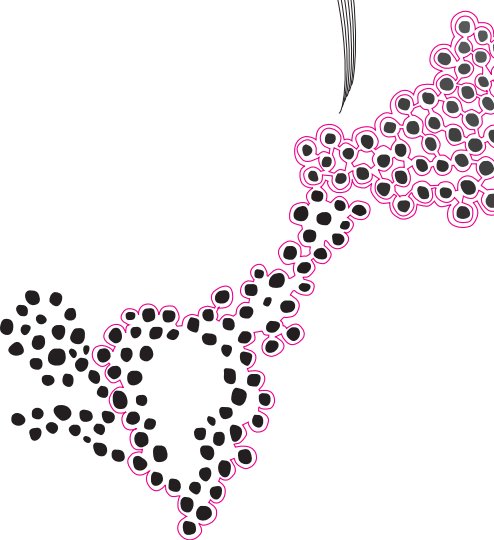




MSc Computer Science
Final Project

A quantitative assessment method for microservices granularity to improve maintainability

Famke Driessen



Supervisors: Luís Ferreira Pires, João Luiz Rebelo Moreira, Anna Sperotto, Sander van den Bosch, Paul Verhoeven

June, 2023

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

Contents

List of Abbreviations	4
List of Tables	5
List of Figures	6
1 Introduction	1
1.1 Motivation	1
1.2 Problem Statement	2
1.3 Research Questions	3
1.4 Approach	3
1.5 Report Structure	4
2 Microservices Granularity	5
2.1 Microservice Architecture	5
2.2 Granularity in MSA	6
2.2.1 (Lack of) Definition	6
2.2.2 Influence on microservice quality	7
2.3 Maintainability Perspective	10
2.3.1 Definition	10
2.3.2 Maintainability in MSA	11
2.4 Metrics	12
2.4.1 Size metrics	13
2.4.2 Coupling metrics	15
2.4.3 Cohesion metrics	20
3 Research Strategy	22
3.1 Overview	22
3.2 Case Selection Requirements	23
3.3 Tooling	24
3.3.1 Instrumentation	24
4 Data Collection	28
4.1 Case 1: Metadata	28
4.1.1 R1.1	29
4.1.2 R1.2	29
4.2 Case 2: Loan eligibility checker	30
4.2.1 R2.1	31
4.2.2 R2.2	32
4.2.3 R2.3	32

4.2.4	R2.4	33
4.2.5	R2.5	33
4.3	Case 3: Spinnaker	34
4.4	Cases Overview	35
4.5	Data Preparation	37
5	Data Analysis	45
5.1	Merges	45
5.2	Decompositions	47
5.3	Hybrid refactors	49
5.4	Spinnaker	51
6	Validation	53
6.1	Assessment Observations	53
6.1.1	Merges	55
6.1.2	Decompositions	57
6.1.3	Hybrid Refactors	60
6.2	Validation Interviews	63
6.2.1	R1.1	63
6.2.2	R1.2	64
6.2.3	R2.1	64
6.2.4	R2.2	64
6.2.5	R2.3	65
6.2.6	R2.4	65
6.2.7	R2.5	65
6.2.8	Spinnaker	66
6.3	Discussion	67
6.3.1	Implications	67
6.3.2	Prioritising refactoring candidates	70
6.3.3	Research limitations	72
7	Final remarks	74
7.1	Related Work	74
7.2	Conclusion	76
7.2.1	Contributions to Research	78
7.2.2	Contributions to Industry	78
7.3	Future Work	79
	References	80
	Appendices	87
A	Research Planning	88

List of Abbreviations

API	Application Protocol Interface
CC	Change Coupling
CC_{A→B}	Directed Change Coupling from A to B
CD	Continuous Delivery
CF	Change Frequency
CI	Continuous Integration
DDD	Domain-Driven Design
GWF	Global Weighting Factor
IQR	Inter-Quartile Range
LOC	Lines of Code
LWF	Local Weighting Factor
MSA	Microservice Architecture
PSD2	Payment Services Directive
REST	Representational state transfer
RX.Y	Refactor Y from case X
SC	Structural Coupling
SIDC	Service Interface Data Cohesion
SME	Small Medium Enterprise
SOA	Service Oriented Architecture
SVN	Apache Subversion
SX.Y.Z	Service Z, involved in refactor Y from case X
TW	Temporal window
UML	Unified Modelling Language
WSIC	Weighted Service Interface Count

List of Tables

2.1	WSIC thresholds proposed by [Bogner et al., 2020]	15
2.2	SIDC thresholds proposed by [Bogner et al., 2020]	21
4.1	Cases overview	36
4.2	Summary of Data Preparation guidelines	43
5.1	Coupling metrics of R2.2 and R2.3	45
5.2	Average metric values for entire application (R2.2 and R2.3)	46
5.3	Cohesion and size metrics of R2.2 and R2.3	47
5.4	LOC values of the services involved in R2.2 and R2.3	47
5.5	Coupling metrics of R1.1 and R2.4	48
5.6	Average metric values for the entire application (R1.1 and R2.4)	48
5.7	Cohesion and size metrics of R1.1 and R2.4	49
5.8	Coupling metrics for R1.2, R2.1 and R2.5	50
5.9	Average metric values for entire application (R1.2, R2.1 and R2.5)	50
5.10	Cohesion and size metrics for R1.2, R2.1 and R2.5	51
5.11	Coupling metrics for a selection of the service pairs of Spinnaker	52
5.12	Cohesion and size metrics for the services S3.0.4, S3.0.1 and S3.0.3	52
5.13	Average metric values for entire Spinnaker application	52
6.1	Framework supporting the interpretation of the metric values in different refactor contexts	54
6.2	Relation between the assessment outcomes and the expert's observations for the analysed merges	56
6.3	Relation between the assessment outcomes and the expert's observations for the analysed decompositions	58
6.4	Relation between the assessment outcomes and the expert's observations for the analysed hybrid refactors	61
6.5	Effect of different temporal windows on change coupling	68

List of Figures

2.1	Example of an ordering system [Newman, 2021]	10
2.2	Verbosity of different programming languages [Goebelbecker, 2022]	14
2.3	Evolution of three software artefacts over time	16
2.4	Example of a co-change matrix [Oliva and Gerosa, 2015]	18
2.5	Sliding window implementation which increases the number of change sets	20
3.1	Steps of our research strategy	23
4.1	Overview of Metadata’s architecture [Visockis, nd]	29
4.2	Microservices architecture of the lending system	31
4.3	Overview of the Spinnaker microservices [Spinnaker, a]	35
4.4	Illustrative example of a refactor	37
4.5	Fictitious service evolving into two other services	40
6.1	Boxplot calculated over the change coupling values pre-R2.3.	71

Abstract

Microservice architecture is a widely-adapted architectural style for software systems known for its advantages in terms of scalability and maintainability. The granularity of such architectures is of key importance, as inappropriate granularity levels can result in issues such as data consistency problems, increased global complexity, and reduced reusability of individual components. Therefore, determining a suitable granularity level is crucial. Currently, practitioners make decisions on microservice granularity by identifying bounded contexts, a concept from domain-driven design. This identification is done mainly based on experience. This can be limiting as such experience is not always at hand, and more concrete decision support is lacking. To ultimately develop concrete decision support for microservices granularity decisions, we developed an approach which allows for an objective, quantitative assessment of the quality of a granularity of a microservice architecture. The assessment method focuses on the maintainability of a system, as this is strongly influenced by granularity and critical for a project to succeed.

To develop this method, a set of six maintainability metrics was selected, tailored for microservice-based systems. This selection consisted of the following metrics: change coupling, structural coupling, weighted service interface count, lines of code, service interface data cohesion and change frequency. These metrics were derived from literature on maintainability in microservice architecture and service oriented architecture.

To evaluate our assessment method, three microservice-based software projects were selected. Involved engineers were interviewed on refactors that had been carried out or were planned to improve maintainability, which affected the granularity of the system. Subsequently, we performed our assessment on different versions of the systems, in order to investigate what kind of trend in maintainability our assessment method identified. We validated our assessment method by investigating the extent to which it reflected the evolution in maintainability in reaction to the refactors, as perceived by the experts.

Three types of refactors were analysed: merges, decompositions, and hybrid refactors involving the extraction and subsequent merging of functionality from different services. The assessments correctly identified services that were candidates for merging, also in the hybrid refactors, based on the strong change coupling values between the candidates. As for the decompositions, in which we could measure the evolution of maintainability, we observed a high ability to reflect the evolution as perceived by the experts. In the hybrid refactors this ability varied.

We identified several factors that affected the accuracy of the assessment, such as systems with a small number of services, as the interpretation of the cohesion and size metrics is partly based on system-averages, which are less reliable in smaller systems. Another factor was that several systems were behind on maintenance, in the sense that code and interfaces which had become dead weight during the refactor still were not removed.

The contribution of this research is to provide a validated assessment method for microservices granularity, with a focus on maintainability. To learn more about the applicability of our assessment method and its validity in different contexts, we suggest conducting future research to validate it using larger data sets. This would provide a more robust validation of the method's accuracy, demonstrating its applicability beyond specific cases and making it less dependent on individual scenarios.

Keywords: microservice, MSA, granularity, coupling, cohesion, change coupling, architecture, assessment

Chapter 1

Introduction

In this chapter, we provide the motivation behind our research. We subsequently outline the problem that this study aims to address and formulate the research questions that guided our investigation. Additionally, we provide a high-level introduction to our approach, which is discussed in more detail in Chapter 3, and outline the structure of the present report.

1.1 Motivation

The popularity of microservices has increased tremendously over the past years. The term “microservice” was first used at a conference in 2011, where the participants used it to describe a new architectural style that was coming up [Lewis and Fowler, 2014]. Only 10 years later, in 2021, a survey showed that approximately 71% of IT professionals work with microservices in their enterprise [Statista, 2022]. This undeniable hype around microservices is caused by several large companies, such as Netflix, Spotify and Amazon adopting them [Krishna, 2021], and the wide range of benefits they are acclaimed to have over monoliths such as better scalability, language agnosticy and improved maintainability [IBM, 2021a, Ghofrani and Lübke, 2018].

Adopting microservice architecture does not come without some challenges however: practitioners describe how matters like the prediction of the performance of microservices in a production environment and guaranteeing data consistency within such a system can be quite a hurdle [IBM, 2021b]. Also, the aforementioned benefits are not guaranteed: in a survey among 60 practitioners conducted by [Bogner et al., 2019], only 2% of the participants reported zero symptoms of low maintainability in their system. Often mentioned were symptoms related to the evolvability of a system, like the implementation of new functionality being more time-consuming and a high number of defects with new releases.

A common denominator in these problems seems to be the granularity of the system; a coarse-grained system in which transactions are encapsulated in a single service does not impose any data consistency problems. A finer-grained system in which multiple services are involved in a single transaction however does require measures to be taken to guarantee some form of consistency. This shows how the complexity with regard to data management stands or falls by the granularity decisions made. The relevance of granularity is underlined even more by the consequences of a non-optimal granularity: if a certain granularity level enforces tightly-coupled services, this can significantly reduce the maintainability and scalability of the system [Hoday et al., 2020].

Microservice granularity is surrounded by a prominent research gap: there is a lack of tooling to identify service boundaries and evaluate the granularity of a system. Both the identification of service boundaries and the necessary fine-tuning are still mainly done based on experience, which is subjective and not always available in teams new to microservice architecture. There is a need for approaches which allow a quantitative assessment of granularity, to support software engineers in evaluating and reasoning about granularity more objectively. In this research, we will report on our approach and findings in proposing and validating a new assessment framework for the granularity level of an application with regard to maintainability.

