step 2, generating source code that implements detection algorithms through JAVA templates and services within the framework. By replacing well-defined tags in these templates with concrete code, the developer can apply the detection algorithm to a specific code abstraction.

Additionally, it is performed an empirical validation of the algorithms generated by DETEX, in terms of precision and recall, by applying them to 4 code-smells: Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife, on 11 open-source codebases, and then manually validating the results. This validation resulted in a recall of 100% for all the smells and precision between 41.1% and 90%. This classification provided between 5.6% and 15% of the total number of the classes of each codebase, as having smells, which is a fair number of classes to be manually analyzed compared to having to analyze the entire systems. With this, the authors were able to reduce the effort required to identify refactorizations to be made to a system, which goes hand-to-hand with what we want to achieve in this research. However, their approach only identifies antipatterns in the code but does not identify the final design of the system and the specific changes that need to be made, which is something that we want to achieve.

Some other heuristic tools base their analysis on the data contained in version control systems since it provides a good overview of how the elements of a codebase change over time. In [17] the researchers propose an approach, named Historical Information for Smell deTection (HIST), to detect smells based on change history information mined from versioning systems, specifically, by analyzing co-changes between source code artifacts. The technique aims at detecting five specific smells: Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy. The authors based their choice of smells on the need to have a benchmark, including smells that can be identified using change history information and smells that do not necessarily require this type of data. The first three smells, Divergent Change, Shotgun Surgery, and Parallel Inheritance, are defined as historical smells, which means that they can be detected using revision history. However, the last two code smells (Blob and Feature Envy) can also be detected solely relying on structural information about the codebase's classes, which is explored in several approaches using static analysis tools, like [16]. The heuristic algorithms applied for detecting each one of the smells are briefly described below:

- Divergent Change Detection: The detector mines association rules for detecting subsets of methods in one class that often change together. This extraction is done by identifying compositions of

methods that have been committed together in the version control repository.

- Shotgun Surgery Detection: A class affected by this smell contains at least one method changing along with several other methods contained in other classes. Accounting for this, the researchers use association rules for detecting methods that often change together so that the smell is identified in a class if it contains at least one method that changes with methods present in more than δ classes.

- Parallel Inheritance Detection: This smell occurs when adding a subclass to one class requires also adding a subclass to another one. In this approach, this is also detected by identifying, in the version control data, pairs of classes that had subclasses added to them in the same commit.

- Blob Detection: The detection strategy for this smell is quite simple since it only requires identifying classes that were modified in more than α% of the commits involving another class. This detection algorithm assumes that a Blob occurs when a specific class needs to be modified whenever a new change is done to the software system.

- Feature Envy Detection: The contributors state that a method affected by this code smell changes more often with the envied class than with the class it lives in. With this, they develop an approach that identifies methods that are involved in commits with methods of other classes, by a threshold of β% more, than in commits with methods of their own class.

Finally, the researchers proceed to evaluate HIST with two empirical studies. The first, conducted on 8 Java projects, aimed at evaluating HIST's detection accuracy in terms of precision and recall against a manually-produced identification. The second compared HIST, wherever possible, with results produced by approaches that detect smells by analyzing a single project snapshot, such as JDeodorant, and the author's re-implementations of DECOR's detection rules, known as DETEX, in [16].

The results of the study indicate that HIST's precision is between 61 and 80 percent, and its recall is between 61 and 100 percent. HIST tends to provide better detection accuracy compared to alternative approaches, especially in terms of recall, since it can identify smells that other approaches omit because they do not consider historical information.

One of the visible drawbacks of this approach is that it relies on correct usage, within the engineering team, of the version control commit patterns. If more than one system change is made to the codebase, the algorithm's precision will inherently be affected. Also, once again, there is an intrinsic reliance on configuration thresholds, such as in detecting Shotgun Surgeon, Blob, and Feature Envy smells. Our approach will differ significantly from this research since it rellies solely on structural data about the codebase, and instead of having well-defined detection strategies for specific codebases, we will focus on minimizing the migration complexity of the system by reducing the number of inter-service communications in the context of a decomposition.

In [18] the authors present a technique, called TACO (Textual Analysis for Code Smell Detection), that uses textual analysis of the source code to detect a set of code smells with different natures and different levels of granularity: Long Method, Feature Envy, Blob, Promiscuous Package, and Misplaced Class. The proposed process to calculate the probability of a code component being affected by a smell can be summarized in three steps:

1. Textual Content Extractor: the first step is to extract textual content characterizing each code component of the module being analyzed, including source code identifiers and comments. Despite this being one of the important steps of the technique, the authors do not go much in-depth on the approach used to extract this data.

2. Normalization Process: the dataset containing the identifiers and comments extracted in the step above is cleaned by using a *Information Retrieval* normalization process. This normalization applies the following changes: separating identifiers by using a camel case splitting method; reducing all the extracted words to lower case; removing special characters; stemming words to their original roots via Porter's stemmer. Finally, a *term frequency* algorithm is used to reduce the relevance of too generic words contained in the text.

3. Smell Detector: in this step, the normalized dataset of each component is analyzed by applying a set of heuristics. The detector relies on Latent Semantic Indexing (LSI), an extension of the Vector Space Model (VSM), modeling code components as vectors of terms. These vectors are then projected into a reduced k space of concepts to limit textual noise. To obtain the textual similarity between components, they calculate the cosine of the angle of the corresponding vectors. Finally, these similarity values are combined using different heuristics according to the smell being detected to obtain the probability that a code component is smelly.

Below we proceed to describe the heuristics used in the Smell Detector step to detect each code smell:

- Long Method: the authors define that a method is affected by this smell when it is composed of sets of statements semantically distant from each other. To detect this, they use an implementation of a text analysis algorithm, called SEGMENT, which automatically segments a method into a set of consecutive statements implementing a high-level action. For each pair of statements in the list, they apply LSI and cosine similarity to calculate the similarity value.

- Feature Envy: for this smell, the authors define that *a method more interested in another class is characterized by a higher similarity with the concepts implemented in another class, when considering the concepts of the class it is in*. To calculate the probability of this smell, the authors derive

the class having the highest textual similarity with the method. If the resulting class is not the class where the method is placed, then Feature Envy occurs.

- Blob: the author's conjecture for this smell is: *Blob classes are characterized by a semantic scattering of contents*. To detect this, they calculate the mean cosine similarity between each pair of methods contained in the class, named $ClassCohesion$, with values between 0 and 1. This is then used to calculate the probability of the class being a Blob by applying the formula: $P_B(C) = 1 - ClassCohesion(C)$.
- Promiscuous Package: *packages affected by this smell are characterized by a subset of classes semantically distant from the other classes of the package.* This conjecture is applied by applying a very similar method as the Blob smell, but at a package level, where the mean cosine similarity is calculated for each pair of classes in the package, named $PackageCohesion$, which is then used to calculate the probability of the Promiscuous Package smell occurring, by applying the formula $P_P P(P) = 1 - PackageCohesion(P)$.
- Misplaced Class: the conjecture for this case is: *a class affected by this smell is semantically more related to a different package with respect to the package it is actually in*. To detect this, the authors go by retrieving the package with the highest textual similarity with the class being analyzed. If the resulting package is different from the actual package of the class, then the class should be moved.

They run TACO on ten open source projects, comparing its performance with existing smell detectors purely based on structural information extracted from code components. The analysis of the results indicates that TACO's precision ranges between 67% and 77%, while its recall ranges between 72% and 84%. This approach presents itself as a good candidate for solving the dependence on configurable thresholds and strict detection rules that tend to occur in usual heuristic strategies. It relies on well-defined conjectures for each smell that depend purely on the textual characteristics of the source code. However, the final purpose of the smell detection is different from the one we present in our research since we focus only on a smell present in regular Object-Oriented Programming (OOP) software, where modules have a very fine-grained communication behavior with many calls to separate classes since this is the smell that highly increases the migration complexity to a microservices architecture of any functionality. Additionally, we also want to effectively fix the smell in the context of a decomposition, instead of just detecting it, as is done in [18].

When it comes to machine-learning classification approaches to identifying code-smells and design patterns, a considerable amount of work has been done by a cluster of researchers led by Francesca Fontana. The techniques presented by the authors in [21–24] tend to be more flexible and independent compared to heuristic classification, since learning by example allows for better handling of distinct

scenarios. However, they require extensive manual classification work to train the machine learning algorithm.

In [21], the authors developed a benchmark for design pattern detection tools, composed of a work-flow with two main steps:

1. Exact Matching: a module called *Joiner* extracts all the pattern instances contained in a software codebase. This module executes logic to find all the instances that match an exact rule, which is very general and considers only the fundamental behavior of the pattern. As a result, this *Joiner* tends to produce a large number of results, but with low precision.

2. Classification: a module called *Classifier* classifies as correct or incorrect the instances detected by the *Joiner*, resulting in a better filter for the results.

By exploring this workflow, the contributors were able to create a base model for identifying design patterns in the context of a machine-learning approach and formulate the problem specifically as a supervised classification task. Moreover, they also generated a large dataset of manually verified design pattern instances, which is required to evaluate the base model.

This work distinguishes itself by resorting to a two-step data collection step, with a classification step after an exact matching step. Additionally, this process does not depend on the supervised machine-learning classification algorithm used, which promotes future experimentation.

In [22] the researches pick up on previous work done in [21] and implemented the Design Pattern Detection (DPD) approach to the the Metrics and Architecture Reconstruction Plug-in for Eclipse (MARPLE) project. Here, they enhance the previous experimentation by:

- Developing test cases with more design patterns.

- Implementing and testing the model with more machine learning techniques.

- Testing the algorithms on a larger dataset.

- Applying an automatic and systematic method for the optimization of the algorithm's parameters.

Additionally, the detection process presented in [21] is improved by introducing a clustering algorithm for particular cases where the pattern structure is flat. The rationale for this was to provide the classifier algorithms with a more direct representation of the pattern instances.

The approach is tested to detect five specific design patterns: Singleton, Adapter, Composite, Decorator, and Factory Method, on ten open-source software systems. In this experimentation phase, the authors divided the patterns into two groups: one for the patterns Singleton and Adapter, where they applied only classification models, and a second group containing Composite, Decorator, and Factory Method, where they applied a cascade of clustering and classification models. The approach obtained

good performance values, especially for detecting the Singleton, Adapter, and Factory Method patterns. The authors attributed lower performances in other patterns to the lack of examples in the dataset, which shows that a larger, more diversified dataset should be used. As for the best classification models, Support Vector Machines (SVM), Decision Trees, and Random Forests resulted in the best results for this specific dataset.

In [23], Fontana et al, extend the work on exploiting supervised machine learning techniques by proposing a methodology to support a learn-by-example process to build code smell detection rules. The resulting models can provide a confidence value of how the results fit the detected pattern and also, in some cases, provide human-readable rules, which allowed the researchers to analyze which combinations of input metrics have more influence on the detection of a given pattern.

Once again, this research required manually evaluating a set of example class instances for each code smell, by classifying them as *affected* or *not affected* by the antipattern. This manual evaluation required scraping dozens of projects in search of those instances, which was made more efficient by applying a stratified random sampling approach on 74 codebases, guided by the results of pre-existing code smell detection tools in the market, which the authors called *Advisors*. The usage of such a strategy ensures that the selection of instances to evaluate is homogeneous and prioritizes the labeling of instances with a higher chance of being affected by a code smell. This dataset of instances is then used to train each one of 32 supervised machine learning classification algorithms on detecting four code smells Data Class, God Class, Feature Envy, and Long Method.

To summarize, the proposed approach requires five sequential steps, described below:

1. *Data Collection*: The authors consider a collection of 111 Java systems, from where they extract 74, which are compiled correctly in order to compute metric values. The selected projects are quite heterogeneous in size and application domain. They state that the size of the dataset to be the largest available for code smell detection using machine learning algorithms and provide a high enough number of systems which ultimately results in a generalized model.

2. *Metrics Extraction*: A large set of well-known OOP metrics was computed using a variance of the Eclipse JDT library on all the 74 systems. Some of these metrics describe characteristics of the codebase at class, method, package, and project level, which are needed to feed the *Advisors* chosen. Others are simple standard metrics such as complexity, cohesion, size, and coupling, which are pretty similar to the metrics generated by our Mono2Micro system.