1.2 Problem Statement

When designing a microservice architecture (MSA), either from scratch or by decomposing an existing application, it is crucial to carefully determine the boundaries for each microservice: sub-optimal microservice scopes can have undesired effects such as data consistency problems, an excessive global complexity or low reusability of individual components [Auslander, 2017a, Fritzsche et al., 2019, Homay et al., 2020, Shadija et al., 2017].

Currently, practitioners identify service boundaries primarily based on their experience and insight without making use of any frameworks or tools, except for some Domain-Driven Design (DDD) concepts. DDD is described often as an approach to determine microservice granularity, but fails to offer concrete decision support, as the guidance provided for determining service boundaries is not much more than an outline of how bounded contexts for each domain concept can be identified [Zimmermann, 2017]. This leaves a lot of room for subjectivity, which can result in an architecture with a sub-optimal granularity. Furthermore, the possession of experience in determining microservice boundaries is not a given in every project. The apparent lack of concrete decision support for reasoning about microservice granularity is a research gap that needs to be addressed. To allow any reasoning about granularity at all, it is essential to first establish an assessment framework which allows for quantitatively assessing MSAs.

An important consideration when discussing the assessment of granularity is that it is not possible to simply define an optimal granularity which can serve as a "goal granularity", as in this case, one man's meat is another man's poison; different requirements require different granularities. To give an example: while the local complexity of services might decrease at a finer granularity, it could impede the flexibility of the system in case it results in tightly coupled services. The most appropriate granularity level depends on the (non-functional) requirements of an application. It follows that, for a tool to be useful in making granularity decisions, the main condition is that it is not requirement-agnostic: it should assess granularity from the perspective of a specific requirement.

One of the system properties that is influenced by granularity to a great extent is maintainability. Intuitively, the maintainability of a system improves when adopting MSA (which, if proper design patterns are followed, is naturally more granular than other architectural styles), as it is now possible for team members to work on different services in parallel [Li et al., 2020]. On the other hand, a sub-optimal granularity which results in a high number of dependencies between the different services can result in a maintenance nightmare: the tight coupling can enforce change propagation, which requires an engineer to update multiple services as a consequence of modifying one service. Due to maintainability being

a common challenge for teams working with MSA, as pointed out by [Bogner et al., 2019], this research will investigate the assessment of granularity from a maintenance perspective.

The ultimate goal of this research is to reduce the need for experience in making microservice granularity choices, by proposing an approach which enables the quantitative assessment of the granularity of microservice-based applications with regard to maintainability.

1.3 Research Questions

Based on the aforementioned research objective, the following research questions have been formulated:

Main Question

How can the quality of the granularity of a microservice architecture be improved with regard to the application’s maintainability?

Sub-Questions

1. How can granularity be assessed from a maintainability perspective?
 - (a) Which metrics allow a quantitative assessment of maintainability in MSAs?
 - (b) How can these metrics be derived from existing projects?
2. How does this assessment method obtained from our results relate to the intuitive understanding of the experts?
 - (a) In what context can the performance of our assessment method be compared to the intuitive understanding of the experts?

1.4 Approach

In this thesis, we identified a set of maintainability metrics from which we constructed an assessment method for the granularity of MSAs. Subsequently, we collected microservice-based software projects in which refactors took place which affected the granularity of the microservices. Using the version control system in place, we retrieved the versions of the applications before and after a refactor was carried out, which enabled us to perform our maintainability assessment on the two different versions of the system (before and after refactoring). We analyzed how the metrics reflected these changes in granularity, and compared this with the intentions and experience of the engineers who have been working with the system.

Our research consists of two parts: in the first part, we performed a literature review during which we identified six maintainability metrics which are applicable to MSAs. Those metrics will be discussed in Chapter 2. The second part of this thesis focuses on the construction and testing of our assessment method, covering the necessary preparations, the data collection process, the results and the validation of the assessment method.

1.5 Report Structure

This report is structured as follows: Chapter 2 gives the necessary background on MSA and how its quality is related to granularity. In this chapter, we also explain why we will investigate granularity from a maintainability perspective and introduce the maintainability metrics we found in the literature which are suitable for MSAs. Chapter 3 provides a detailed explanation of our research strategy. Chapter 4 introduces the software projects we include as cases in our research and describes how we pre-processed the data to reduce noise and make the data compatible with the selected tooling. In Chapter 5 we present the results of our assessments, which we subsequently validate and discuss in Chapter 6. Finally, in Chapter 7, we discuss work closely related to our research, present our conclusions and give recommendations for future work.

Chapter 2

Microservices Granularity

In this chapter, microservice granularity is discussed, by giving some background on microservice architectures, discussing the definition of microservice granularity and finally discussing the influence of granularity on the quality of an MSA. Subsequently, the rationale behind our decision to approach granularity from a maintainability perspective in this research is explained.

2.1 Microservice Architecture

To fully understand the microservice architectural style and the problems it addresses, a recap on architectural styles for software design is provided. The conceptually simplest architectural style is monolithic architecture. The three parts that are common to enterprise applications, a client-side interface, a service-side application and a database, are all unified in a single code base in a monolithic architecture. The disadvantages of a system consisting of a single deployment unit are that the system always needs to be deployed as a whole and that maintaining the system simultaneously with multiple engineers is complex as modules are interdependent. Additionally, due to this interdependency, a single change requires testing of the entire system [Awati and Wigmore, 2022].

To deal with these problems, service-oriented architecture (SOA) has been introduced, long before microservice architecture. In SOA, the business logic of a system is broken down into smaller components. These components each define a range of capabilities, i.e., a service, through their APIs. In this context, a service can be seen as a collection of capabilities, which offer their functionality through an API. This allows an enterprise to reuse parts of a system in other applications within the enterprise [Erl, 2017]. Another advantage is that developers can work on different services in parallel, as the services are loosely coupled [RubyGarage, 2019].

Microservice architecture (MSA) is a form of SOA. MSA distinguishes itself by its scope: where most forms of SOA focus on exposing services on an enterprise-wide level, MSA is aimed at exposing services within a single application [IBM, 2020]. This extensive granularity is one of the main advantages of MSA; small units can be reused in other applications as they are less application-dependent, and the scalability of microservice-based systems is outstanding compared to other architectural patterns, as each microservice can be scaled independently based on its workload [Harsh, 2022]. There is no widely-accepted definition of a microservice, but there are some characteristics that are commonly associated with microservices [Lewis and Fowler, 2014]:

- Cohesive blocks: the functionality encapsulated in a microservice should belong together, in accordance with the single-responsibility principle: a module should only have one reason to change [Richardson, 2020]. Adhering to this principle enforces cohesive services with a set of strongly related functions.
- Focused on business capability: MSA advocates for organizing microservices and their operating teams around business capabilities instead of technical layers as a UI team, a database team and a team handling the server side of the system. This results in cross-functional teams which can release new features independently. This newly gained independence allows for decentralized governance; teams bear responsibility for their own microservice.
- Loose coupling: where in SOA the integration of services, i.e. the application of business rules and the routing of messages, is often handled by an integration layer (e.g., an Enterprise Service Bus), in MSA it is considered good practice to encapsulate such complexity in the services themselves. This self-containment reduces coupling between the microservices.
- Decentralized data management: the database-per-service pattern is often adhered to in MSA, as connecting each microservice to the same central database would lead to a tight coupling between the microservices.
- Automated integration, delivery and deployment: in MSA, automation in the form of continuous integration (CI), continuous delivery and continuous deployment (CD) is embraced. Such infrastructure automation prevents teams from carrying the burden of a high number of microservices to manage (manually test and deploy).
- Graceful degradation: due to the distributed nature of MSA and the encouragement to keep components loosely coupled, the failure of one component in a microservice-based application should not have much impact on the other microservices.

2.2 Granularity in MSA

2.2.1 (Lack of) Definition

Whereas the relevance of microservice granularity is widely acknowledged, the definition of microservice granularity is not; there are several definitions to be found in literature, but one that is commonly accepted is still lacking [Vera-Rivera et al., 2021]. The most straightforward and, inherently, the most intuitive definition of the granularity of a microservice would be the size of a service, which is a definition described in several papers [Hassan et al., 2020, Kulkarni and Dwivedi, 2008, Vera-Rivera et al., 2021, Ulander, 2017, Cojocararu et al., 2019]. Their interpretation of size in this context is quite varying however: a range of different metrics is used to define granularity. [Cojocararu et al., 2019] interviewed an expert about metrics currently used in the industry to measure granularity, and he mentioned lines of code (LOC) as a metric still used by many practitioners. A drawback of using LOC to represent the granularity of a system is pointed out by [Bogner et al., 2017b]: due to the technical heterogeneity which MSA allows, the LOC of a service becomes an inaccurate representation of the granularity of a service. The different levels of verbosity of the programming languages used in a microservice-based application distort the indication LOC can give of granularity: a service A, implemented in a verbose language and

only supporting one operation, might falsely appear coarser-grained than a service B containing thrice the number of operations of service A. That this “functional scope” of a service is part of the definition of granularity is supported by several papers investigating granularity [Hassan et al., 2020, Kulkarni and Dwivedi, 2008, Vera-Rivera et al., 2021, Al-Debagy and Martinek, 2021, Glöckner et al., 2016, Ulander, 2017, Jain et al., 2021] and is another interpretation of size. The functional scope of a service can be expressed by the number of operations exposed by the service in its interface [Vera-Rivera et al., 2021, Ulander, 2017, Wang, 2009, Cojocaru et al., 2019]. [Cojocaru et al., 2019] adds here that, in order to draw meaningful conclusions about granularity based on the number of operations per interface, one should only consider relative values, by comparing them with other services’ size values.

Just as important as size in defining granularity, is the number of services that are part of the whole application [Vera-Rivera et al., 2021, Baresi et al., 2017]. [Baresi et al., 2017] defines granularity as a trade-off between these two. [Sellami et al., 2022] also takes the number of services into account in their definition, which is given in the context of a decomposition, by the following formula:

$$Granularity(M) = \frac{|C|}{|M|} \tag{2.1}$$

The granularity of a decomposition M is defined here as the ratio between the original system size, based on the number of classes C , and the total number of microservices in the decomposition. [Khoshnevis, 2023] proposes a similar definition, which is especially useful at design time. In their definition, the number of classes in (2.1) is replaced by the number of activities which should be implemented by the future application. This definition is given as part of the explanation of their algorithm for identifying microservice candidates, so with activities, they refer to the activities defined in the configurable business process model which serves as input to this algorithm. A last definition for the granularity of a service is the complexity of a service [Vera-Rivera et al., 2021, Newman, 2021]. While specific metrics are not explicitly mentioned, the authors of [Vera-Rivera et al., 2021] emphasize that factors such as complexity and dependencies should be given less significance compared to the size and the number of services when determining granularity.