3. *Choosing the Advisor tools*: the researchers analyzed the literature related to code smell detection tools and handpicked the ones that respected a set of criteria: can be implemented as an external tool; must perform batch computation and export data in a parsable and documented format; have a different approach to advisors for the same smell, so that they avoid possible correlations among

similar rules. The authors also considered tools only defined by research papers, if however, the rule is clearly described and is replicable.

4. *Labeling*: During this phase, the Advisors were executed for each one of the codebases in order to extract instances that are classified as having a code smell. However, since these automated classifiers are prone to error, the authors implemented an additional manual labeling phase. In this phase, a group of three MSc students individually evaluated each selected instance by inspecting the code and classified them according to 4 severity levels: *0 - non smell*, *1 - non-severe smell*, *2 - smell* and *3 - severe smell*. In the context of the machine learning training phase, these labels were grouped as: 0 - INCORRECT, 1, 2, 3 - CORRECT, which ultimately showed to provide more information to the model when comparing to traditional binary classification, producing more accurate results.

5. *Experimentation*: A set of machine-learning algorithms was trained on the datasets and tested using 10-fold cross-validation. This dataset was separated per code smell, with two datasets per smell. Each row of the dataset represented a class or method instance, with one attribute for each metric and the last column containing a boolean representing the label that states if the instance is a code smell or not. Some of the machine learning classifiers used are J48 Decision Trees, JRip, Random Forest, Naive Bayes, SVM, and SMO.

2.3 Conclusion

The research on the migration of monolith systems to a microservices architecture that uses automatic and semi-automatic methods already use heuristic, e.g. [9, 12], and machine learning, e.g. [25, 26], techniques to the identification of candidate decompositions, but there is no work on the automatic identification of code-smells and recommend refactorings to ease monolith functionality migration in the context of a candidate decomposition.

3

Solution

Contents

3.1 Strategy 24

3.2 Functionality Migration 26

3.3 Saga Refactorization Algorithm 29

In this chapter is presented the process that resulted in the decision of the best approach to follow to solve the problem presented in Chapter 1. This decision process involved analysing the Saga refactorization done by an expert in a specific codebase and identifying patterns in his decision making. Later, is presented the formalization of concepts around the task of migration a functionality from a monolith architecture to a microservices architecture applying the Saga pattern, which also includes the complexity metrics that will be minimized by the solution. Finally, is presented the algorithm responsible for the proposition of Saga orchestrations, in the context of a monolith decomposition and the functionalities callgraph.

3.1 Strategy

In order to make a justified decision about the approach that better fits this use case, we start by analyzing how the human developer decides which changes must be done to the functionalities callgraphs to refactor them as Saga orchestrations, and if there is a way to mimic this decision-making. Based on the analysis of the human refactorizations and the data that we have available, then proceeded to cement the approach that will be followed in the rest of the thesis.

In previous research [5], the authors defined a set of operations that can be made to a functionality callgraph, in the context of a candidate decomposition, in order to refactor it as a Saga orchestration without breaking the data-state layer of the codebase:

- *Sequence Change*: the flow of execution of the functionality is changed, which happens by swapping the order of the original sequence of local transactions. In Fig. 3.1 we see the application of this operation to a functionality that involves two clusters, A and B, where B performs two local transactions, and A does only one. Suppose the local transaction in B, with *id* 2, does not depend on data that is read by the local transaction in A, with *id* 1. In that case, they can be swapped without breaking the data dependencies of the functionality.

- *Local Transaction Merge*: used when two local transactions in the same cluster become adjacent in the sequential callgraph. Since it does not make sense to have a remote invocation between the same cluster, we can merge both the local transactions as is seen in Fig. 3.2. When this is done, it is necessary to integrate both execution sequences which results in a reduced number of intermediate states.

- *Define Coarse-Grained Interactions*: this operation happens when both Sequence Change and Local Transaction Merge operations are applied sequentially, in that order. More visually, this is represented by joining both Fig. 3.1 and Fig. 3.2

Figure 3.1: Sequence Change operation applied to a decomposition callgraph.

Figure 3.2: Local Transaction Merge operation applied to a decomposition callgraph.

After defining these operations, the authors proceeded to manually refactor a set of functionalities in the LdoD codebase[1] into Saga orchestrations, and evaluate the results when it comes to the reduction of the Functionality Migration Complexity (FMC) and System Added Complexity (SAC) metrics.

Picking up on this work, we started by reverse-engineering the author's refactorings to understand his decision-making and conclude if it follows a standard pattern. With this, we defined a set of questions to be answered:

- How are data dependencies preserved when the Sequence Change operation is applied?
- How is the orchestrator cluster chosen?

We extended this analysis to 8 functionalities of the codebase, for which we extracted the initial sequential callgraph and the final callgraph resulting from the refactorings of the authors, and analyzed the source code to make conclusions on the design decisions made. With this, we reached the following conclusions:

- When it comes to the data dependencies, where two local transactions must not be swapped if the second depends on data read by the first, we saw that not always did the developer respect them. This conclusion does not suggest that the developer broke the data state changes in the refactored design, but instead, it shows sometimes when having a write operation in cluster B after a read operation in cluster A, the write in B does not actually use the data read in A.
- The rules defined in [5] were not enough to reach the refactorings made by the authors, since some data dependencies where not taken into consideration while refactoring the functionalities. This outcome indicates that some design decisions are particular to the codebase itself and how the developer approaches the refactorization task, and indicates that heuristic rules tend to be too strict and do not account for the deviation in the codebases implementations. This deviation in the data is something that a deep learning model trained on a large dataset of refactorized codebases may handle better.

[1] `https://github.com/socialsoftware/edition`

- Regarding the last question to be answered, we could not find a pattern regarding the decision, by the authors, of the clusters that should orchestrate each one of the functionalities. These orchestrators are not always the ones that have more access to domain entities.

By answering the questions above, we concluded that heuristic tools based on defined rules are too strict and do not account for the large number of levels of freedom that codebases have. The rules may result in many test cases, but at some point, they become a bottleneck when trying to reach the least complex design possible. These observations lead us to believe that a machine-learning classifier approach may be the end-game at developing a refactorization system that can analyze codebases implementing all kinds of design patterns, since theoretically, it is possible to train the classifier with a large dataset of functionalities, their monolithic metrics, and the resulting orchestrations to achieve a much better behavior when the model views data that is new and was not included in the dataset.

However, this characteristic of the machine-learning classifier, which requires an extensive dataset with human-made refactorizations, is challenging to solve. It is hardly possible to have a dataset of at least 100 functionality refactors analyzed purely by human developers without using existing automated tools simply because it takes an immense amount of time to complete. The only dataset we have available has the refactorizations made by the author in [5], and they are nowhere near enough for training a deep learning model. This shortage of data became a stepping stone for any attempt at following an approach based on machine-learning classifiers.

Considering this, and although their limitations, we decided to approach the problem from a purely heuristic standpoint.

3.2 Functionality Migration

Definition: Monolith. A monolith is defined as a triple (F, E, T), where F denotes its set of functionalities, E the set of domain entities, T a set of traces of the monolith functionalities accesses. The traces are defined as a triple (A, S, D), where $A = E \times M$ is a set of read and write accesses to domain entities ($M = \{r, w\}$), $S = A \times A$ a execution sequence relation between elements of A, which indicates that the first element of the pair was invoked, in the context of a monolith functionality, immediately before the second element, and $D = A \times A$ the data dependencies between accesses in the context of a sequence, where the first element is a read access, the producer, and the second element a write access, the consumer. Given a sequence, $s \in S$, its transitive closure, s_t, is a total order, in particular, any element is comparable and there is no circularities, $\forall_{(a_i,a_j)\in s_t}(a_j, a_i) \notin s_t$ For each functionality $f \in F$, $f.s \in S$ denotes its set of sequence accesses. Additionally, the data dependencies occur in the context of a functionality sequence of access and conform to the sequence order, $\forall_{(a_i,a_j)\in D}\exists_{s\in f.s}(a_i, a_j) \in s_t$.

Definition: Monolith Decomposition. The decomposition of a monolith into a group of candidate

clusters is defined by the set of clusters C, where each cluster represents a microservice and contains a set of domain entities, which are tightly coupled and represent a specific abstraction of the system. Given candidate decomposition, the monolith functionalities are decomposed into a set of local transactions, where each local transaction corresponds to the Atomicity, Consistency, Isolation, Durability (ACID) execution of part of the functionality domain entity accesses.

A decomposition relaxes the consistency of the monolith's functionalities, and the level of impact in the system depends on the introduction of intermediate states on a functionality that was previously atomic.

Definition: Functionality Migration. The migration of a monolith functionality is based on the information collected of the functionalities accesses to the monolith domain entities, the sequences of accesses, and their data dependencies.

Functionality migration occurs in the context of a candidate decomposition, in which each candidate microservice is represented by a cluster of domain entities. Therefore, given the set of sequence of accesses $f.s$ of a functionality f and a decomposition into a set of clusters of the domain entities, $C \subseteq 2^E$, where the clusters are non-empty and a domain entity is in exactly one cluster, the partition of a sequence s of a functionality f, $s \in f.s$, $P(s,C) = (LT, RI)$ is defined by a set of local transactions LT and a set of remote invocations RI, where each local transaction:

- is a tuple (A, S)
- is a subsequence of the functionality sequence of accesses, $\forall_{lt \in LT} : lt.s \subseteq s$;
- contains only accesses to the domain entities of a single cluster, $\forall_{lt \in LT} \exists_{c \in C} : lt.a.e \subseteq c$;
- contains all consecutive accesses in the same cluster, $\forall_{a_i \in lt.a, a_j \in s.a} : ((a_i.e.c = a_j.e.c \wedge (a_i, a_j) \in s) \Rightarrow (a_i, a_j) \in lt.s) \vee ((a_i.e.c = a_j.e.c \wedge (a_j, a_i) \in s) \Rightarrow (a_j, a_i) \in lt.s)$;

Definition: Functionality Partition. A partition of a functionality f, given a decomposition C, is the union of the partition of each one of its sequences, $P(f,C) = \cup_{s \in f.s} P(s,C)$, where local transactions and remote invocations that are in the common prefixes of sequences are not repeated. Additionally, sequence and data dependence relations between local transactions are inferred from the functionality original sequence and data dependence relations and are denoted by $<_S$ and $<_D$, respectively.

The partition of a functionality f, given a decomposition C, and obtained using the above rules, is called the initial partition of f and it is denoted by $P_i(s,C)$ and $P_i(f,C)$, for, respectively, a sequence s and all sequences of f.

Definition: Functionality Refactor. The refactor of a functionality, given a decomposition, is done from its initial partition to a *Saga* implementing an orchestration, where the orchestrator is a pivot of the interactions between local transactions. Therefore, a partition of a functionality sequence of accesses

is an orchestration if it is involved in all remote invocations, c is an orchestrator if $\forall_{(a_i,a_j)\in RI} : a_i.e.c = c \vee a_j.e.c = c$. Note that, by definition, a remote invocation occurs between different clusters.

The refactor of a functionality sequence of accesses is the change of its initial partition into a partition that is an orchestration. This refactoring is done to minimize its migration complexity in the context of a distributed transaction, because in the new implementation the functionlity business logic has to be migrated. This complexity is measured by the number of local transactions and the impact that a local transaction has on other functionalities migrations. This is due to the lack of isolation that each local transaction brings to the functionality migration in a decomposition, which requires the introduction of compensating transactions and the need to handle intermediate states of the domain entities accessed by different functionalities [4, 5].

Definition: Functionality Migration Complexity (FMC). The complexity of migrating a functionality f, given its partition (LT, RI) in a decomposition, is the sum of the complexity of each one of its local transaction:

$$complexity(f,(LT,RI)) = \sum_{lt\in LT} complexity(lt)$$

Definition: Local Transaction Complexity. The complexity of a local transaction depends on the domain entities it writes because it is necessary to implement compensating transactions for when the functionality execution has to rollback. It also depends on the domain entities it reads because it is necessary to consider the intermediate states that other functionalities introduce when they write them in their local transactions.

$$complexity(lt) =$$

$$\#writes(\{lt.s\}) + \sum_{e\in reads(\{lt.s\})} \#\{f_i \in F \setminus \{f\} : \exists_{lt_j \in P(f_i,C)} e \in writes(lt_j.s)\}$$

where

$$writes(s) = \{e : \exists_{s_i\in s}(e,w) \in prune(s_i)\}$$

$$reads(s) = \{e : \exists_{s_i\in s}(e,r) \in prune(s_i)\}$$

and the prune function, when applied to a sequence of accesses, removes all read accesses to a domain entity after the first read access to that entity, and removes all accesses to an entity after the first write access to that entity. This function identifies which accesses are relevant for other functionalities, because the local transaction executes with strict consistency properties and so, after a write local reads do not need to concern about external writes done by other functionalities, and only the first read has to consider intermediate states generated by other functionalities.

Definition: System Added Complexity (SAC). The migration of a functionality impacts other functionalities migrations complexity. For instance, if a write is done on an 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 handle the possible intermediate states. Hence, the cost of migrating f_j depends on the number of writes done by f_i in entities that f_j reads.

$$addedComplexity(f_i, (LT, RI)) =$$
$$\sum_{lt \in P(f_i, C)} \sum_{f_j \in F \setminus \{f_i\}} \#(\{e : \exists_{lt_j \in P(f_j, C)} e \in reads(lt_j.s)\} \cap writes(lt.s))$$

3.3 Saga Refactorization Algorithm

The approach we present in this thesis applies a brute-force heuristic algorithm to calculate, for each functionality, what is the Saga orchestration that result in the lowest migration cost to a *Saga* microservices architecture. In order to properly describe the algorithm, we first present and high-level flowchart analysis of the main execution logic, and then present specific code modules responsible for refactorizing the functionality callgraph.

3.3.1 Flowchart Analysis

In Fig. 3.3 we present a flowchart with the high-level execution flow of the algorithm. It performs one refactorization for each one of the clusters in a functionality, picking in each iteration a different cluster as orchestrator.

In order to create a Saga design, the algorithm copies the initial callgraph structure and inserts one invocation of the orchestrator cluster between each invocation of two other clusters in the functionality. These invocations are added without containing any accesses to domain entities, to create the intermediate state where each service reports back to the orchestrator to inform if its transaction was successful or not. After this, the tool proceeds to execute a recursive method that iterates through each invocation in the functionality callgraph and checks if it can be merged with the previous invocation of the same cluster.

As we will explore later, two invocations can be merged if there are no read accesses in the latest δ invocations of other clusters before the second one. This scope allows us to have some configuration of the rigidity of the data dependence classification, and theoretically, lower values could result in more merge-operations.

If the invocations are mergeable, they collapse into the first one, and their domain entity accesses are pruned. The recursive method stops once we reach the end of a cycle through the callgraph where the algorithm did not find any invocation that could be merge, which indicates that the functionality is at the most simplified state that does not break data dependencies.

After all the invocation merges are complete, the Saga refactorization is saved in a data structure.

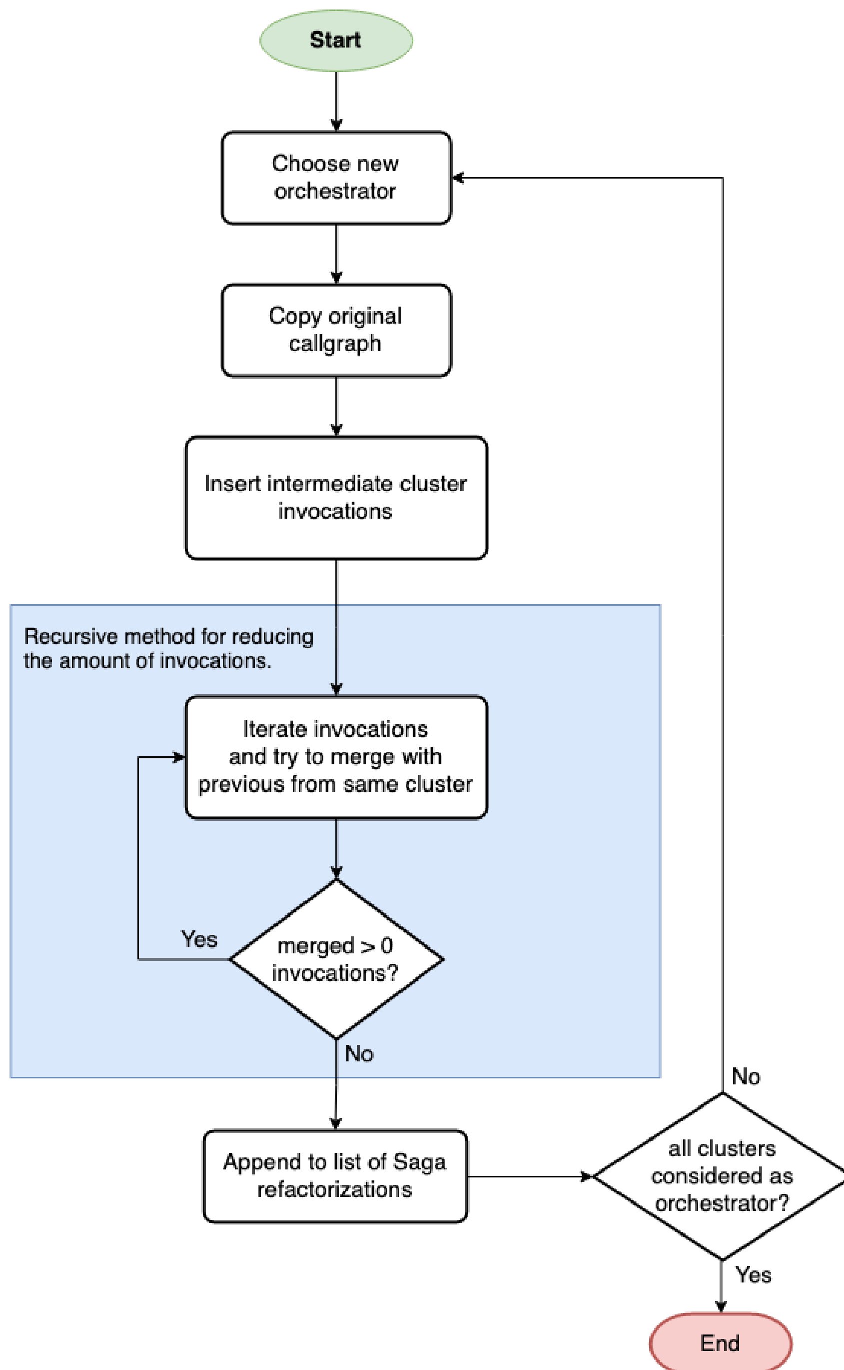

Figure 3.3: Flowchart for the algorithm that computes the list of possible Saga orchestrations for a functionality.

The algorithm proceeds to execute the logic again, now with another cluster as orchestrator. When all of the clusters in the functionality have been considered, the work is done, and the execution reaches its end.

3.3.2 Pseudocode Analysis

The main algorithm $estimateSagas(F, C)$, Listing 3.1, uses the functionalities initial partitions P_i, given a candidate decomposition, and calculates, for each functionality, what is the migration cost when considering each of the clusters that are involved in the functionality implementation as orchestrators. The result is an ordered array of migration complexities where the cluster that has the lower value is the recommended orchestrator for the functionality refactoring. The algorithm has three steps: (1) sets a cluster orchestrator of the functionality, $setOrquestrator(P_i(s, C), c)$; (2) merges the invocations between the same pairs of clusters, to obtain coarse-grained invocations, $mergeInvocations(p, c)$; (3) calculates the complexity, $complexity(f, p_c)$.

Listing 3.1: *SAGA* estimator

```
1  estimateSagas(F,C) {
2      complexities := array[F.size, C.size]
3      for f := range F {
4          for c := range C {
5              pc := {}
6              for s := range f.s {
7                  p := setOrquestrator(Pi(s,C), c)
8                  p := mergeInvocations(p, c)
9                  pc := pc ∪ p
10             }
11             complexities[f,c] := complexity(f, pc)
12         }
13     }
14     complexities[f].sorted
15 }
```

To set a cluster as the orchestrator of an initial partition of a functionality it is necessary to add empty local transactions to the orchestrator cluster and remote invocations to the other clusters, Listing 3.2. The sequence of execution of the local transactions is preserved because the only change is the introduction of the empty local transactions, which works as pivot between the invocations. Therefore, the data dependencies between the local transactions are not changed. When the the added remote invocation contains an element where lt.previous or lt.next does not exist, the remote invocation is discarded. This is done to avoid the introduction of conditionals in the algorithm.

Listing 3.2: Set Orchestrator for Partition

```
1  setOrchestrator((LT,RI),c) {
```

```
2      resultLT := LT
3      resultRI := {}
4      skip := false
5      for lt = range LT.sortedBy(<S) {
6          if (!skip) {
7              if (lt.c == c) {
8                  resultRI := resultRI ∪ {(lt.previous, lt), (lt, lt.next)}
9                  skip := true
10             } else {
11                 emptyLT := new emptyLT(c)
12                 resultLT := resultLT ∪ {emptyLT}
13                 resultRI := resultRI ∪ {(lt.previous, emptyLT), (emptyLT, lt)}
14                 skip := false
15             }
16         } else {
17             skip := false;
18         }
19     }
20     return (resultLT, resultRI)
21 }
```

Fig. 3.4 exemplifies a transformation made according to *setOrchestrator* in Listing 3.2, where we set the empty pivot orchestrator invocations between each one of the other cluster invocations.

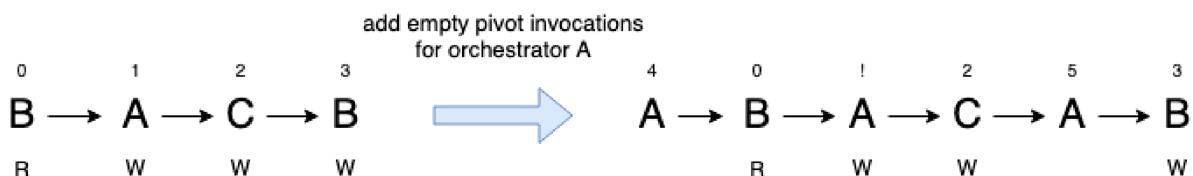

Figure 3.4: Operation of adding pivot orchestrator invocations

The merge of invocations, Listing 3.3, merges the invocations that occur between the orchestrator and another cluster, if there is no data dependence with a local transaction that occurs in between. It repeats recursively while two remote invocations can be merged into a coarse-grained remote invocation. The `canMerge` condition, which applies to pairs of invocations between the same clusters, is defined by the data dependence relation $<_D$, such that, it does not exist a local transaction between the two remote invocations that the local transactions in the second remote invocation have a data dependence on. As a result, the merged local transaction sequences are concatenated and the data dependencies between them are preserved.

Listing 3.3: Refactor through merge of fine-grained invocations into coarse-grained.

```
1  mergeInvocation((LT,RI),c) {
2      resultLT := LT
3      resultRI := RI
4      while (ri1 in range RI, ri2 in range RI.after(ri1), canMerge(ri1,ri2)) {
5          ri1.caller.prune(ri2.caller)
6          ri1.callee.prune(ri2.callee)
7          resultLT := resultLT \ {r21.caller, ri2.callee}
8          resultRI := resultRI ∪
9              {(ri2.previous.caller, ri2.next.callee)} \
10             {ri2.previous, ri2, ri2.next}
11     }
12     return (resultLT, resultRI)
13 }
```

The $canMerge(ri_1, ri_2)$ function can be parameterized to allow variations on the scope (δ) of data dependencies to be considered. When $\delta = 1$ a data dependence is only considered if it occurred in the local transaction immediately before the local transaction being analyzed, however, if $\delta = \infty$ are considered all the local transactions in between the ones to be merged. Allowing different scopes permits different evaluations because a previous read does not necessarily imply that it is used in a subsequent write.

Figure 3.5 illustrates a transformation performed by *mergeInvocation* in Listing 3.3, where all invocation that do not have data dependencies, according to the value of δ, are merged.