2.2.2 Influence on microservice quality

The relevance of determining an appropriate granularity for an MSA lies within its influence on the quality of a microservice-based system. A granularity level can maximize the quality of an MSA with regard to certain system properties, but potentially also hinder the system quality. Assessing the appropriateness of a granularity level is not a straightforward task: a range of trade-offs should be taken into account as granularity influences the quality of an MSA in several ways [Ulander, 2017]. One of these trade-offs is related to complexity: in a fine-grained system in which the services are relatively small, the complexity per service might be lower than in a coarser-grained system, but as each service only contains a small portion of functionality, the number of dependencies in the system rises. This increase in dependencies can result in a system with high coupling, which negatively influences the overall system complexity and the understandability of the system [Ulander, 2017]. Such trade-offs also exist for other system properties, like performance and maintainability, and complicate the development of evaluation frameworks for microservice granularity

[Hoday et al., 2019, Hoday et al., 2020, Vera-Rivera et al., 2021, Li et al., 2020]. In this section, we discuss the relationship between granularity and several system quality attributes of MSAs.

Performance and Reliability

Several papers describe the relationship between system performance and granularity. From a performance perspective, optimal service granularity is a delicate balance between isolating the services consuming a lot of resources or time and minimizing communication overhead due to extra service calls [Zórnio, 2020]. This overhead can be in terms of, e.g., bandwidth consumption or data propagation.

Splitting the modules which heavily make use of resources into separate services can increase the scalability of a system and with it the system’s performance, but this will result in more communication between services, which in its turn can negatively affect performance in terms of response time [Vera-Rivera et al., 2021, Auslander, 2017b, Brown, 2020a]. Both scalability and performance were the most-mentioned reasons for migrating to microservices in the systematic literature review conducted by [Vera-Rivera et al., 2021] and are also considered as important quality attributes for microservice granularity.

Another attribute affecting the performance is the so-called chattiness of a system. This refers to the amount of service calls that the system makes, which is largely influenced by granularity [Hoday et al., 2020, Vera-Rivera et al., 2021, Hoday et al., 2019, Wang et al., 2021, Zórnio, 2020, Auslander, 2017b]. This influence can be explained by the fact that a process distributed over multiple services will require more inter-service calls than a process only involving one service. The accumulated latency of these inter-service calls negatively impacts the response time and inherently the perceived performance of the system [Shadija et al., 2017].

[Wang et al., 2021] notes how the communication between services can be minimized by grouping functions with the same dependencies into one service. These groups form sensible initial microservice candidates, as grouping based on dependencies not only reduces the communication overhead but also the complexity of the system [Zórnio, 2020]; if the services work in a choreography and are split without considering dependencies, this will result in a spaghetti of service calls which can be hard to oversee. Furthermore, a higher number of network calls compromises the overall reliability of the system [Zórnio, 2020]. [Hoday et al., 2019] describes how there is a trade-off here, between a finer-grained system resulting in more message-passing which can be time-consuming, and a coarser-grained system which is more process-demanding, and subsequently has a higher power consumption. The computing time can also be affected negatively by a lower granularity, as a coarser-grained service has increased computational complexity and has to process larger data units [Hoday et al., 2019]. Communication time and computing time have to be balanced for an optimal granularity level.

Complexity

Several factors contribute to a system’s complexity, which are each influenced by its granularity. That a microservice-based system is per definition granular to some extent already implies the system has a higher global complexity compared to a monolith

[Waseem et al., 2021]. [Li et al., 2020] explains that a more distributed development leads to a higher global complexity for the development teams to deal with. Software professionals interviewed by [Fritzsich et al., 2019] confirm this, by stating how fine-grained systems can enforce a complex macro-architecture (the relations between microservices). Such extra complexity increases the technical costs, another driver to take into account when deciding on granularity [Li et al., 2020].

[Fritzsich et al., 2019] describes a tendency among its participants to choose a more coarse-grained architecture, in order to avoid the extra global complexity caused by a more elaborate infrastructure enforced by a finer-grained system. [Laskowski, 2019] encourages this defaulting to coarser-grained services, but notes that one must search for the optimal balance between extra (global) complexity and the advantages associated with microservices, such as agility, scalability and resilience. [Seroukhov, 2020] points out that a lower granularity is not only in favour of complexity; although the interaction between the services may be simpler, the services will be larger and subsequently have a higher local complexity. This statement is supported by [Hoday et al., 2019] and [Hoday et al., 2020], who state that local complexity increases in coarser-grained systems.

A finer granularity often leads to higher reusability of the microservices [Hoday et al., 2020, Li et al., 2020, Shadija et al., 2017]. This is because the services become more generic and less complex, which makes it easier to reuse them in another system that requires that specific functionality.

Data Management

In a database, multiple tables can be updated in a single transaction. If a team adopts the database-per-service pattern, this can result in the distribution of transactions across multiple databases if multiple microservices are involved in a single transaction. [Newman, 2021] illustrates these distributed transactions and their consequences using a simple ordering system as an example (see Figure 2.1). In Figure 2.1A, the system communicates with one monolithic database, that contains two different tables, one for orders and one containing entries with what needs to be picked by the warehouse team (picking table). In Figure 2.1B, each table resides in a separate database. [Newman, 2021] explains how the data monolith in Figure 2.1A provides transactional safety; if one of the tables cannot be updated for any reason, the entire transaction involving both tables will not be executed. This transactional safety is lost in Figure 2.1B since the ordering process now spans transaction boundaries. The consequence is that in case an order record is inserted successfully, but the insertion of the picking record fails, this could bring the system into an inconsistent state.

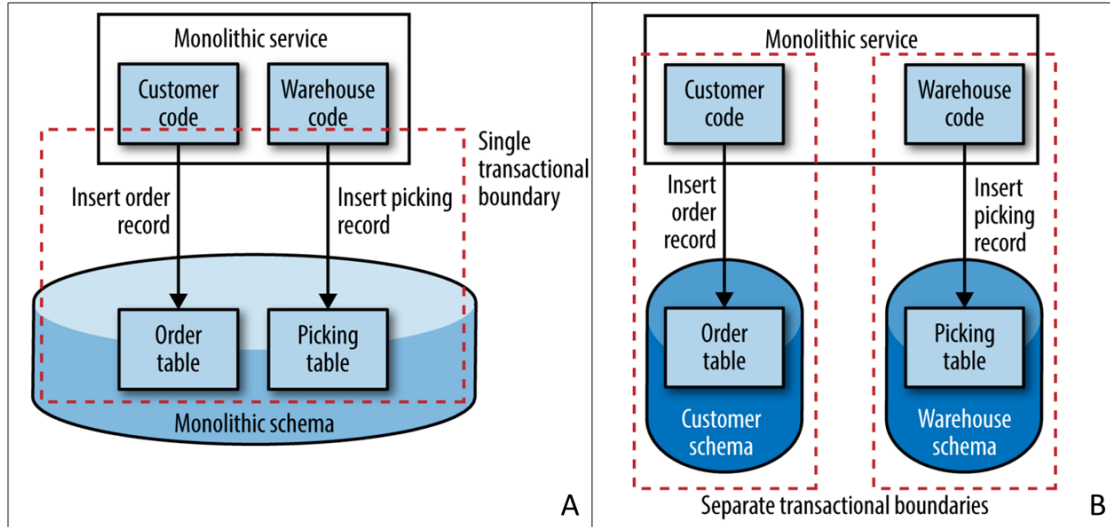


FIGURE 2.1: Example of an ordering system [Newman, 2021]. A shows a system with a monolithic database, whereas B shows a system updating multiple databases.

Solutions to roll back the entire transaction in such a case do exist, but these solutions substantially increase the global complexity of the system.

2.3 Maintainability Perspective

Maintainability is another system attribute which is, according to literature, strongly influenced by granularity. Obtaining an optimal granularity is challenging because of the many system properties that are influenced by the level of granularity, each in a different way. There are many trade-offs to take into account, which complicates the evaluation of a system's granularity: one cannot simply rate the general quality of a granularity level, since this can only be done with regards to a specific (set of) requirement(s). As maintainability is an important system property, which can be critical for a project to succeed or not, this research aims to assess microservices granularity mainly from a maintainability perspective. In this section maintainability as a software characteristic is discussed, covering its definition and its relation to granularity.

2.3.1 Definition

ISO 25000, a series of standards which define a wide range of terms used in the context of software product quality, defines maintainability as "a characteristic representing the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in the environment, and in requirements" [ISO25000, nd]. Maintainability forms an umbrella for a set of other system attributes, which are defined in the ISO series:

- Modularity: the degree to which an application is composed of discrete components such that a change to one component has minimal impact on other components.

- **Reusability:** degree to which an asset can be used in more than one system, or in building other assets.
- **Analysability:** degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
- **Modifiability:** the degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
- **Testability:** degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met

The relevance of aiming for high maintainability in any software project is emphasized by research that shows how only 10 percent of the lifecycle of a software product is spent on the development of the product, while the other 90 percent is spent on maintaining activities, in terms of time and money [Engelbertink and Vogt, 2014].

2.3.2 Maintainability in MSA

Microservices are acclaimed to have higher maintainability than traditional monoliths, which makes maintainability a popular driver for adopting microservices [Ghofrani and Lübke, 2018, Knoche and Hasselbring, 2019, Cojocar et al., 2019]. This improvement in maintainability is explainable as in essence, a microservice architecture is beneficial regarding maintainability compared to a monolithic architecture because having all functions in a single project obstructs team members from doing maintenance in parallel [Li et al., 2020]. Furthermore, the strong separation of components in MSA prevents a spaghetti of dependencies from developing [Knoche and Hasselbring, 2019, Newman, 2021]. MSA is also known for advocating a relatively small service size compared to other architectural styles like traditional SOA, which improves the maintainability of the services even further [Mella et al., 2019]. These maintainability advantages do not come for free, however, when adopting microservices. Granularity plays a large and complicated role in this; for a system to have a high modifiability for instance, it is not obvious which granularity level supports this the best. If a system has a low granularity, which can result in a tightly coupled structure with complex services, modifiability is affected negatively [Homay et al., 2020, Auslander, 2017a]. Yet if multiple services have to be changed in order to implement a new business feature, which reduces modifiability, the system might be too fine-grained [Brown, 2020b].

A high modifiability is essential in case of a high frequency of change. Some of the professionals interviewed by [Wang et al., 2021] indicate that they see the frequency of change as the most important splitting criterion. They explain how a high granularity can create development overhead: “If you want to release, for example, a security patch for Java, [...] you have to re-release each of your services. The more you split up, the more overhead you have.” [Shadija et al., 2017] points out that in case of a high frequency of change, it is common practice to let the functionality itself determine the size of the microservices based on its change frequency, and thereupon the granularity.

The degree to which an application facilitates change is also dependent on its testability, another sub-characteristic of maintainability. A coarser-grained service is larger and

more complex than a fine-grained one, and [Brown, 2020b] explains that the testing effort required in response to a modification will be larger than in the case of a finer-grained service. However, [Newman, 2021] argues that the overall testing effort will be smaller in a coarser-grained system as having bigger chunks simplifies the interactions between the services; in order to test the consumption of service S, each consuming service has to be stubbed into the context of S. It would be a smaller effort to only stub a coarser-grained API than having to stub multiple finer-grained ones.

Maintainability can be improved by having proper monitoring and diagnostics in place. [Homy et al., 2020] stresses how those properties are related to granularity in a control system context: if the whole machine is controlled by one service, and this service fails, the whole machine will be marked as erroneous. By letting multiple services each control a part of the machine, the failure of a service will only lead to the failure of the corresponding machine part, which enables more accurate diagnosing.

The aforementioned dualities seem to be inherent to the relation of maintainability and microservice granularity, which complicates reasoning about granularity from a maintainability perspective. Concrete guidance for teams who want to optimize their application’s granularity in order to improve its maintainability would be valuable here.

2.4 Metrics

This research focuses on evaluating microservice granularity primarily from a maintainability perspective, as maintainability plays a crucial role in the success of a project. In this section, we discuss a variety of metrics and heuristics that assess maintainability in the context of microservices architecture (MSA). These metrics and heuristics formed the basis of our assessment method.

As maintainability entails a variety of sub-characteristics, quantifying it is not a trivial task. Over the years, maintainability metrics have been proposed for increasingly higher levels of abstraction. As pointed out by [Bogner et al., 2017a], the earliest metrics which allowed assessing the maintainability of a system had quite a low-level scope and were mainly relevant for software written in a procedural style. Examples are McCabe’s Cyclomatic Complexity [McCabe, 1976] and Halstead’s metrics [Halstead, 1977], which are based on the number of independent decision paths to take in a program, and on the program length and vocabulary, respectively. In reaction to the rise in popularity of the object-oriented programming paradigm (OOP), other metrics became popular. These metrics were partially inherited from the traditional ones, but also novel metrics, such as the lack of cohesion in methods and the coupling between object classes, were introduced. These metrics were still mainly relevant for analysis at a class level, and a need for module-level maintainability metrics arose. This need was answered by [Lindvall et al., 2003] among others, who introduced the coupling between modules metric. Service-oriented systems require a set of tailored metrics, as SOA brings in a new level of abstraction by considering services instead of modules. According to the results of the literature review performed by [Bogner et al., 2017a], most maintainability metrics can be accommodated under either size, coupling or cohesion. These metrics are discussed in the sequel.

2.4.1 Size metrics

A well-known, but controversial software metric, related to maintainability, is size. [Bogner et al., 2017a]. Although size metrics need to be complemented by other maintainability metrics since they are not sufficiently accurate themselves, they can be of use in a relative sense: they can provide insight into how large a component is with respect to the rest of the system, which is valuable as in the end, a larger size implies lower maintainability [Bogner et al., 2017a]. Several metrics can be used to measure the size of a microservice. We will discuss two prominent ones, which will also be included in our metric suite: lines of code and the weighted service interface count.

Lines of Code

The controversy surrounding size as a measure for maintainability is partly caused by the most frequently used metric to assess size: lines of code (LOC) [Siket et al., 2014, Bogner et al., 2017a]. At first glance, LOC appears to be a simple metric, which can be measured in a straightforward way, namely by counting lines of code. No consensus is yet reached however on how comments and blank lines should be dealt with, as [Siket et al., 2014] points out. During this research, we adhered to the definition of LOC as the number of lines of code from which blank lines and comments are excluded.

A motivation for including LOC as a metric is its relation to complexity. Complexity is not directly represented by the selected metrics selected, as it is challenging to find a complexity metric suitable for microservices: due to the freedom in choosing programming languages MSA is known for, which often results in heterogeneous systems, common complexity metrics such as cyclomatic complexity are hard to measure as the required tooling is language-dependent. There is, however, a strong correlation between LOC and cyclomatic complexity [Heitlager et al., 2007]. LOC itself might be a simple metric to measure, but it is also language dependent. Therefore, when it comes to the interpretation of the metric, the programming language of a service should be taken into account, since the level of verbosity differs per language, as illustrated by the code fragments in Figure 2.2 [Bogner et al., 2017a]. In both fragments, an array is created and subsequently printed to the console, but the fragment written in Java (lower) requires eight lines whereas the Python fragment (upper) represents the same functionality in three lines. This advocates for a weighting factor to be able to compare the LOC of services written in different languages.

```
stuff = ["Hello ,_World!"]
for i in stuff:
    print(i)
```

```
public class Test {
    public static void main(String args []) {
        String array [] = {"Hello ,_World"};
        for (String i : array) {
            System.out.println(i);
        }
    }
}
```

FIGURE 2.2: Verbosity of different programming languages [Goebelbecker, 2022]. The fragments show the implementation of Hello World in Python (above) and Java (below).

[Bogner et al., 2017a] state that an absolute service size is not of much help to reason about maintainability, as acceptable value ranges for this metric are lacking. By comparing it to the average LOC of all services in an application, however, it can help to discover which services are relatively large and might be candidates for refactoring. In case different languages are used in the system, one should harmonise the measures using weight factors or only directly compare measures of similar languages.

Weighted Service Interface Count

As LOC is sometimes considered a sub-optimal metric to assess an MSA implemented with multiple languages, a size metric which is programming language-independent would be useful to include in our metric suite, to complement LOC. [Hirzalla et al., 2009] proposes a size metric which is based on a service's interface, instead of its source code, namely the weighted number of exposed interfaces (WSIC), which represents the number of interfaces or operations per service. The underlying rationale is that a higher number of interfaces implies a more complex service, and a higher complexity for the system as a whole since more interfaces directly require higher constructing and testing effort. Additionally, more monitoring is required with every extra interface and on a service level a higher number of interfaces can complicate problem determination [Hirzalla et al., 2009]. The metric could be weighted in different ways to account for the number and complexity of the parameters of each operation. As to our knowledge no validated weighting methods exist yet, we used the default weight of 1. The interpretation of WSIC values during this research was based on the thresholds proposed by [Bogner et al., 2020], who used a benchmark-based approach to perform a threshold derivation for a set of maintainability metrics. They labeled the different quartiles of the observed quartile distribution from best to worst. For the WSIC this resulted in the intervals shown in Table 2.1. Important to highlight is that Bogner et al. decided to exclude APIs with fewer than 5 operations from their benchmark analysis. The rationale behind this exclusion was the abundance of such small interfaces, which could potentially bias the thresholds towards favouring extremely small APIs.

TABLE 2.1: WSIC thresholds proposed by [Bogner et al., 2020]

Quartile	Top 25%	25-50%	50%-75%	Worst 25%
WSIC	[5,8]	[8,15]	[15,31]	[31,1126]

Interplay of LOC and WSIC

Although LOC is a controversial metric, especially when applied to a polyglot MSA, it is a useful complement to WSIC. WSIC might be a much more service-oriented metric than LOC, as stated by [Bogner et al., 2017a], but it has not been empirically validated yet and is more focused on complexity than size [Munialo et al., 2019]. By including both metrics in our metric suite we aim to obtain a reliable representation of the size property of maintainability.

2.4.2 Coupling metrics

The two coupling metrics which we included in our metric suite are structural coupling and change coupling, also known as logical coupling. These metrics are often defined in the literature and are applicable to MSAs.

Structural coupling

According to traditional software design principles, the coupling between two modules should be as low as possible, whereas the relations between the artefacts within the module should be strong [Yourdon and Constantine, 1979]. Coupling has a strong effect on maintainability, as changes or bugs in a strongly coupled module are likely to propagate to other modules, lowering maintainability [Panichella et al., 2021]. Two software modules are structurally coupled if there are any code or structural dependencies between them. In the context of MSA, such a structural dependency can be in many forms, e.g., in the form of service calls or a producer-consumer relation. [Panichella et al., 2021] proposes a definition of structural coupling specifically for microservices, which is given in the following formula:

$$StructuralCoupling(s1, s2) = 1 - \frac{1}{degree(s1, s2)} * LWF * GWF \quad (2.2)$$

This definition is based on the local weighting factor (LWF) and the global weighting factor (GWF), defined by formulas (2.3) and (2.4), respectively.

$$LocalWeightingFactor(s1, s2) = \frac{1 + outdegree(s1, s2)}{1 + degree(s1, s2)} \quad (2.3)$$

$$GlobalWeightingFactor(s1, s2) = \frac{degree(s1, s2)}{max(degree(allservices))} \quad (2.4)$$

The LWF considers the degree and the out-degree from s1 to s2, where the degree represents the total number of structural dependencies between s1 and s2, and the out-degree(s1,s2) the number of dependencies among the total degree which is directed from s1 to s2. The GWF weighs the degree between two services with the highest degree between a service pair existing in the application, considering all combinations of services in the application as possible pairs.

By basing the definition of structural coupling on the LWF and GWF, the metric also takes into account the general dependencies which are distributed to other services. Due to the normalization which is done in the formula, the structural coupling between two services will always be a value between 0 and 1 [Panichella et al., 2021]. A value close to 1 indicates a high structural coupling. As this metric has been validated on 17 open-source projects, we will adhere to the aforementioned definition of structural coupling and add it to our metric suite as-is [Panichella et al., 2021].

Change coupling

As an alternative to the traditional code-based metrics, some researchers have started investigating the use of metrics derived from developer activity to assess maintainability [Oliva and Gerosa, 2015, Zimmermann et al., 2005]. Already in 1997, [Ball et al., 1997] described how version control systems contain a lot of data, which, if leveraged wisely, can provide insights into the evolution of a software system regarding its properties. Version control data can also be of help in monitoring and assessing maintainability. Below we discuss one of the metrics used in this research which is mined from version control data, namely change coupling.

The change coupling between two software artefacts, also known as logical coupling, is defined as “the implicit and evolutionary dependency of two software artefacts that have been observed to frequently change together during the evolution of a software system” [Fluri et al., 2005, Oliva and Gerosa, 2015]. "Changing together" can be defined in many alternative ways, and the most appropriate definition depends on the purpose of the analysis. However, an intuitive definition is to let a commit define a logical change set and to consider artefacts that change simultaneously in a large number of commits to be change-coupled. Figure 2.3 shows an example of artefacts A and B that co-change through time, while artefact C seems to evolve independently.

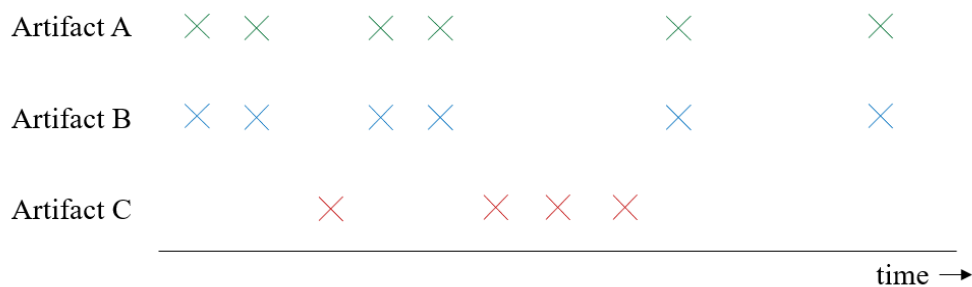


FIGURE 2.3: Evolution of three software artefacts over time. Each cross represents a commit in which the corresponding artefact is modified.

An important property of this metric is that it can uncover relations between software artefacts that are not explicitly present in the code of a system. This makes the metric appealing to apply to MSAs, as it is able to reveal hidden dependencies regardless of hindrances such as REST calls and event buses, which obstruct code-based analyses from discovering these relations [Tornhill, 2015]. Another advantage is that the metric can be derived solely from the version control history, which makes it programming language-agnostic [Oliva and Gerosa, 2015].

The validity of change coupling as a maintainability metric is supported by [D’Ambros et al., 2009] and [Cataldo et al., 2009], which respectively showed how change couplings correlate with defects, and how the relationship between change coupling and fault proneness was of stronger relevance than the effect of structural coupling in two different industrial software projects [Oliva and Gerosa, 2015]. Additionally, [Hassan and Holt, 2004, Hassan and Holt, 2006] compared the effectiveness of structural coupling and change coupling in predicting the propagation of changes, and concluded that change coupling is significantly more relevant.