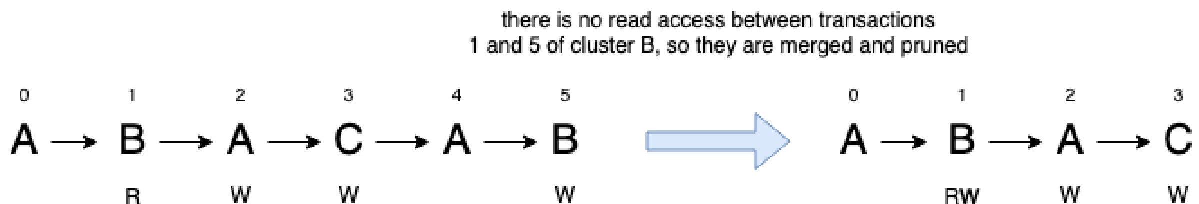

Figure 3.5: Operation of merging and pruning invocations without data dependence

In Figure 3.6 are showed two distinct cases of merging and pruning invocations, according to different values of the data dependence threshold. As it can be seen, when the threshold is 1, the algorithm allows for the merging of the invocations in A. However if the threshold is bigger, the algorithm will consider that the last invocation of A depends on the read operation made in B, and due to this, it doesn't merge both A's invocations.

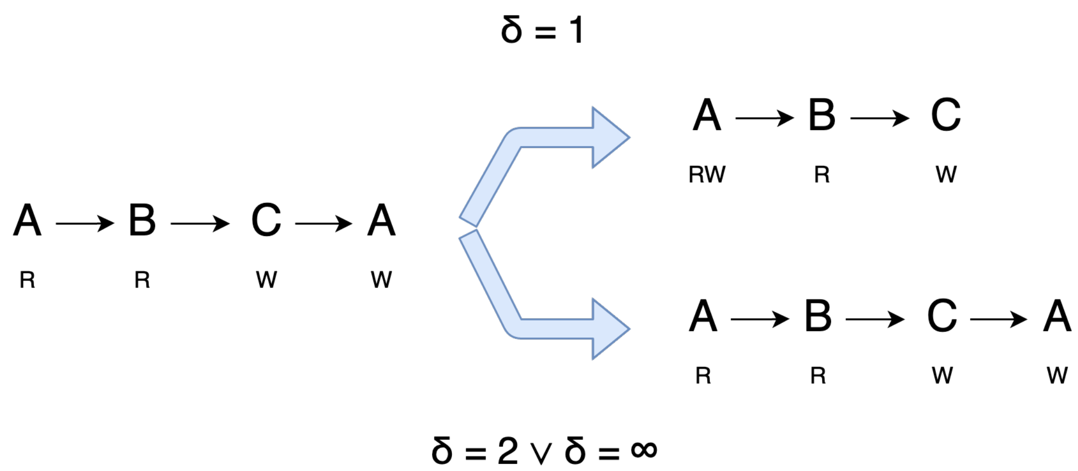

Figure 3.6: Operation of merging and pruning invocations with data dependencies, according to data dependence threshold

4

Implementation

Contents

4.1 Functional Requirements 36
4.2 Web Service 37
4.3 API Specification 42
4.4 Mono2Micro Integration 44

The main contribution of this work is a Functionality Refactorization Service[1] which given a codebase analyzed by the Mono2Micro static analyzer and a cluster decomposition, applies the algorithm presented in Chapter 3 which can estimate the Saga refactorings that minimize the migration cost of a codebase to a microservices architecture. By integrating this service with the features already provided by the Mono2Micro system, we can help a software architect on identifying the changes to make to a monolithic codebase and making it faster and effortless.

4.1 Functional Requirements

In order to establish the functional requirements for the application we first need to define the use cases for the service, in other words, how a user or programmatic client will interact with it. This interactions can be visualized in Fig. 4.1 and are listed below:

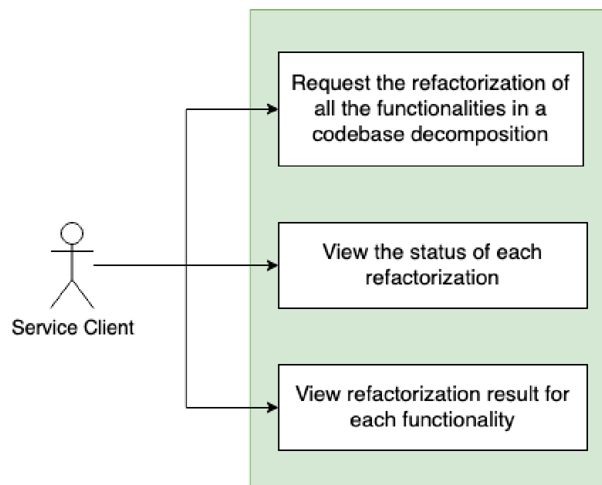

Figure 4.1: Use case diagram for the Refactorization Service.

- *Request Refactorization*: the user can request the refactorization of the controllers of a codebase's decomposition made by the Mono2Micro system. The user either specify which controllers to be refactored or refactor the entire codebase.
- *View Status*: the user can keep track of the status of the refactorization of each one of the controllers in the codebase independently.
- *View Refactorization*: the user can request to view the refactorization of each one of the controllers, which include the final complexities of the system, the orchestrator cluster chosen by the algorithm, and the sequential callgraph of the final design.

[1] https://github.com/socialsoftware/mono2micro/tree/master/tools/functionality_refactor/src

Having well-defined use cases makes it easier to set the functional requirements that will help define the service architecture that perfectly fits its purpose. With this, we identified a set of requirements that have to be met to assure the scalability, efficiency, and responsiveness of this web application:

- Support concurrency and parallelism for a faster execution time.
- Expose an API layer to receive refactorization requests.
- Have an asynchronous behavior so that the user can request a refactorization and return later to visualize the results.
- Refactorizations must respect the data dependencies inside the callgraph. A read operation followed by a write operation cannot be swapped.
- The algorithm must choose the refactorization that minimizes the sum of the FMC and SAC.

In the following sections, we present the solutions for the requirements presented above by demonstrating a brief overview of the service architecture, execution flowchart and the primary logic interfaces that perform computations, followed by the API specifications containing examples of how to use the service, and finally an explanation of how the service was integrated with the Mono2Micro system.

4.2 Web Service

The web service, Fig. 4.2 was designed to consume the data that is created when the user generates a decomposition for a given codebase, and compute the refactorization for each one of the functionalities that are valid for a *Saga* design, hence have more than 2 clusters and perform state-changing transactions. Finally, the refactorizations are stored in the file system.

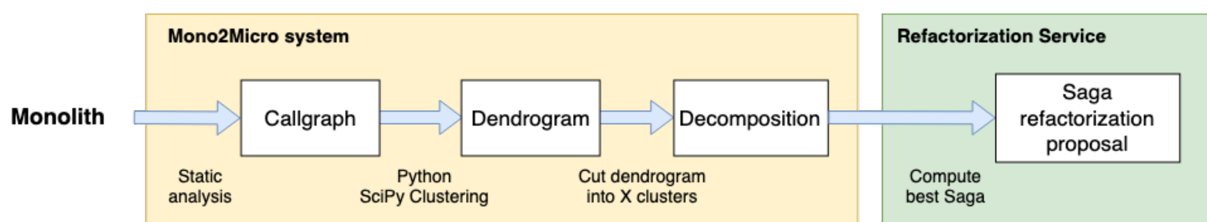

Figure 4.2: Workflow from the moment a codebase is analyzed in Mono2Micro until it is refactored by the service.

4.2.1 Architecture Overview

This service was developed in Golang[2], a programming language published in 2009 and developed by Google's Rob Pike, Robert Griesemer, and Ken Thompson. Although it is based on the syntax

[2]`https://golang.org/doc/`

of C, it comes with some changes and improvements to safely manage memory usage and provide static or strict typing. For a more efficient development process, we used the package Go-Kit[3], a programming toolkit to build microservices in Go, which provides straightforward interfaces for things like HTTP requests and Logging. The main reason for going with Golang, and not the same language as Mono2Micro, JAVA, was its speed and outstanding performance in handling concurrency, wich was one of the main functional requirements of the service.

The abstractions that provide interfaces for handling concurrency in Go are called *goroutines*[4]. One tends to find similarities in the functional behavior of threads and *goroutines*. However, they are relatively different. A *goroutine* is a lightweight method that is executed independently and simultaneously with the main routine and is managed by the Go runtime. They have a straightforward mechanism to communicate with each other with low latency, have a swift startup time, and are very cheap when consuming memory resources since they start with a stack as small as 2Kb, increasing the size only when necessary. *Goroutines* are multiplexed onto a minimal amount of OS threads which typically means programs require far fewer resources to provide the same level of performance as languages such as Java. Creating a thousand *goroutines* would require one or two OS threads at most, whereas if we were to do the same thing in Java, it would require 1,000 whole threads, each consuming a minimum of 1Mb of heap space.

Table 4.1: Comparision between the behaviour of goroutines and OS threads.

Goroutine	**Thread**
Managed by the Go runtime	Managed by the kernel
Provides built-in communication mechanisms (channels)	Communication is more difficult and results in more latency
One gorourine requires only 2Kb of allocated memory	One thread requires a minimum of 1Mb of allocated memory
Dynamic stack size	Fixed stack size
Cooperatively scheduled	Preemptively scheduled
Provides buil-in mechanism to avoid race conditions and deadlocks	The developer must handle all the logic to avoid race conditions

In addition to this, the language provides straightforward interfaces to handle the synchronism between *goroutines*. In the context of this application, one use case for this was the usage of mutex synchronization primitives to assure that two *goroutines* do not write to the same shared memory space at the same time. For instance when writing data to a file in the file system, its very easy to assure that we dont run into a race conditons by applying a mutex that only lets one *goroutine* update the file at any given time.

[3]https://github.com/go-kit/kit
[4]https://golangbot.com/goroutines/

4.2.2 Execution Flowchart

As it can be seen in Fig. 4.3 the tool starts by extracting the valid controllers of the codebase decomposition that can be implemented using a Saga Pattern.

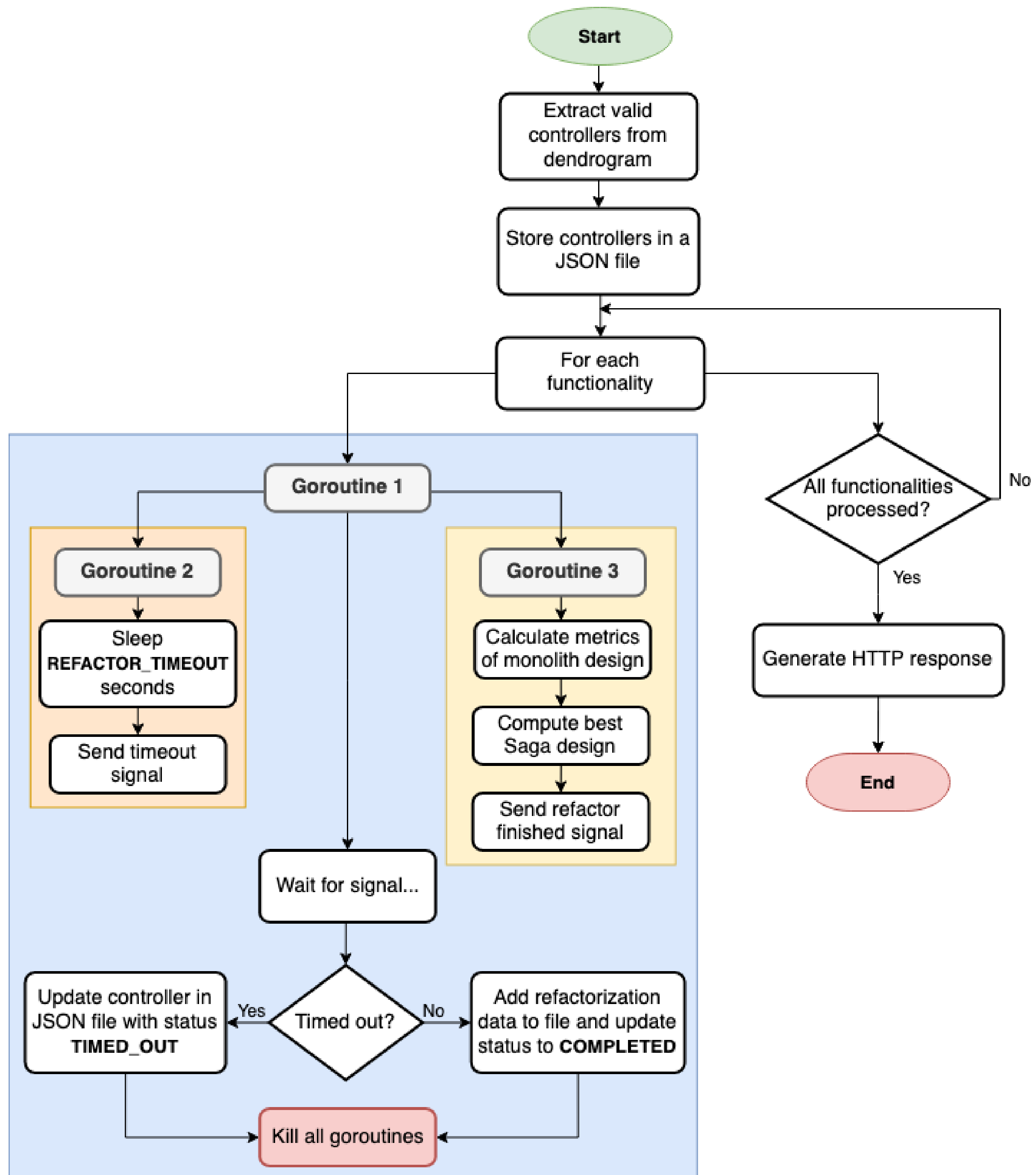

Figure 4.3: Flowchart of the high level logic of the refactorization tool.