The application of change coupling analysis in MSA is far from a mature research area. To our knowledge, [Tornhill, 2015] is the only (grey literature) work where the application of change coupling to MSAs is explicitly mentioned, although the concept is not far-fetched: change coupling analysis is often applied to assess the modularity of an architecture, where a module is a generic concept that can correspond to a microservice [Silva et al., 2014, Zimmermann et al., 2005, Oliva and Gerosa, 2015].

Raw counting

Based on the raw counting approach for change coupling described by [Oliva and Gerosa, 2015] and used by [Gall et al., 2003, ?, Oliva and Gerosa, 2011], we propose our own identification approach which is tailored towards MSAs.

The concept of raw counting can be best explained by visualizing a co-change matrix (Figure 2.4). This is a symmetric matrix, which should be built based on the version control data. The value in each cell $[i,j]$ (with $i \neq j$) represents the frequency with which artefact i and j were part of the same change set during a period X . Cell value $[i,i]$ represents the number of changes artefact i was subjected to in that same period X .

[Oliva and Gerosa, 2015] distinguishes two types of change coupling relationships, namely non-directed relationships and directed relationships. To infer the non-directed change coupling relationship between artefact A and artefact B , they simply count the number of shared revisions (3 in the example of Figure 2.4). Every cell $[i \neq j]$ in Figure 2.4 containing a non-zero value represents a change coupling between artefacts i and j , and the value represents the strength of the change coupling. This strength, equal to the number of shared revisions of two artefacts, is also referred to as the support value, a term borrowed from the data mining field [Oliva and Gerosa, 2015]. In a directed relationship, a coupling strength is calculated from one artefact to another. To calculate the change coupling strength from artefact A to artefact B formula (2.5) can be used:

$$CC_directed(A, B) = \frac{shared_revisions(A, B)}{revisions(A)} \quad (2.5)$$

In the matrix in Figure 2.4, this is equivalent to $cell[1,2]/cell[1,1] = 3/4 = 0,75$, while the

	Artifact A	Artifact B	Artifact C	...
Artifact A	4	3	0	...
Artifact B	3	9	1	...
Artifact C	0	1	7	...
...

FIGURE 2.4: Example of a co-change matrix [Oliva and Gerosa, 2015]

change coupling from B to A is weaker (0,33). Directed relationships provide insight into whether two artefacts are evenly dependent on each other, or whether one artefact is more dependent on the other than vice-versa. This measure, in which the shared number of revisions is normalized by dividing it by the absolute number of revisions, is also known as confidence [Oliva and Gerosa, 2015].

To keep our change coupling analysis as granular as possible, increasing its usefulness in pinpointing modules which are candidates for refactoring, we inferred directed relationships from the raw countings.

Identifying which change couplings indicate a merge is a research field in itself. The support value, which refers to the number of shared revisions between two artefacts, can assist in differentiating between artificial change couplings and meaningful ones. It is important to establish an appropriate threshold for the support value, taking into account the specific context of the study. Naturally, the relevant intervals for the support value would differ between a large industrial project and a small personal project. As suggested by [Oliva and Gerosa, 2015], a statistical analysis of the distribution of support values can be performed to determine significant change couplings. One approach is to conduct a quartile analysis, where it is assumed that change couplings requiring a refactor have a support value above the third quartile (Q3). In other words, these are the pairs belonging to the top 25% of pairs with the highest support values. Extra attention should be paid to outliers on the higher end, which is defined as support values exceeding the sum of Q3 and 1.5 times the interquartile range (IQR), which represents the difference between the median of the lower half of the data and the median of the upper half of the data [Interquartile range, nd].

A high support value is not per definition a reason for refactoring a service pair: if the coupling between artefacts A and B has a support value which is a high outlier in the distribution, 100 for instance, but artefacts A and B are each revised independently another 900 times, this should not form a direct indication for merging A and B.

An explanation for the high support value could be that A and B were both subject to a lot of changes in comparison to the rest of the system, and because they both have a higher change frequency the support analysis incorrectly prioritises them as refactor candidates. This suggests the need for incorporating a confidence threshold, as suggested by

[Oliva and Gerosa, 2011], alongside the support threshold. This confidence value, which provides an indication of the level of coupling between two artefacts by calculating the ratio of the total number of revisions to the shared number of revisions, is what we refer to by the term "change coupling" in this report. [Oliva and Gerosa, 2011] proposes the following change coupling intervals to classify coupling strengths:

- [0.00, 0.33]: low coupling
- [0.33, 0.66]: regular coupling
- [0.66, 1.00]: strong coupling

Logical change set

Necessary in order to derive the change coupling between microservices is the definition of a logical change set. In most literature available on change coupling, the analysis is done on a file level, which makes a commit a sensible and straightforward scope for a logical change set. In MSA however, to emphasize the clear separation of responsibilities, projects often are stored in multiple repositories in which each repository contains the code of one microservice [Mathews, 2021], and has its own version control history. Commonly used version control systems like Git [Beckman et al., 2021] do not permit commits crossing repository boundaries. This obstructs the analysis of change coupling in a multi-repository system based on individual commits, as logical change sets will not offer insights into the change coupling between different microservices. As recommended by [Oliva and Gerosa, 2015] and [Tornhill, 2015], an alternative is to consider a set of commits connected to the same task id as a change set. Such a task id represents a task in the issue tracker of an application and provides information on which commits are related to each other and on the context of the commit(s). This context is useful when applying change coupling as a maintainability metric, as it allows one to categorise the commits based on their context, and thus to extract the commits which were specifically done for maintainability purposes.

If issue tracking was not actively done during the lifecycle of a microservice-based project, another alternative is to group commits in the same temporal window [Tornhill, 2015]. The rationale behind this is that in case each time a microservice A is updated, there is also a commit to another service B on the same day, this hints at the existence of relevant change coupling between service A and B. This temporal window does not have to be a day but can be adjusted to the commit frequency in the project under study. However, a correct implementation of an algorithm to group commits which are within the same temporal window will have to use a sliding window (Figure 2.5).

Service A has only been revised once, as indicated by the single green cross in Figure 2.5. However, grouping all commits within a temporal window of two days in the scenario pictured in Figure 2.5 classifies service A as part of two logical change sets, namely set 2, in which it is coupled to a revision to service B, and set 3, in which it is coupled to another revision to service B. Therefore, when considering the output of a change coupling analysis in which commits were grouped based on a temporal window, one should be aware that the number of shared revisions can be higher than the absolute number of revisions to a service.

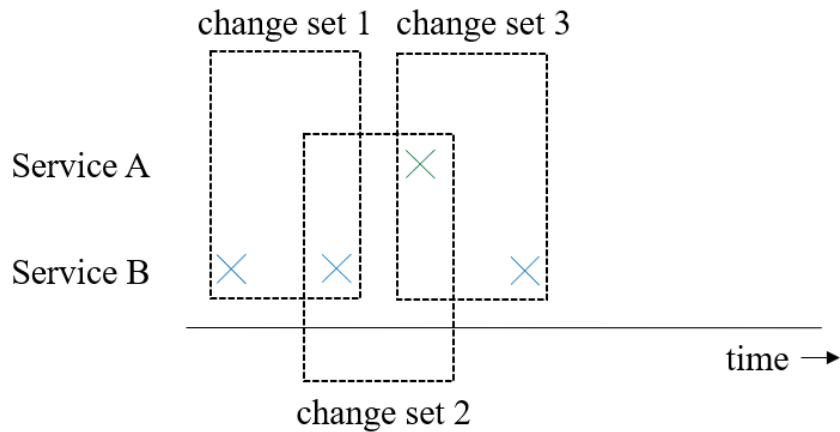


FIGURE 2.5: Sliding window implementation which increases the number of change sets. In this example, every revision (represented by a cross) took place on another day, and they were made on four consecutive days.

Interplay of change coupling and structural coupling

In some applications, change coupling may perform better as a maintainability metric than structural coupling [Oliva and Gerosa, 2015]. Especially in predicting the propagation of changes, as shown by [Hassan and Holt, 2004, Hassan and Holt, 2006], change coupling seems to excel. However, a significant drawback of using change coupling as a standalone maintainability metric is that it requires historical data. In new green field projects, measuring structural coupling is the only way to gain insight into to which extent components are coupled [Oliva and Gerosa, 2015]. As in smaller hobby projects, the availability of version control history can also be limited, we included both structural coupling and change coupling in our metric suite to ensure we are able to still assess coupling to some extent in such projects.

2.4.3 Cohesion metrics

The cohesion metrics that have been identified from the literature are change frequency and service interface data cohesion. These metrics can both be tailored to MSAs.

Change frequency

Change frequency (CF) is another metric based on developer activity. In the context of MSA, this metric corresponds to the number of times a service is modified (i.e., the number of commits) per time unit.

A high change frequency with respect to other services in the system is not a direct indication for decomposing but in combination with size metrics, CF can be of help in pinpointing low-cohesive services, which are candidates for refactoring [Tornhill, 2015]. The reasoning here is that large services with a relatively high change frequency could be covering multiple bounded contexts, and an engineer should make an informed decision on whether to

split these services. Also when only a single bounded context is covered by a service, a high change frequency could be a reason for refactoring as it might be desirable to extract the portion which is frequently updated to a separate service which can be labelled as of higher risk and be governed by a small dedicated team [Ferguson, 2017]. To allow for comparisons between different CFs, we consistently calculate CF by dividing the absolute number of changes by the number of months the changes were accumulated over.

Service Interface Data Cohesion

Service interface data cohesion (SIDC) is another notion of cohesion, which is a more service-specific metric as it is measured on an interface level. This is one of the few cohesion metrics from the extensive literature review of [Bogner et al., 2017a] that is automatically derivable and suitable for MSAs. The limited availability of such metrics is due to the semantic nature of cohesion, which complicates the collection of its metrics [Bogner et al., 2017a]. SIDC considers the equality between the parameter types of operations in an interface [Athanasopoulos et al., 2015] so that if all operations defined in an interface use a common parameter data type, this indicates that the corresponding service is highly cohesive [Perepletchikov and Ryan, 2011, Bogner et al., 2017a]. Formula (2.6) defines SIDC, which is equal to the sum of operations with identical data types divided by the total number of distinct operations, normalising the metric to values between 0 and 1 [Bogner et al., 2017a, Athanasopoulos et al., 2015]:

$$SIDC(S) = \frac{OC(IS)}{OD(IS)} \tag{2.6}$$

According to this formula, the SIDC of a service S corresponds to the number of operations of its interface IS which have at least one parameter data type in common (OC), divided by the total number of discrete operations defined in IS (OD). As for the evaluation of SIDC values, [Bogner et al., 2020] provided us with thresholds. These thresholds were calculated using a benchmark-based approach, and the resulting SIDC intervals can be seen in Table 2.2.

TABLE 2.2: SIDC thresholds proposed by [Bogner et al., 2020]

Quartile	Top 25%	25-50%	50%-75%	Worst 25%
SIDC	[1.00, 1.00]	[1.00, 0.64]	[0.64, 0.55]	[0.55, 0.00]

Chapter 3

Research Strategy

This chapter discusses the strategy we put into effect during this research. We first give a high-level overview of our research plan, after which the different steps are discussed in more detail.