In [7], Richardson states that Sagas only make sense when more than 2 clusters are involved, and

when the functionality involves a state-changing transaction that contains a write operation, and this is the principle that we use to filter out invalid functionalities. After this, a JSON file is written in the file system containing data about the codebase being refactored, the configuration parameters of the request, and the controllers classified as valid to be implemented as an orchestration. From this moment forward, if the user performs a *View Status* request to the service, he will receive the data in this JSON file, which will be updated as refactorizations finish.

With this, the service starts iterating through each one of the controllers and instantiates a *goroutine* to handle the computations. This way, we can achieve concurrency and parallelism, assuring that Golang's scheduler manages the routines memory allocation and uses multiple processors to reach a lower execution time. After spawning a *goroutine* for each controller, the service performs the HTTP response to the client immediately, without waiting for the refactorization results, which allows for reduced latency.

When it comes to the controller refactorization *goroutine*, in Fig. 4.3 named *Goroutine 1*, it spawns two *goroutines*, *Goroutine 3* which computes the metrics for the monolith design and then proceeds to compute the best *Saga* refactorization, and the *Goroutine 2* which acts as a sidecar to the previous one allowing for a monitorization of the whole process and killing the refactorization if a timeout is reached. This Sidecar Pattern that promotes future scalability was adopted to have more control over the asynchronous routines, so that we have control over the maximum amount of time that the goroutine responsible for the refactorizations can be alive. The first secondary goroutine (2 or 3) to finish its execution sends a signal to *Goroutine 1* which updates the status of the controller in the JSON file with either *TIMED_OUT* or *COMPLETED* and adds the results of the refactorization in the later case.

4.2.3 Business Logic Interfaces

The interface is a very common abstraction used in Golang, and it consists of a collection of methods that any given Type can implement. Hence an interface defines, but does not declare, the behavior of the Type. This pattern can be used to achieve run time polymorphism since a method call is resolved at run-time instead of compile-time, promoting more modular and decoupled code, reducing dependencies between different functionalities, and providing loose coupling.

The interface RequestHandler[5] presented in 4.1 is the main entry point to the business logic layer and is responsible for receiving data from the transport layer endpoints and execute the processing for the two available resources, so it depends on the other three interfaces presented below. It contains one method to handle the request to the RefactorDecomposition endpoint and another to handle the request to the ViewRefactorization endpoint.

[5] https://github.com/socialsoftware/mono2micro/blob/master/tools/functionality_refactor/src/app/handler/handler.go

Listing 4.1: Declaration of the RequestHandler interface

```
type RequestHandler interface {
    HandleRefactorDecomposition() //entrypoint of the RefactorDecomposition business logic
    HandleViewRefactorization() //entrypoint of the ViewRefactorization business logic
}
```

The interface FilesHandler[6] shown in 4.2 is responsible for the I/O operations with the file system. It contains methods to read and write in files such as: the codebase file generated by Mono2Micro containing the initial decomposition's callgraph and the file generated by the refactorization operation containing the result metrics and resulting callgraph of each functionality in the codebase.

Listing 4.2: Declaration of the FilesHandler interface

```
type FilesHandler interface {
    ReadCodebaseDecomposition() //read codebase decomposition file
    ReadDecompositionRefactorization() //read refactorization JSON file
    WriteDecompositionRefactorization() //write refactorization JSON file
    UpdateControllerRefactorization() //update controller in the refactorization JSON file
}
```

The MetricsHandler interface[7] shown in 4.3 is responsible for calculating the metrics described in [4, 5] for each controller, for both the initial monolithic and final *Saga* designs.

The metrics calculated by this module are used to decide which one of the SAGA's is the better fitting.

Listing 4.3: Declaration of the MetricsHandler interface

```
type MetricsHandler interface {
    CalculateControllerMetrics() //calculate complexity metrics of a controller
}
```

The RefactorizationHandler interface[8] shown in 4.4 is the main logic module of the service since it contains the algorithm responsible for the refactorization of a monolithic callgraph into a *Saga* design that minimizes the complexity of the migration.

[6] https://github.com/socialsoftware/mono2micro/blob/master/tools/functionality_refactor/src/app/files/files.go

[7] https://github.com/socialsoftware/mono2micro/blob/master/tools/functionality_refactor/src/app/metrics/metrics.go

[8] https://github.com/socialsoftware/mono2micro/blob/master/tools/functionality_refactor/src/app/refactor/refactor.go

Listing 4.4: Declaration of the RefactorizationHandler interface

```
1 type RefactorizationHandler interface {
2     RefactorDecomposition() //method that implements the refactorization algorithm
3 }
```

4.3 API Specification

In order to communicate with external services, the transport layer of the tool exposes a REST Application Program Interface (API) with support for HTTP requests. The reason for choosing the REST model was because the existing Mono2Micro project already contains specific logic modules for performing these requests, opposed to what would happen if the Refactorization Service used RPC, which required implementing an RPC client in Mono2Micro. This would increase the complexity of integrating both services.

The API contains two endpoints listed in Table 4.2. The URL for both endpoints takes as parameters the name of the codebase, the name of the dendrogram to be used, and the name of the decomposition to refactor.

Table 4.2: Available endpoints of the Refactorization Service.

Method	Endpoint	Description
GET	/codebase/*:str*/dendrograms/*:str*/decompositions/*:str*	View the refactorization results related to a codebase decomposition
POST	/codebase/*:str*/dendrograms/*:str*/decompositions/*:str*	Request the refactorization of a codebase decomposition

In 4.5 and 4.6 are presented examples of cURL requests to the ViewRefactorization and RefactorDecomposition endpoints, respectively.

Listing 4.5: Example cURL HTTP request for the ViewRefactorization endpoint

```
1 curl -X GET 'http://127.0.0.1:5001/api/v1/codebase/LdoD/dendrogram/D1/decomposition/N4'
```

Listing 4.6: Example cURL HTTP request for the RefactorDecomposition endpoint

```
1 curl -X POST -H "Content-type: application/json" -d '{
2     "controller_names": ["AdminController.deleteAllFragments"],
3     "data_dependence_threshold": 2,
4     "refactorization_timeout": 120,
5 }' 'http://127.0.0.1:5001/api/v1/codebase/LdoD/dendrogram/D1/decomposition/N4'
```

Table 4.3: Description of the JSON fields in the body of the RefactorDecomposition request.

Field	Type	Description
controller_names	Array	List of the controllers of the codebase to be refactored. If its empty the tool will refactor every valid controller.
data_dependence_threshold	Integer	Value for the data dependency threshold used in the refactorization algorithm.
refactorization_timeout	Integer	Maximum duration for the refactorization of each controller after which the respective goroutine will be killed.

Both resources respond with the same data format, which can be seen in 4.7. This response contains the refactorization data for each of the controllers in the codebase. For each controller are presented the complexity metrics for the monolithic design, and the results of the refactorization, which include: the complexity metrics of the new design, the orchestrator chosen, the sequence callgraph of the *Saga* and the values of complexity reduction when comparing to the original design.

Listing 4.7: RefactorDecomposition successful HTTP response body

```
1 {
2     "codebase_name": "LdoD",
3     "dendogram_name": "D1",
4     "decomposition_name": "N4",
5     "controllers": {
6         "AdminController.deleteAllFragments": {
7             "monolith": {
8                 "metrics": {
9                     "system_complexity": 3007,
10                    "functionality_complexity": 946,
11                    "invocations_count": 48,
12                    "accesses_count": 149
13                }
14            },
15            "refactor": {
16                "metrics": {
17                    "system_complexity": 1663,
18                    "functionality_complexity": 578,
19                    "invocations_count": 11,
20                    "accesses_count": 62
21                },
22                "orchestrator": {
23                    "name": "0"
24                },
25                "callgraph": [
26                    {
```

```
                        "cluster_id": 0,
                        "accesses": [
                            {
                                "type": "R",
                                "entity": 50
                            },
                            {
                                "type": "RW",
                                "entity":  15
                            }
                        ]
                    }
                ]
            }
        }
    }
}
```

In 4.8 its presented the response format for errors related to [400 - Bad Request] or [500 - Internal Server Error] errors. The response provides the error message so that the client receives context on the failure reason.

Listing 4.8: RefactorDecomposition error HTTP response body

```
{
    "error": <error_message>
}
```

4.4 Mono2Micro Integration

A critical task to assure the complete implementation of the Refactorization Service involves deploying it in a way that it can be used by the Mono2Micro interface and implementing a graphical user interface in Mono2Micro's React.JS web application, so that the automated refactorization is completely integrated with the existent system as an extension of the functionalities already developed, and efficiently improves the capacity of the system to help software architects on the task of migrating a monolithic codebase to the microservices architecture.

4.4.1 Deployment

The Refactorization Service stands as a separate Docker[9] container that runs in port 5001 and shares a file system volume with Mono2Micro's backend application, which is used to read the codebase files generated by the other functionalities of Mono2Micro, as can be seen in Fig 4.4.

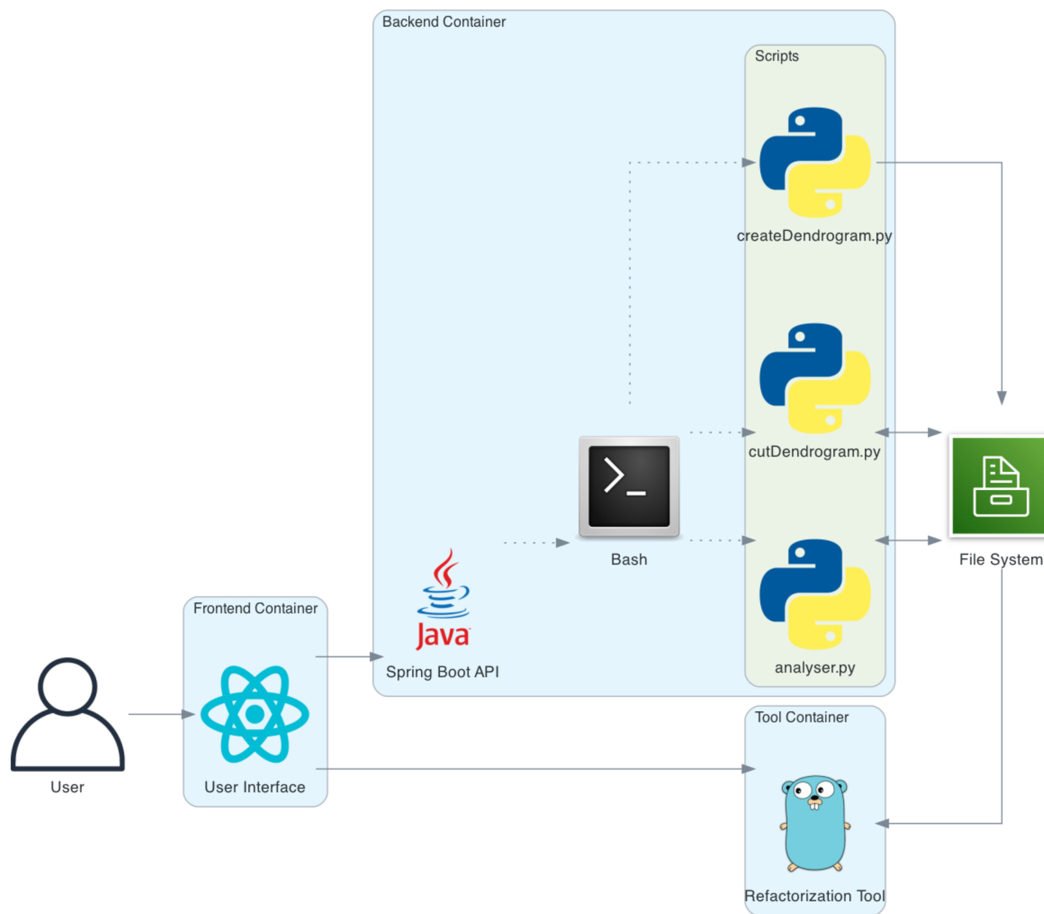

Figure 4.4: Containerized infrastructure of the Mono2Micro system integrated with the Refactorization Service.