3.1 Overview

Our strategy consisted of five steps, as depicted in Figure 3.1. Based on the maintainability metrics identified in Section 2.4, we first formulated two sets of requirements, one for the software projects we use as cases and one for the instrumentation we selected. Subsequently, we performed our data collection. During this phase, initially, the cases were selected. The compatibility of the tools with the selected cases is not always a given: for some cases, it was not possible to derive certain metrics while for others some case-specific adjustments to the instrumentation had to be made. The preparation of the data, in terms of extracting and cleaning the data, was an essential step in our research and is discussed extensively in Chapter 4. During this phase, the input data required by the tools were extracted from the cases, and any necessary adjustments to the tooling were made. When all input data required by our assessment was available, this ushered in the data analysis phase. In this phase, we analysed the different metrics derived during our assessment and in Chapter 5 an overview of these calculated metrics is presented per refactor type. To validate whether the findings of our assessment method corresponded with reality, per case an expert who worked on the project and was involved during the refactor(s) was interviewed. During these semi-structured interviews, the experts were asked about the team's intentions for a refactor, i.e., which system property they tried to improve by that refactoring, and the aftermath, i.e., the extent to which the refactor can be considered successful. Finally, the findings of our assessment method were compared to the statements of the interviewees for each case. The validity of our assessment method depended on whether it could reflect the same evolution in maintainability as experienced by the experts.



FIGURE 3.1: Steps of our research strategy

3.2 Case Selection Requirements

To constitute a proper validation for the assessment identified in the previous phase, some standard is needed to which our assessment can measure up. An accessible option would be the opinion of an expert who has been actively involved in the development of a system, as this would make him a golden standard for any reasoning about that system’s granularity. To not fully rely on the expert’s *competency* regarding granularity decisions, a second golden standard has been based on the *experiences* of the expert with the project, in the sense of how the expert experienced these decisions. Engineers who were actively involved in the development of an application, are likely to have experienced any fluctuations in the maintainability of their application over time. By focusing on applications in which refactors that affect the granularity of the system took place, involved engineers can be interviewed to report how these changes in granularity affected their application’s maintainability in their opinion. In the validation phase, having this information on their experiences allowed us to compare the maintainability evolution our assessment framework showed with the evolution experienced by the experts, in reaction to changes in the granularity level that took place. This second validation method required us to be able to assess older versions of an application, i.e., before a refactor took place. Version control data can provide access to these older versions of an application, which is why the availability of this data was a requirement for the cases we assessed.

In order to find suitable MSAs for this study we formulated the following set of criteria:

- The application should be microservice-based.
- The source code should be accessible.
- Version control data should be available.
- The application should be implemented to address real-life business needs, i.e., it should not be a sample application, for instance, to demonstrate a pattern or usage of a framework.
- At least one refactor, in which the granularity of the application was affected, took place.
- Version control data should be available through which the code before the refactoring took place can be recovered.
- The intentions for this refactor and the outcome, as in accordance with the experience of the engineers working on the application, should be retrievable.

To be able to draw any conclusions on the generalizability of our assessment, testing it on only one case would have been insufficient, as it is then hard to rule out chance and speculate about whether the assessment would be applicable in other contexts as well. That is why we included three different contexts (i.e., projects) in this study that fulfil all the aforementioned criteria. A goal herein was to find three cases which are different from each other in such a way, that together they can represent a wide range of MSAs. This is why we aimed to include both a:

- Smaller application consisting of <3 services and a large application consisting of 10> services.
- MSA using orchestration and one based on choreography.
- System using synchronous API calls and system communicating asynchronously via events.
- One open-source project and one project developed in an enterprise context.

3.3 Tooling

To instrument our research, we identified available approaches to derive the metrics introduced in Chapter 2 from existing applications. As the main motivation of our research is to support practitioners in the granularity decisions they face, the ease of use of the approaches required attention. In order to ultimately be able to hand practitioners a tool in which these approaches are combined, each approach had to allow for automation in order to be included. Manual derivation of metrics is often not feasible in MSA due to the relatively high number of components MSA is known for [Bogner et al., 2017a]. To ensure the feasibility of our research, in which we aimed to perform our assessment on at least 3 different cases, another requirement was that the metrics were derivable statically (i.e., not only during runtime). It would have required a significant effort to get each system under analysis to run correctly locally, and as we did not want to exclude larger enterprise-sized projects from our analysis, requiring access to the in-vivo system would have been a very limiting criterion.

3.3.1 Instrumentation

In this section, the set of tools is discussed which was used to apply our assessment method to the cases. This set was selected while respecting the criteria formulated in the previous section. We aimed to select an instrumentation set which enabled automatic derivation and calculation of metrics to a large extent. This automation requirement, which is less prevalent among SOA-based metrics, is caused by the high number of small services that MSA often consist of, compared to traditional SOA [Bogner et al., 2017a]. To preserve the feasibility of deriving metrics in MSA, automation is essential. The instrumentation presented in this section however does not allow for full automation: for some metrics, manual interventions were inevitable and additional steps that had to be taken were project-specific. Such data preparation steps are discussed in detail in Chapter 4.

MicroDepGraph

As [Perepletchikov et al., 2007] explains, coupling is a broad term due to the various levels of abstraction on which components can be coupled. Originally, the term was defined as

the degree of association between two components, and as back then most systems were procedural, the only connection types defined were data and control. Alternative types of connections were required to make the notion of coupling applicable to object-oriented systems, which came with several new coupling dimensions, such as, e.g., inheritance coupling [Perepletchikov et al., 2007, Chidamber and Kemerer, 1994]. Although this enforced the development of new coupling metrics and an extension of the definition, these new metrics were not directly applicable to SOA. This is explainable because one of the main design principles in SOA is to reuse services only through their interfaces, without any direct calls between the services themselves [Perepletchikov et al., 2007, Erl, 2005, Singh and Huhns, 2005]. Therefore, essentially, a new layer of abstraction is added.

As we were interested in the interplay between coupling and service granularity, we aimed to measure structural coupling at the level of services, which matches the abstraction level the structural coupling metric introduced in Section 2.4 was proposed for by [Panichella et al., 2021]. [Panichella et al., 2021] also proposed MicroDepGraph, which is the tool we used in this research to extract the dependencies between the microservices in an application from the dependencies defined in a Dockerfile and to calculate the structural coupling [Rahman and Taibi, nd]. The tool checks a specified folder for a docker-compose.yml file, which is commonly used to define the entire service composition of a system. The dependencies are subsequently derived from the “depends on” section of the file. This section is used by Docker to understand which services a service depends on, and Docker adheres to the dependency order when starting up and shutting down services [Docker, nd].

Due to the focus of the tool on Docker configurations, the tool has low applicability in event-driven architectures in which services produce and consume to and from an event bus. Event-driven architectures adhere to the loose coupling principle, in the sense that services are agnostic of which services consume their events and of which services produced the events they consume [IBM, nd]. The tool might be able to identify the dependencies between the components interacting with the message broker in place, but the dependencies between services remain outside the scope of the analysis.

Panichella’s metric does not allow for any weighting of the coupling between two services, in contrast to the metrics of Perepletchikov in which the number of calling implementations was counted [Bogner et al., 2017a]. Although such a lower-level coupling analysis might have additional value, as it allows a finer-grained insight into a system’s maintainability, to our knowledge there is no tooling yet which automatically derives these metrics from MSA.

Code-Maat

Several tools can calculate the change coupling between two artefacts over a specified time period. In this research, we used the tool Code-Maat, as the tool is open-source and allows for highly customisable analysis [Tornhill, nd]. The tool mines version control data and is able to perform several analysis types, such as code age analysis, ownership analysis (in which the distribution of efforts among developers on a certain artefact can be extracted) and change coupling analysis.

Code-Maat implements change-coupling analysis in a way that corresponds to our definition of change coupling given in Section 2.4. Code-Maat allows for the specification of a temporal window, in which all commits should be considered to be part of the same

logical change set. This temporal window can be one or more days. Code-Maat groups the commits using a sliding time window, as depicted in Figure 2.5, and from this point on the analysis is independent of how the change sets were grouped (using a time window or based on commits). Subsequently, for each coupled pair i, j the number of shared revisions is counted, next to the total number of revisions for both entities i and j . The actual coupling degree is calculated by the tool as the number of shared revisions of two entities i and j , divided by the average of the individual revision numbers of i and j :

$$CC(i, j) = \frac{\text{shared_revisions}(i, j)}{\left(\frac{\text{total_revisions}(i) + \text{total_revisions}(j)}{2}\right)} \quad (3.1)$$

Code-Maat generates a table in which each entry contains the change coupling defined as in the formula above, the entity names, the number of shared revisions and the average number of revisions of the two entities. This coupling relation is based on the average number of revisions and therefore undirected. Code-Maat also offers the option to give a more verbose output of the analysis, which adds the total number of revisions per entity to the generated entries. This information enabled us to calculate the directed coupling relations between the entities ourselves, using the formula presented in Section 2.4.

Change frequency

We decided to implement our own tool to calculate the CF of a service. The implementation is straightforward: the version control history of a service S is analysed and we count the number of commits (C) that were committed within the time interval under analysis. C is subsequently divided over the number of months (M) the time interval spanned to determine the CF. By doing so, we harmonise the values which can now be compared, as the absolute numbers of changes were accumulated over different periods of time in each project and refactor. The formula below summarises this calculation:

$$CF_S = \frac{C}{M} \quad (3.2)$$

cloc

We used an open-source tool named cloc to measure the LOC of a service [cloc]. cloc is able to recognize a wide range of programming languages and can differentiate between comment lines, code lines and blank lines for each of these languages. It uses an algorithm which:

1. Recursively lists all files in a specified directory
2. Based on the file extensions determines in which language the file content is written
3. Opens files without an extension and reads the first line to determine the language
4. Counts the number of original lines
5. Removes the blank lines and repeats the counting
6. Applies comment filters tailored to the programming language of a file, which deletes the comments, resulting in the residual consisting only of lines of code, which is the equivalent of the LOC metric as defined in Section 2.4.

Finally, the average LOC (LOCavg) for all services in an application has been calculated, to be able to identify services with a size substantially larger than the average, which might be candidates for refactoring.

RAMA-CLI

RAMA-CLI is a tool developed by a research group at the University of Stuttgart to derive maintainability metrics from interface specifications [Rama-Cli, nd]. It is a command-line tool instrumented to calculate 9 different metrics, which represent either size, complexity or cohesion. The tool can parse three types of RESTful API specification languages, namely OpenAPI, RAML and WADL. The metrics calculated by RAMA-CLI of interest in this research are service interface data cohesion (SIDC) and weighted service interface count (WSIC).

SIDC quantifies to which degree a service S is cohesive based on the similarity of data types used in the operations specified in the services interface [Bogner, Perpetklov]. To calculate the SIDC of a service, RAMA-CLI evaluates the number of operations defined in the interface which have common input parameter types and the number of operations with common response types. The final formula used to calculate the SIDC deviates slightly from the one in Section 2.4: in RAMA-CLI operation pairs with common input types and operation pairs with common response types are counted separately. Subsequently, the sum of these numbers of pairs is divided by the number of all operation pairs possible for the service interface under analysis. To calculate WSIC, RAMA-CLI simply counts the number of distinct methods exposed in an API specification.