A container is a unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings.

By implementing the frontend and backend applications of Mono2Micro as containers, the integration of the additional Functionality Refactorization Service was much more efficient since all it required was the configuration of a new container packaging the Golang application and configuring the frontend service to make requests to this container's port.

[9] https://www.docker.com/

With this architectural decision, the service became easily replicable and completely independent from the operating system of the machine it runs on, so it can be easily deployed in a local machine, a remote server, or in a cloud provider's virtual machine, such as Amazon's AWS EC2, by cloning the GitHub repository and executing the command in 4.9.

Listing 4.9: Command used to deploy all the components of the Mono2Micro system.

```
1    docker-compose up --build
```

4.4.2 User Interface

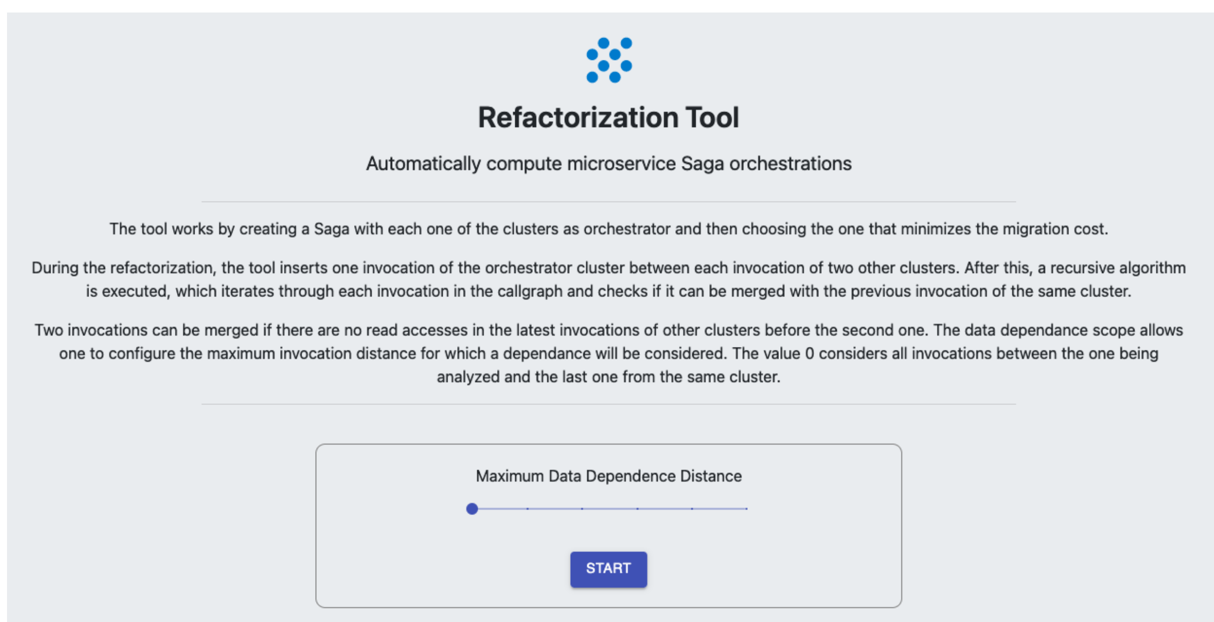

Figure 4.5: Mono2Micro dashboard that is used to interact with the Refactorization Tool.

When it comes to the graphical interface contained in the Mono2Micro system frontend application, we identified a set of requirements that needed to be met in order to interact with the Refactorization Service properly and analyze the results of the automatic refactorizations:

- The user must have some context of how the tool works.
- The user must be able to request the refactorization of a codebase and come back later to check its status.
- The user must be able to see the metrics of the monolithic design and compare them with the metrics of the *Saga* design.

- The user must be able to see the refactored callgraph so that he knows what are the easier changes to make to the controller's interactions.

In order to fill these requiremetns, was created a new static page, shown in Fig. 4.5. This interface is built using the ReactJS framework and communicates directly with the Refactorization Tool. In the top of the page, is presented the introduction to the service and the configuration section. In this section, the user can configure the data dependence threshold and start the refactorization. After the refactorization is requested, the user is free to leave the page and come back after.

UPDATE TABLE

Results of last estimation for the valid Saga controllers:

Controller	Status	Orch.	FRC	SAC	Invocations	Accesses	FRC Reduction	SAC Reduction	Merges
AdminController.deleteAllFragments	COMPLETED	0	570	1649	11	62	376	1312	39
AdminController.deleteFragment	COMPLETED	0	593	1636	12	63	369	1355	39
AdminController.deleteVirtualEdition	COMPLETED	1	946	2536	10	83	943	3187	34
AdminController.loadVirtualCorpus	COMPLETED	3	1198	3155	12	100	2004	4513	79
AdminController.removeTweets	COMPLETED	3	85	393	6	19	12	-2	3
VirtualEditionController.deleteVirtualEdition	COMPLETED	3	974	2394	8	75	977	3394	36

Figure 4.6: Dynamic table of results were each row corresponds to a functionality being refactored.

Once the functionalities refactorings are completed, one by one, their status will be updated in the table shown in Fig. 4.6, and the user will be able to instantly visualize the final complexity metrics and the complexity reductions resulting from the refactoring.

When the user clicks in a functionality rows of the table in Fig. 4.6, a dropdown pannel appears, Fig. 4.7, where it is shown the resulting Saga orchestration that minimizes the migration cost. Also, the interface shows the initial complexity metrics of the monolithic functionality. By visualizing the sequence of cluster invocations, and the domain entity accesses in each one of them, the architect is able to approach the migration task in a more efficient manner.

The user interface is responsive and responds to all the requirements that we previously set, which promotes a good user experience and assures the usage of the results from this research, in the context of the Mono2Micro software system.

AdminController.deleteVirtualEdition	COMPLETED	1	946	2536	10	83	943	3187	34
AdminController.loadVirtualCorpus	COMPLETED	3	1198	3155	12	100	2004	4513	79

Saga redesign proposed:

Cluster	Entity Accesses
3	
2	RW -> 10 | W -> 26 | RW -> 12 | W -> 35 | RW -> 62 | RW -> 32 | RW -> 11 | W -> 42 | RW -> 8 | RW -> 61 | RW -> 30 | RW -> 70 |
3	RW -> 55 |
1	W -> 20 | RW -> 71 | W -> 19 | RW -> 52 |
3	
2	RW -> 8 | RW -> 62 | RW -> 10 | RW -> 5 | W -> 42 | RW -> 12 | W -> 70 | W -> 32 | RW -> 44 | RW -> 61 | W -> 26 | RW -> 11 |
3	
1	RW -> 71 | RW -> 46 | RW -> 27 | W -> 25 | RW -> 4 | W -> 36 | W -> 9 | W -> 56 | RW -> 52 | RW -> 48 | RW -> 29 | W -> 19 | RW -> 31 | W -> 20 | RW -> 40 |

Figure 4.7: Refactorization dropdown which demonstrates the proposed Saga.

5

Evaluation and Analysis

Contents

5.1 Complexity Reduction Analysis 50
5.2 Refactorization Accuracy Analysis 52
5.3 Orchestrators Characterization 55
5.4 Performance Analysis 56
5.5 Summary 58
5.6 Threats to Validity 59

To answer the first research question and validate that refactorings can minimize the cost associated with the migration, we executed the tool for a dataset of 78 codebases and analyzed the results to evaluate the reduction of complexity. To validate the accuracy of refactorizations we compared with the refactorizations done by an expert [5] on the LdoD codebase in terms of complexity reduction and feasibility.

5.1 Complexity Reduction Analysis

The 78 codebases[1] are monolith systems implemented using Spring-Boot and an Object-Relational Mapper (ORM), where Java Persitence API (JPA) is used in 75 of the codebases and Fénix Framework[2] in the other 3. The Spring-Boot controllers are used to implement the monolith transactional functionalities that access the persistent domain entities implemented using the ORM. The data set was generated through static analysis of the source code, where sequences of read/write accesses to the domain entities are obtained for the functionalities.

Each codebase was decomposed into clusters, and, after applying the initial partition to the functionalities sequences, the functionalities were selected based on two conditions: (1) the sequence of accesses has least one write access; (2) the sequence of accesses includes more than two local clusters. These are pre-conditions for the implementation of the functionalities as *SAGA*s, by excluding queries and functionalities that can not be implemented as *SAGA*s because only have one or two clusters. After these selections, the dataset was formed by 652 functionalities.

Table 5.1: Average values for running the recommendation heuristic for 652 functionalities, considering $\delta = 1, 2, \infty$.

Metric	Initial	Final			% Reduction		
		$\delta = 1$	$\delta = 2$	$\delta = \infty$	$\delta = 1$	$\delta = 2$	$\delta = \infty$
FMC	329.0	78.4	86.7	95.6	76.1%	73.6%	70.9%
SAC	404.9	150.7	170.6	201.3	62.7%	57.9%	50.3%
Local Transactions	35.5	4.3	4.7	5.1	87.9%	86.8%	85.6%
Entity Accesses	38.7	10.3	11.3	12.4	73.4%	70.8%	67.9%
Number of Merges		22.5	21.7	21.4			
Execution time in seconds		3.76	4.13	4.40			

Table 5.1 presents the results, on average, of applying the refactoring heuristic to the 652 functionalities. It compares the initial complexity with the complexity of the refactored functionalities. The heuristic was applied for three variations of the data dependencies between local transactions, where $\delta = 1$ means that there is no data dependence with the local transaction that executed immediately before,

[1] github.com/socialsoftware/mono2micro/tree/master/tools/functionality_refactor/codebases
[2] https://fenix-framework.github.io/

$\delta = 2$ means that there no data dependence with the two local transactions that executed immediately before, and $\delta = \infty$ that there is no data dependence with any local transaction that executes in between the data transactions to be merged. It is necessary to analyze these cases because the data dependencies are obtained from an analysis of the sequences and this may not necessarily mean that a local transaction effectively uses the data read on previous local transactions. For instance, if there is a write access in a local transaction we consider that data dependencies exist with all the local transactions that occur before and that have at least one read access, because the value read may be used to calculate the value that is going to be written. This is a limitation of the static analysis procedure that captures the monolith behavior, which does not capture the data-flow, and so we considered the worse situation in terms of data dependencies.

The table also presents the percentage of reduction in the number of local transactions and domain entities access. Additionally, it includes the average number of merges, and the heuristic performance, which was calculated running 10 samples on an Intel i5 2.90GHz, 2 Cores with 4 Threads each.

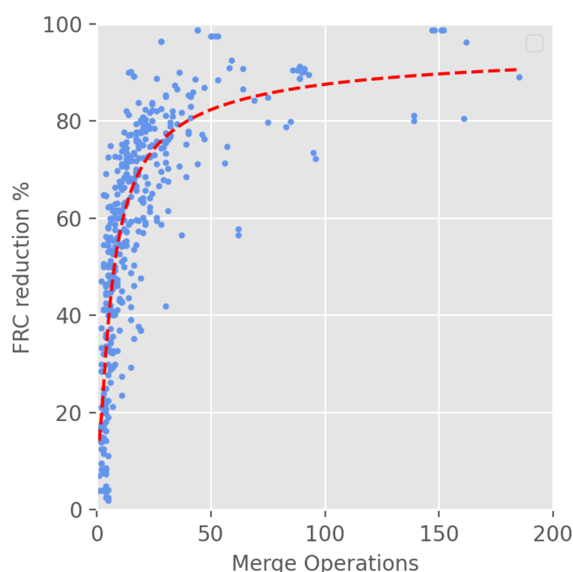

Figure 5.1: Percentage reduction of the Functionality Migration Complexity in function of the number of merge operations made.

From the observation of the Table 5.1 and Fig. 5.1 it be concluded that there is a significant positive impact of the refactorings on the complexity reduction and that the scope of data dependencies also impacts on final complexity, as expected a larger scope will reduce the number of possible merges, but it is not significant, which means that in most of the functionalities there is a data dependence between a local transaction and the local transaction that executes immediately before. It is also worth mentioning the reduction in the number of domain entities access, which results from the fact the access are localized inside the same local transactions and, so, the visibility of their effects to other local transactions is

reduced.

The analysis of the codebases allows us to partially answer the first research question, due to the reduction in the complexity, but it is necessary to analyze the accuracy of the results: are the recommended refactorings feasible?

5.2 Refactorization Accuracy Analysis

The tool was applied to the LdoD archive[3], which consists of 133 controllers and 67 domain entities. The decomposition used was done by an which cuts the system into 5 clusters. The analysis involved comparing the results with the refactorings of 9 controllers of LdoD [5], where the expert refactored the code and calculated the initial and final Functionality Migration Complexity (FMC) and System Added Complexity (SAC). We do not compare the SAC complexity because it depends on the type of local transaction, which the algorithm cannot predict. For instance, whether a local transaction is compensatable or not. The heuristic always considers the worst-case scenario.

Table 5.2: Functionality Migration Complexity reduction resulting from a refactoring using the tool with $\delta = 1$ and by an expert.

Functionality	**Initial FMC**		**Final FMC**		**Reduction %**		**Orchestrator distance**
	Tool	Expert	Tool	Expert	Tool	Expert	
removeTweets	151	134	109	82	27.8%	38.8%	0%
getTaxonomy	317	317	159	192	49.8%	39.4%	1.24%
createLinearVE	2978	1790	263	383	91.1%	78.6%	0%
signup	1550	1490	314	376	79.7%	74.8%	0%
approveParticipant	193	190	113	147	41.5%	22.6%	0%
associateCategory	1813	1803	642	662	64.6%	63.3%	14.97%
deleteTaxonomy	261	253	187	164	28.4%	35.2%	11.79%
dissociate	806	772	358	489	55.6%	36.7%	0%
mergeCategories	485	453	187	253	61.4%	44.2%	37.87%
Averages	950.4	800.2	259.1	305.3	55.5%	48.2%	

Table 5.2 presents the results when the heuristic was applied with a data dependence scope of 1. It can be observed a clear relation between the complexity reduction of the functionalities when refactored by the heuristic algorithm and by the expert, since the average reduction from both differs only 7.4%, with the tool being able to achieve bigger complexity reductions on average. In the last column, we present the relative distance between the complexity of the best refactoring calculated by the tool and the complexity of the refactoring with the same orchestrator as chosen by the expert. As it can be seen

[3] `https://github.com/socialsoftware/edition`

in cells with a 0% distance value, 5 out of the 9 functionalities, had both the expert and the tool selecting the same orchestrator. Note that the value is 0% because orchestration distance comparison is not done with the complexity values reported by the expert, but with the values calculated by our tool.

After having the best refactoring estimated by the algorithm, the next step of the evaluation involved manually verification in the source code, to check if the recommended orchestrations are feasible, for instance, that they do not break any data dependence between entity accesses.

The selection process of the functionalities was based on the values from Table 5.2 and the functionalities selected fall in one of the following categories:

- The tool recommendation has a bigger complexity reduction.
- The expert refactoring has a bigger complexity reduction.
- The orchestrators selected by the tool and the expert are different.
- The initial complexity value calculated by the tool is much different than the one calculated by the expert.

This allows us to identify 6 cases where to do this analysis.

The refactoring of the *mergeCategories* functionality has a complexity reduction of 62.7% when compared to the initial architecture and converted 32 independent cluster invocations to 5. The algorithm chose a different orchestrator than the expert and achieved a complexity 18.5% lower. After manually checking the source code, we verified that this refactor is a valid option, all the operations are valid and the data dependencies that exist in the functionality are all taken into consideration. We can see that the tool computed fewer cluster invocations, fewer entity accesses, and fewer write operations, which reduces the complexity compared to the expert refactor. By looking at the code, we can see that the expert choice was based on selecting the cluster that contains more entity accesses as the orchestrator, which resulted in more repetitive invocations of other clusters. By choosing an orchestrator that was not so obvious at first glance, the algorithm was able to obtain a better refactor. The same happened for the functionality *getTaxonomy*, where the tool recommend a different orchestration than the expert, but it was able to obtain an higher reduction of the complexity by having all the operations condensed in 1 invocation per cluster, while the expert refactorization created more intermediate states. The source code also revealed that no data dependencies were broken by the tool's operations, and the sequence is applicable.

In the functionality *approveParticipant*, both, the tool and the expert, selected the same orchestrator cluster and did a very similar refactorization, where only 1 additional merge operation by the tool was enough to reduce the complexity in 41.5% when compared to the expert's 22.5%. The functionality *dissociate* falls in this same category since it shows that the tool was able to achieve a lower complexity

with the same orchestrator, however by checking the source code we see that some invocations were merged while having a data dependence with previous invocations, which reveals invalid merges. This occurred because the data dependence distance had a scope > 1 and due to the test being executed with $\delta = 1$ the tool did not consider it when verifying if an invocation can be merged.

The refactor of *removeTweets* saw a complexity reduction of 26.8% when compared to the initial architecture and converted 14 independent cluster invocations to 7. Both the algorithm and the expert chose the same orchestrator and performed a very similar refactor but the expert refactor achieves a higher complexity reduction. After manually checking the source code, we verified that this refactor is a valid option, except for one invocation that could have been merged with a previous one. Since in the original trace there was a data dependence between that operation and the one immediately before, the tool did not do the merge. This behavior reveals that the heuristic rules can be too restrictive and since the static analysis is not capturing the data-flow, we cannot accurately conclude that a write following a read does not use data from the first, which is something that an expert can directly observe by reading the code. At most, this rule follows the worst-case scenario since by not breaking data dependencies, we are sure that the refactorization will work. However, it is possible that the result of the final refactoring is not the one that has the lowest migration cost.

In functionality *createLinearVE* there is a significant difference of 1188 between the tool's initial complexity and the one calculated manually by the expert. This functionality is quite complex, with 108 local transactions and 210 entity accesses, and from looking at the source code, it involves many if/else conditions that define which accesses are performed during run-time. Ultimately, this difference relates to how the static analysis builds the functionality trace since it does not account for conditions and so all branches are collapsed in a single sequence, as is seen in Fig. 5.2. When the expert manually calculates the complexities, she only considers one of the if/else conditions, which results in fewer entity accesses and lower initial complexity.

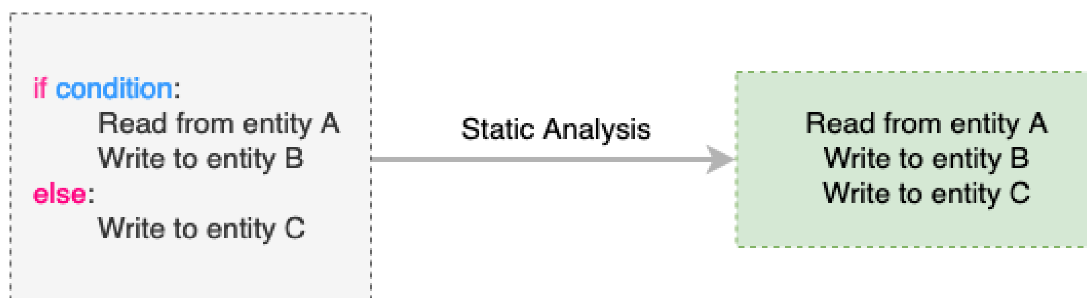

Figure 5.2: Behavior of the static analysis step when encountering conditional branching.

To answer the first research question fully, we can say that most of the refactorizations applied to LdoD were feasible. However, there was at least 1 case where data dependencies were broken due to the configuration of the data dependence scope δ as 1. This difficulty when defining thresholds is still one

of the major drawbacks of most heuristic techniques for code-smell and design-pattern detection [20].

5.3 Orchestrators Characterization

The validation of the refactorization of the LdoD codebase, when compared to the expert, has shown that the recommended refactorings highly reduce the complexity of migrating the functionalities and they, in most of the cases, even when the scope is 1, do not break the data dependencies between the accesses. Another interesting observation is that the heuristic can suggest better refactors than the ones envisioned by the expert. To try to explain why the experts intuition may be misleading, take us to the second research question: what are the characteristics of a cluster that make it a better fit for being a *SAGA* orchestrator.

We defined 4 metrics that measure, for each cluster and in the context of the monolithic functionality being refactored, several aspects, like the number of read and write accesses to the cluster domain entities, the number of times the cluster is invoked in the context of the functionality, and the number of the invocations that cannot be merged due to a data dependence.

Table 5.3: Correlation between the metrics for the orchestrator cluster and the reduction of the functionality migration complexity and system added complexity

Metric	**Correlation (r)**		
	FMC	SAC	FMC+SAC
Number of read accesses to domain entities	0.145	-0.083	0.014
Number of write accesses to domain entities	-0.145	0.083	-0.014
Number of times the cluster is invoked in functionality	-0.103	-0.090	-0.112
Number of initial data dependencies, when $\delta = \infty$	-0.110	0.210	0.087

These metrics were visualized for the orchestrators of each one of the recommended refactorizations for 78 codebases and correlations were calculated between the value of the metrics and the reduction of the complexity. Table 5.3 shows the resulting correlations with values between $[-0.145, +0.210]$, which demonstrates that the data has a big variation and does not necessarily follow a trend line, with points that are too dispersed in space. The metric with higher correlation was related to the metric for reads accesses of orchestrators, but, even though, not significative.

One of the metrics that showed a more considerable correlation was the number of read accesses of the cluster when compared to the number of read accesses in the other clusters of the functionality, which can be measured by calculating the probability of a read access being in that cluster. By looking at Fig. 5.3 we see a pattern where bigger FMC reductions were related with a more significant probability of having read accesses, which is inversely proportional to the probability of write accesses. This leads

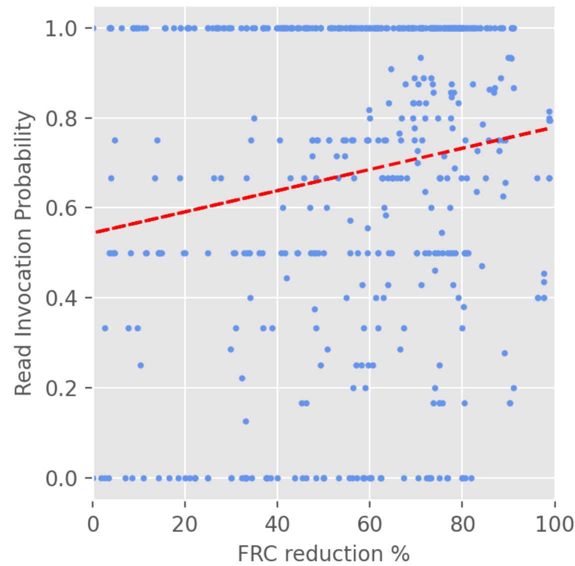

Figure 5.3: Probability of the orchestrator cluster having a read access in the initial monolithic functionality, in function of the final FMC reduction.

us to believe that better orchestrators have a greater chance of being the ones with fewer write accesses to their entities. However, the results show a considerable deviation, and even though a regression can be drawn, there are many clusters with a considerable distance from it.

The low correlation values show that the metrics cannot characterize the orchestrator, due to the great diversification of code patterns in the functionalities. A good orchestrator for a given codebase might have a lot of invocations, for example, while in another codebase we verified that the opposite may happen, which makes it hard to set a characterization for an orchestrator that applies to all the cases.