As RAMA-CLI has been designed specifically for the analysis of RESTful APIs, the tool is not directly applicable to other types of service interfaces. In event-driven architectures, services can produce and consume events in parallel to exposing a RESTful interface, but for services which are not required to communicate with the RESTful interface of a front-end service, exposing any REST endpoints might be redundant. RAMA-CLI is not able to calculate SIDC for such back-end services in an event-driven architecture which are not exposing any relevant endpoints. Such services will only expose an endpoint to the message broker to post events to, but the calculated metrics would not be of value as the interface does not correspond to the functionalities implemented in the service [Dhanushka, 2021].

Chapter 4

Data Collection

In this chapter, we explain how the data collection for this research was performed. We first discuss the cases that were studied during this research and the refactors that took place during the life cycle of the cases, which have been the focus of this study. The presented information is the result of unstructured interviews with the case experts, during which they were interviewed on the refactors that were carried out, the intentions for those refactors and how they perceived each refactor to have influenced the maintainability of their system. Subsequently, we provide the rationale behind our data preparation steps and present an overview of data preparation guidelines which should be respected to correctly apply our assessment method.

4.1 Case 1: Metadata

Metadata is a microservice-based metadata-driven user interface (UI) provider. It is an open-source project, developed by one developer working solo, and is available on GitHub [Visockis, nd]. The application offers a solution to teams with a high back-end competence and focus, rather than UI, by offering metadata-driven UI generation. Teams responsible for database management and access are a representative example: for such database access systems a minimalistic front-end suffices, especially if it is only used internally. Metadata essentially aligns elements by invoking a set of endpoints or queries that provide UI metadata specified by the user, such as the cardinality, font size and font. It allows for the specification of the UI metadata via REST endpoints, as well as via GraphQL queries. The architecture of the Metadata system can be found in Figure 4.1, which shows that only four microservices are present in the system's architecture; the other modules are provided as binaries instead of services, to protect the user from having to deploy yet another microservice. Based on whether the user decides to use the REST endpoints or the GraphQL queries, only the corresponding two microservices will be active. The two pairs of microservices will not have to run simultaneously and are structurally completely independent of each other.

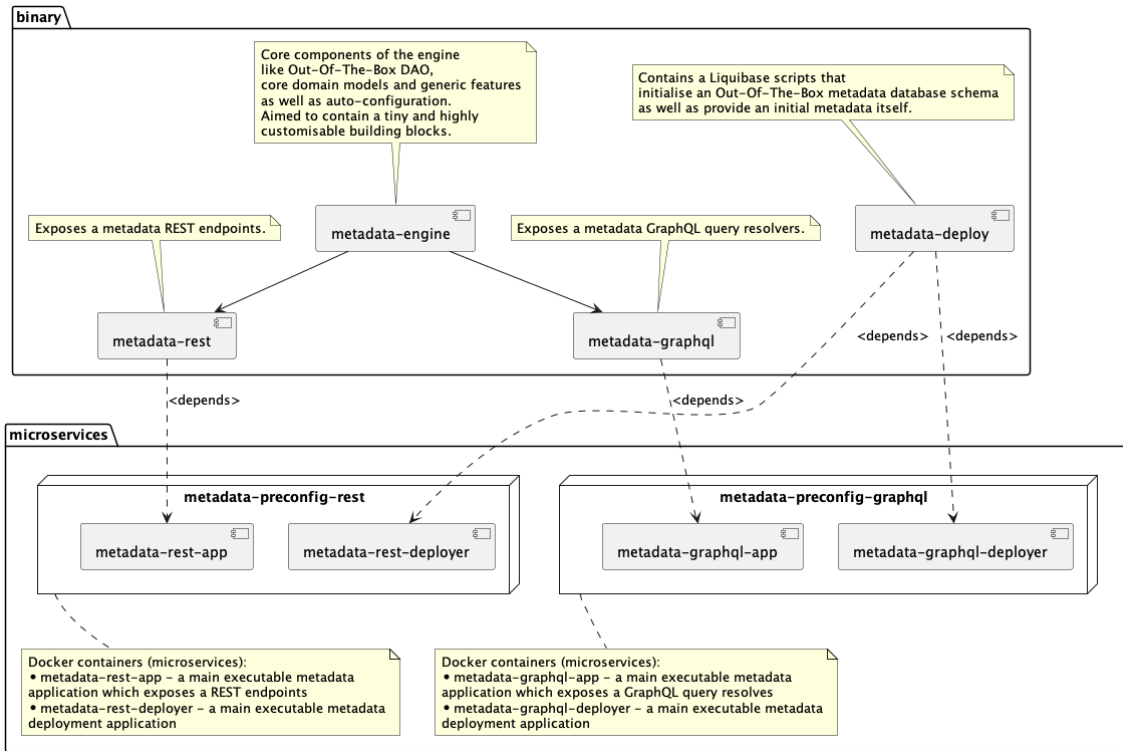


FIGURE 4.1: Overview of Metadata’s architecture [Visockis, nd]

4.1.1 R1.1

Throughout the development process of Metadata, one significant refactor took place. In the architecture depicted in Figure 4.1, Metadata’s current architecture, the `metadata-app` and `metadata-deployer` are two separate services (both in the case of the REST-oriented and of the GraphQL-oriented components). Initially, the deployer and application were a single service however, forming the `ref-impl` service. To increase the separation of concerns, as is in line with the single responsibility principle which is a design principle in MSA, the developer decided to decompose this initial `ref-impl` service. By doing this, functionality related to data management now resides in `metadata-deployer`, whereas the operations which are performed on the data are all in the `metadata-provider-app` of their corresponding provider.

4.1.2 R1.2

The `metadata-rest-app` and the `metadata-graphql-app` have a lot of functionality in common. Besides the REST and GraphQL-specific code, the operations available to both services are similar as in the end they implement the same features while only using a different type of API. Currently, adding a field to a domain model in the `metadata-engine`, which is used by both services, requires one to also add this field to both the `graphql-app` and the `rest-app` to keep both services compatible with the other components of the system. The developer of Metadata pointed out the common functionality of the services to be a candidate for being extracted into a separate service. However, this refactor was never

actually carried out, as the developer foresaw how this would negatively affect the usability of the application: extracting the shared functionality into a separate service would require the user to also deploy this extra service, which results in a possible deployment overhead. Another option would be to add the common functionality to the binary depicted in Figure 4.1, but this would reduce implementation freedom, as it will no longer be possible to modify fields in the graphql-app and rest-app independently from each other. We include this refactor in our study, although it was never actually implemented, as we want to investigate to what extent the described trade-off is reflected by our assessment of the system.

4.2 Case 2: Loan eligibility checker

The second case we study is an industrial project, in which a bank wanted to automate the process for small and medium-sized enterprises (SMEs) to validate their eligibility for a loan. This system has primarily been implemented using Java and has been designed and developed by a team consisting of over 40 engineers. It has been under active development for four years. The customer journey of this system, which gives an impression of the system's features, is as follows:

1. The user submits some necessary information: company name, the loan being requested, the reason, and the company's structure.
2. The eligibility of the user for a loan is calculated by the system.
3. Additional information is requested by the system, based on the reason for the loan request.
4. User provides transaction history.
5. Transaction data is analyzed using risk models.
6. The system returns a "loan denied" or an offer with certain conditions.
7. In case a loan is offered, the user can adjust some parameters, e.g., the loan term and indirectly the corresponding interest rate.
8. The user can accept the offer, which results in a so-called term sheet.
9. The term sheet is forwarded to the customer for a last check and the formal acceptance from the customer's side.
10. The formal acceptance leads to the money being transferred.

In the background, a workflow is followed by the system, which is depicted in Figure 4.2, which gives a schematic overview of the system architecture. In reaction to step 1 in the customer journey, the front-end makes a REST API call to the journey API service (A in Figure 4.2), which then publishes an event to Kafka's event bus (B). The rule engine (C) listens to this event. This rule engine service orchestrates the other services; the rule engine knows what the subsequent steps in the process are based on the incoming events. Triggered by the event published in step 1, the rule engine triggers the logic which checks whether the customer is eligible for a loan, and the business logic, which decides which subsequent questions should be asked to the customer, based on the reason for the loan request. Throughout the journey, state events are produced which allow their consuming

services to stay synchronised. The journey API service also listens to these state events and locally keeps track of the state of the request. By doing so, the journey API service is able to tell the front end what information to request from the customer next. This description does not explain the entire workflow, as it only focuses on the outline. Other microservices come into play to handle domain-specific functions. A few examples are given below:

- Risk model services (D): these services decide whether the loan request is approved, and decide under which conditions the loan will be offered based on the calculated risk score.
- Transaction processing service (E): processes the transaction data, provided in a PSD2 or MT940 format, to provide the risk models with input.
- PDS2 service (F): retrieves the customer’s transaction data from the customer’s bank account (with the consent of the customer).
- Term sheet service (G): gathers all the data for the term sheet and sends it to the back end of the bank.

Considering inter-service communication, only the services that communicate with the API gateway directly have REST endpoints at their disposal. The remaining services that are not part of the front-line, communicate solely by means of events. One of the engineers explained how services implemented in Python are an exception here. Although services like the risk model services (D in Figure 4.2) do not communicate with the front-end, they do expose a set of endpoints: an Apache Kafka framework is also compatible with Python-based services, but as Python does not allow multi-threading, explicit endpoints are more convenient as they can run on the main thread. Each refactor that took place during the lifecycle of the project is discussed in the sequel.

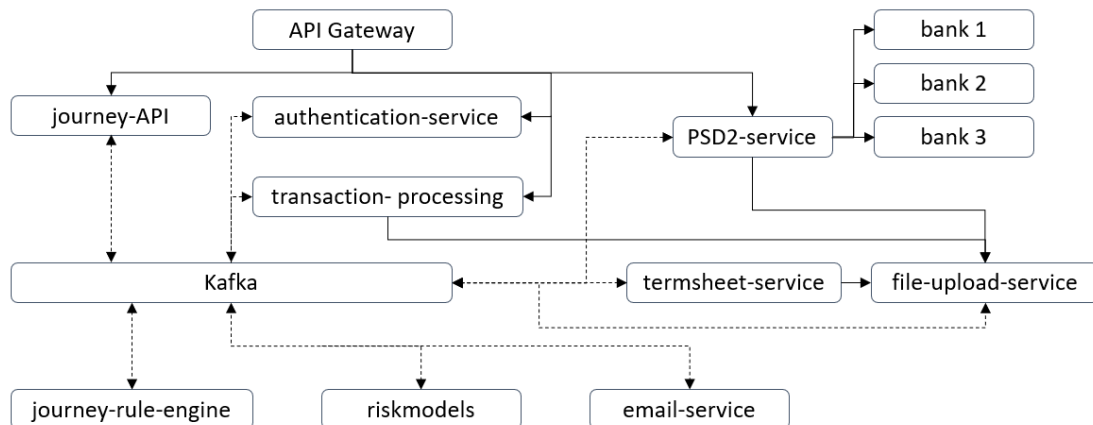


FIGURE 4.2: Microservices architecture of the lending system. Dashed lines represent events, whereas solid lines represent (synchronous) API calls.