To answer the second research question we can conclude that it was not possible to extract enough metrics from the functionality sequence to characterize what is a good orchestrator. This increases the need of the recommendation heuristic tool which applies refactorization operations and validates if a given candidate orchestrator has the lowest complexity.

5.4 Performance Analysis

The performance analysis of the tool was done in the context of the large dataset of 78 codebases and considering 3 values for the data dependence threshold to evaluate how the tool behaves for different configurations. The set of machines used for the performance analysis are shown in Table 5.4 and consist of the local machine of the author, and two Amazon Web Services EC2 dedicated instances that were started exclusively to test the tool, and did not share resources with other users of the cloud service.

Table 5.4: Machines used to test the Refactorization Tool.

Machine	CPU		
	Model	Frequency	Cores
Local	Intel i5	2.90 GHz	4
AWS m5.xlarge	Intel Xeon Platinum 8000 (Skylake)	3.1 GHz	4
AWS c5.4xlarge	Intel Xeon Scalable Processors (Cascade Lake)	3.4 GHz	16

The test results were extracted by refactoring all of the codebases in the dataset ten times for each value of the data dependence threshold and each one of the machines. Each metric is the result of averaging the same metric for all ten executions.

In Tables 5.5, 5.6 and 5.7 we present, for each machine and data dependence threshold, the average, lowest and highest duration, both for refactoring a complete codebase and for refactoring a single functionality in any codebase of the dataset.

Table 5.5: Execution times when refactoring the dataset in the **Local** machine.

Data Dependence	Codebase Refactor			Functionality Refactor		
	Average	Lowest	Highest	Average	Lowest	Highest
$delta = 1$	4080ms	1.40ms	64520ms	4440ms	1.38ms	64520ms
$delta = 2$	3340ms	1.38ms	63050ms	3690ms	1.35ms	63030ms
$delta = \infty$	2790ms	1.49ms	68430ms	3060ms	1.47ms	68320ms

Table 5.6: Execution times when refactoring the dataset in the **AWS m5.xlarge** machine.

Data Dependence	Codebase Refactor			Functionality Refactor		
	Average	Lowest	Highest	Average	Lowest	Highest
$delta = 1$	2170ms	1.05ms	52660ms	2160ms	1.03ms	52660ms
$delta = 2$	2210ms	0.98ms	52940ms	2190ms	0.96ms	52940ms
$delta = \infty$	2140ms	1.05ms	52.84s	2130ms	1.03ms	52780ms

Table 5.7: Execution times when refactoring the dataset in the **AWS c5.4xlarge** machine.

Data Dependence	Codebase Refactor			Functionality Refactor		
	Average	Lowest	Highest	Average	Lowest	Highest
$delta = 1$	1100ms	0.86ms	21910ms	1040ms	0.84ms	21910ms
$delta = 2$	1100ms	0.86ms	21800ms	1050ms	0.84ms	21800ms
$delta = \infty$	1060ms	0.86ms	21670ms	1020ms	0.84ms	21670ms

As expected, the value of $delta = 1$ results in higher refactorization times since this is the case where the algorithm has more freedom to execute sequence change operations. Since fewer data dependencies are considered, this results in more recursive iterations of the merge-invocations method

described in Chapter 3. On the other hand, the value $delta = \infty$ results in the lowest execution times both for refactoring a complete codebase and for refactoring a single functionality, since, with this threshold value, the tool will consider more data dependencies and will not be able to execute as many sequence change operations as it would with another value.

By evaluating the metrics, there is a relationship between the number of CPU cores in the machine and a lower refactorization duration for each codebase and its functionalities. This is a very CPU intensive application that requires a lot of logical computations, so a more powerful CPU like the one in the AWS c5.4xlarge machine will ultimately lower the execution time.

When it comes to the parallelism of the tool, it is interesting to note that the codebase that took the longest to refactor matches the functionality that also took the longest to refactor. This behavior clearly shows the parallelism of the application, since while the most extensive functionality is refactoring, all the other ones are finishing their refactor, so the total amount of time that the tool takes to compute all the Sagas in a codebase is equal to the time it takes to compute it for the heaviest functionality.

Table 5.8: Cumulative bytes allocated for heap objects during the refactorization of the 78 codebases dataset.

Data Dependence	Allocated Memory
$delta = 1$	10377 Mib
$delta = 2$	10594 Mib
$delta = \infty$	10009 Mib

In Table 5.8 is presented the total amount of memory that the tool allocated in the heap of the machine during the refactorization of the dataset. This metric has the same result for all three machines since the amount of memory allocated does not depend on the hardware specifications. Also, it increases as heap objects are allocated but does not decrease when objects are freed, so it is a purely cumulative value. On average, the tool allocated a total of roughly 10 Gb to refactor the dataset of 78 codebases. Looking at these numbers, we are sure that the execution times of are quite acceptable and are low enough for it to be used by a software architect.

5.5 Summary

In summary, the analysis has shown that:

- There is a very positive impact on the complexity reduction when analyzing the refactorings made by the service.
- The service is able to reduce both the number of external requests between microservices and the number of domain entities access.

- The impact of changing the scope of data dependencies is not significant, since in most of the functionalities there is a data dependence between a local transaction and the local transaction immediately before, so increasing the value won't change the proposed refactorization.
- Most of the refactorizations analysed in depth are feasible, however we have seen cases where a wrong configuration of the data dependence scope resulted in breaking changes.
- We were not able to determine metrics that can characterize possible good orchestrators in the monolithic version of the controllers.
- The performance analysis of the tool for a large codebase showed that the algorithm is very efficient when it comes to the time it takes to refactorize a codebase.

5.6 Threats to Validity

In terms of internal validity, it is verified that the refactorings provide a significant reduction of the complexity, even though the data dependencies are inferred from the analysis of the sequence of accesses and not directly obtained from the code. This may have impact on the accuracy of the results, but the experiment with variations of the scope of data dependencies has shown consistent results. Therefore, even if the architect uses the wider scope, the recommended refactorings do not have a significant variation on the complexity. The sequences of accesses used for the validation, obtained from static analysis linearize all the accesses of a functionality in a single sequence. However, we obtained results similar to the expert refactorizations, sometimes even better. This also depends on the used dataset, a future version of that static analyser that captures the data-flow will allow to work with a dataset that will provide more precise results.

In terms of external validity, the dataset is for a small set of technologies, Spring-Boot and JPA, but the functionalities logic is technology independent.

6

Conclusion

The research documented in this thesis proposes a heuristic approach to help software architects on the task of identifying the possible refactorings of monolithic functionalities, and reducing their migration cost to a microservices architecture by applying the *SAGA* pattern that minimizes the total system complexity.

The results have shown that the Refactorization Service could recommend refactorings that significantly reduce the functionalities migration cost. Interestingly, some recommended refactorings provide higher reductions than those identified by an expert in previous research, which cements, even more, the method efficiency.

A study with four metrics that characterize the clusters accessed by the functionality has shown that there is no statistical significance in the correlation between the values of the metrics and the reductions in complexity. These results suggest that it is really hard to tell, by looking at initial metrics and without any computation of refactoring, which one of the clusters that participate in the *Saga* is the one that will promote a more significant reduction of complexity when acting as orchestrator. The results also suggest that more complex metrics need to be created and extracted from the initial design, should a machine-learning approach be implemented.

The good results on the tests to the Refactorization Service, allied to a low correlation between the initial metrics and the best orchestrator for any given *Saga*, and the fact that it is so challenging to create quality datasets to train a machine-learning classification model indicate that an efficient heuristic approach which calculates all refactorization combinations fits these use cases better for now.

An important task to complete in future work is improving the static or dynamic analysis that performs the data collection in the monolithic codebase, so that even more valuable data can be added to the sequential callgraphs. Currently, the data dependencies between local transactions are not being captured, but only which domain entities are being accessed in each cluster. This behavior results in the current implementation of the refactorization algorithm, where the data dependencies are estimated given a configuration threshold representing the maximum distance between the current local transaction and a previous read operation. This approach leaves to the software architecture the decision on which threshold to use, and leads to a more strict refactorization operation as we saw for some of the functionalities in LdoD where some Sequence Changes could have been done but were not, due to wrongly implicitly inferred data dependencies. By extracting which fields are being read and written in the domain entities, the Refactorization Service would be able to create a more detailed map of the data accesses in each local transaction and understand if what is being read in a domain entity on a cluster is necessary for writing into another domain entity on another cluster.

Also, more profound research can be done on studying possible metrics in order to try to characterize better what is a good candidate for a *Saga* orchestrator while trying to discover a pattern behavior in monolithic codebases for this types of clusters.

Bibliography

[1] M. Fowler, “Microservices.” [Online]. Available: http://martinfowler.com/articles/microservices.html

[2] ——, “Monolithfirst.” [Online]. Available: https://martinfowler.com/bliki/MonolithFirst.html

[3] L. Nunes, N. Santos, and A. R. Silva, “From a monolith to a microservices architecture: An approach based on transactional contexts,” *Lecture Notes in Computer Science*, vol. 11681, p. 37–52, 2019.

[4] N. Santos and A. R. Silva, “A complexity metric for microservices architecture migration,” 2020.

[5] J. F. Almeida and A. R. Silva, “Monolith migration complexity tuning through the application of microservices patterns,” *Lecture Notes in Computer Science*, vol. 12292, pp. 39–54, 2020.

[6] M. Fowler, “Break monolith into microservices.” [Online]. Available: https://martinfowler.com/articles/break-monolith-into-microservices.html

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

[8] H. Garcia-Molina and K. Salem, “Sagas,” in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’87. New York, NY, USA: Association for Computing Machinery, 1987, p. 249–259. [Online]. Available: https://doi.org/10.1145/38713.38742

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

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

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

[12] 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*, pp. 1–1, 2019.

[13] M. H. Gomes Barbosa and P. H. M. Maia, "Towards identifying microservice candidates from business rules implemented in stored procedures," in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2020, pp. 41–48.

[14] M. Daoud, A. E. Mezouari, N. Faci, D. Benslimane, Z. Maamar, and A. E. Fazziki, "Automatic microservices identification from a set of business processes," in *Smart Applications and Data Analysis*, M. Hamlich, L. Bellatreche, A. Mondal, and C. Ordonez, Eds. Cham: Springer International Publishing, 2020, pp. 299–315.

[15] A. Selmadji, A. Seriai, H. L. Bouziane, R. Oumarou Mahamane, P. Zaragoza, and C. Dony, "From monolithic architecture style to microservice one based on a semi-automatic approach," in *2020 IEEE International Conference on Software Architecture (ICSA)*, 2020, pp. 157–168.

[16] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 36, pp. 20–36, 2010.

[17] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 268–278.

[18] F. Palomba, A. Panichella, A. Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," 05 2016.

[19] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *2009 Ninth International Conference on Quality Software*, 2009, pp. 305–314.

[20] F. Pecorelli, F. Palomba, D. D. Nucci, and A. D. Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," *ICPC '19: Proceedings of the 27th International Conference on Program Comprehension*, vol. 11681, pp. 93–104, 2019.

[21] F. A. Fontana, A. Caracciolo, and M. Zanoni, "Dpb: A benchmark for design pattern detection tools," in *2012 16th European Conference on Software Maintenance and Reengineering*, 2012, pp. 235–244.

[22] M. Zanoni, F. A. Fontana, and F. Stella, "On applying machine learning techniques for design pattern detection," *The Journal of Systems and Software*, vol. 103, pp. 102–117, 2015.

[23] ——, “Comparing and experimenting machine learning techniques for code smell detection,” *Empirical Software Engineering*, vol. 21, p. 1143–1191, 2016.

[24] M. Zanoni and F. A. Fontana, “Code smell severity classification using machine learning techniques,” *Knowledge-Based Systems*, vol. 128, pp. 43–58, 2017.

[25] S. Ma, Y. Chuang, C. Lan, H. Chen, C. Huang, and C. Li, “Scenario-based microservice retrieval using word2vec,” in *2018 IEEE 15th International Conference on e-Business Engineering (ICEBE)*, 2018, pp. 239–244.

[26] M. Abdullah, W. Iqbal, and A. Erradi, “Unsupervised learning approach for web application auto-decomposition into microservices,” *Journal of Systems and Software*, vol. 151, pp. 243–257, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121219300408