4.2.1 R2.1

In the initial version of the system, the workflow was implemented as a choreography: there was no service which centrally controlled the flow, and services invoked each other,

i.e., communicated in a peer-to-peer style. For the initial requirements of the system, this communication pattern seemed suitable, but as soon as the reusability requirements grew, the choreography pattern became a bottleneck: to allow the reusing of the services in other journeys, every service needed functionality which enabled them to distinguish the different journeys, in order to identify which journey they were being called in. To avoid having to implement this extra check in all services, the team decided to transition to an orchestration, for which an orchestrator (the rule engine in Figure 4.2) was implemented. By doing so, a central place was created to control the workflow of a journey, which increased the maintainability of the system and most importantly: the other microservices could stay simpler, which increased their individual reusability. As this refactor affected the entire system, the team took this refactor as an opportunity to revise other aspects of the architecture as well and create a new version of the system from scratch in a new DevOps environment. To minimise the bias caused by other modifications than the introduction of an orchestrator, we have only performed our assessment on the services which directly correspond to services in the system before the refactor. Other services that underwent other refactors in parallel during the transition have been ignored. The only exception here is that the offer-api-service and offer-proxy-service were merged during the transition into the offer-service, but as the functional scope of the two services pre-refactor directly translates to the functional scope of the post-refactor service these services are not excluded from the analysis.

4.2.2 R2.2

The journey API service and the rule engine both keep track of the state of the journey. As they form the bridge between the front-end and the back-end of the system, the two services need to be in the same state for each process instance to inherently keep the front-end and back-end in a consistent state. To achieve this, they need to have the same data at their disposal, so in the architecture depicted in Figure 4.2, complex state-carrying events are constantly being exchanged by the two services. To improve the global complexity of the system and reduce the network consumption of the system, the engineering team decided that the rule-engine and journey-api should be merged. According to the engineers, this would directly benefit the maintainability of the system, not only in terms of complexity but also in flexibility, as changes to the state-carrying events would not require modifications in two services anymore. Although this refactor is acknowledged by the team to be beneficial, it is not implemented yet as it is still on the planning. This implies that we have not been able to measure the effect of the refactor, but we still analysed the system version at our disposal to see whether based on our assessment we can also identify the rule-engine and journey-api-service as candidates for merging.

4.2.3 R2.3

In the year in which the project started, a company-wide policy existed that prescribed a strict separation between business logic and the corresponding APIs. Most bounded contexts were divided over three services back then:

1. A service in which the business logic was implemented.

2. A service which implemented the API of the business logic service, and simply passed the requests and responses.
3. A service which contained the configurations to communicate with the API gateway of the system.

In a later stage, this separation policy was revised. Over the years, many of these trios, as described above, have been merged into one service implementing both the business logic and the API. Unfortunately, the repositories of the API-related and gateway-configuration services have been deleted for each trio that was merged, so we have not been able to analyse these refactors. There is still one trio however, the PSD2 microservices (F in Figure 4.2), for which this refactor has not been carried out yet. This trio is concerned with retrieving transaction data from a customer's bank. One of the engineers pointed these services out as a textbook example of a bounded context divided over multiple services. By considering these services we were able to investigate if our assessment could recognize the services as a single bounded context.

4.2.4 R2.4

By introducing the rule engine, a central point in the system was introduced from where all services were reachable. Because of this central role, over time the rule engine grew. It was a convenient place to add new features: as the rule engine handles the entire workflow, it was relatively simple to add something there. Furthermore, adding a new feature to an existing service was easier than implementing a new service for this purpose. After doing this one or two times a vicious circle started: as this recently added code is already in the rule engine, and the newest functionality needs data from that code, it is even more appealing to add this newest code to the rule engine as well. This accumulation in one service became problematic when the bank wanted to reuse certain functionality that was now mixed into the rule engine in other journeys. Using the rule engine for multiple journeys would become too complex, so the team was forced to extract the new functionality from the rule engine. During the refactor, this functionality was extracted into separate services. Among the extracted functionality were features like the first eligibility check, the overview of loan offers and the logic which decides whether a certain combination of loans is allowed to be offered.

4.2.5 R2.5

Employee journeys, which are initiated by employees of the bank, are orchestrated by the employee-rule-engine and their API gateway is implemented by the employee-api-service. These services are similar to the rule engine and api-service of the customer journeys, but tailored towards the employee journeys. The intention for R2.5 was to partly merge the employee-rule-engine and the employee-api-service to make the constant exchange of complex state events between the two services obsolete. The merge is partial as the employee-rule-engine and employee-api-service are involved in multiple employee journeys, and the merge is only done for one specific journey. This means that the employee-api-service, employee-rule-engine and this new service in which the two are partially merged, the

review-flow-engine, co-exist after the refactor. The team is planning to completely decompose the employee-rule-engine and employee-api-service into journey-flow-engines in the near future.

4.3 Case 3: Spinnaker

Spinnaker is a platform for continuous delivery, which was initiated by Netflix and evolved into a collaboration between Netflix, Google, Microsoft and Pivotal [Netflix, 2015]. It is designed to facilitate application management and deployment across different cloud environments. It enables the release of software changes with high velocity by offering a robust pipeline management system which is compatible with several major cloud providers. The project has been open-sourced and is available on GitHub [Spinnaker, c]. In Spinnaker, one can create Pipelines which correspond one-to-one to delivery processes. Such a Pipeline consists of Stages, which in their turn represent the building blocks of a traditional delivery pipeline. Examples of Stages are the baking of images for deployment or the deployment to a cloud provider. The platform also provides features for the management of deployment clusters. Figure 4.3 gives an overview of Spinnaker's architecture. The platform consists of 12 microservices, each with its own responsibility [Spinnaker, a]. Noteworthy to understand the architecture are the following services:

- Deck: browser-based UI.
- Gate: API gateway.
- Orca: orchestrator.
- Clouddriver: responsible for all calls to cloud providers.
- Rosco: bakes immutable images for deployment on various cloud providers.
- Echo: event bus.
- Fiat: authorization service.
- Halyard: configuration service which manages the lifecycle of the other services.

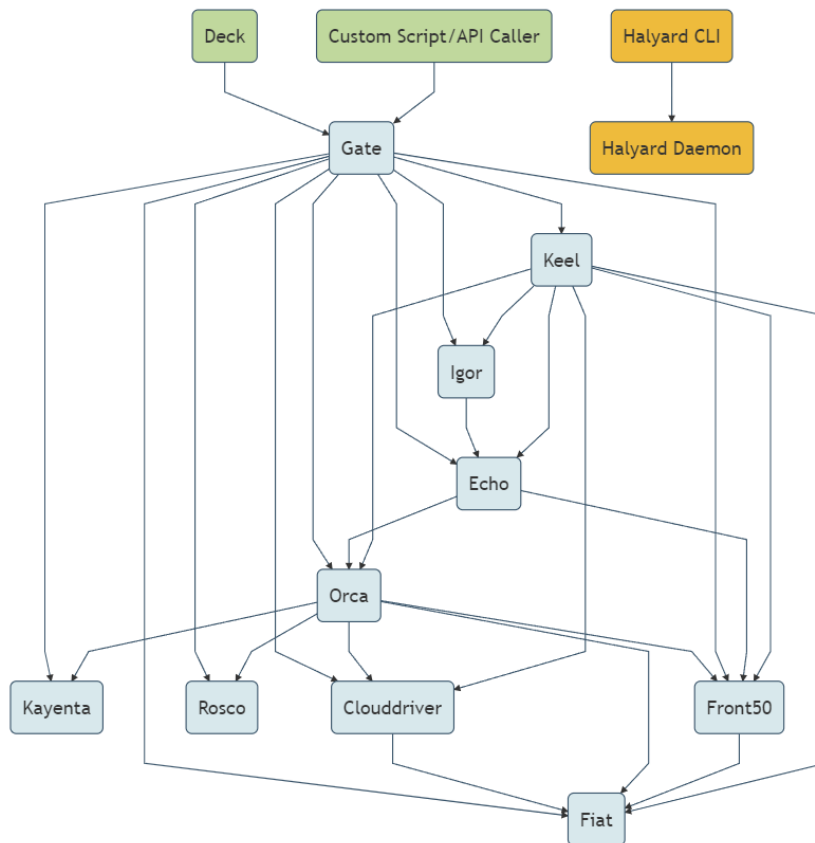


FIGURE 4.3: Overview of the Spinnaker microservices [Spinnaker, a]

Similar to the loan eligibility checker, most services do not expose an API, as the back-end services only communicate via events using Echo (the event bus). A service that does provide a RESTful interface is the API gateway of the system, known as gate.

We are not aware of any refactors performed on Spinnaker which affected the granularity, but we include Spinnaker as a case to validate the findings of our assessment on the most recent version of the system with an involved engineer.

4.4 Cases Overview

As we refer to individual refactors in subsequent sections, Table 4.1 provides an overview of the investigated projects, refactors and services. Table 4.1 serves as a reference tool to quickly locate the specific details of each refactor analysed in this report.

TABLE 4.1: Cases overview

Project	Refactor	Services	Description
Metadata	R1.1	S1.1.1: ref-impl S1.1.2: metadata-app S1.1.3: metadata-deployer	Ref-impl was decomposed into metadata-deployer and metadata-app to separate functionality related to database management from operations which are to be performed on the data.
Metadata	R1.2	S1.2.1: metadata-rest-app S1.2.2: metadata-graphql-app	The common functionality of the involved services was pointed out as a reason for partially merging the services. This refactor was never actually carried out.
LEC	R2.1	S2.1.1: authentication-service S2.1.2: email-service S2.1.3: file-upload-service S2.1.4: offer-api-service S2.1.5: offer-proxy-service S2.1.6: journey-api-service S2.1.7: riskmodel-service S2.1.8: transaction-processing-service S2.1.9: rule-engine S2.1.10: offer-service	This refactor entailed a transition from choreography to orchestration, in which the rule-engine serves as orchestrator.
LEC	R2.2	S2.2.1: journey-api-service S2.2.2: rule-engine	As the journey-api-service and rule-engine constantly exchange complex state events as they have to be in a consistent state, the team plans to merge them to increase maintainability. This refactor has not been carried out yet.
LEC	R2.3	S2.3.1: psd2-retrieval-service S2.3.2: psd2-api-service S2.3.3: psd2-token-service	These services were pointed out to be a bounded context divided over three services. This refactor has not been carried out yet.
LEC	R2.4	S2.4.1: rule-engine S2.4.2: lane-selection-service S2.4.3: source-docs-service	Due to its central position, the rule-engine became a breeding ground for new functionality. During the refactor, this functionality was extracted into the lane-selection-service and source-docs service.

Table 4.1 continued from previous page

LEC	R2.5	S2.5.1: employee-api-service S2.5.2: employee-rule-engine S2.5.3: review-flow-engine	Similar to the services in R2.2, the employee-api-service and employee-rule-engine were kept in consistent states by exchanging complex state events. These services were partially merged into the review-flow-engine.
Spinnaker	-	S3.0.1: deck S3.0.2: gate S3.0.3: orca S3.0.4: clouddriver S3.0.5: rosco S3.0.6: echo S3.0.7: fiat S3.0.8: halyard S3.0.9: igor S3.0.10: front50 S3.0.11: kayenta	In Spinnaker, we did not investigate any refactors in particular, but we performed our assessment on the latest version of the system to identify potential refactor candidates and validated our results with involved engineers.

4.5 Data Preparation

In order to assess the selected refactors, we undertook several steps to prepare and clean our data. This data preparation process is crucial for the proper application of our assessment method. Below, we provide a detailed explanation of our data preparation procedure, ensuring the reproducibility of our research. Additionally, we highlight the challenges we encountered during the process and the decisions we made to address those obstacles. The data preparation procedure is discussed in the context of a fictitious refactor as depicted in Figure 4.4.

At the end of the section, we summarise the data preparation steps required for our method in a comprehensive overview, to offer practical guidance for those interested in applying our assessment method.

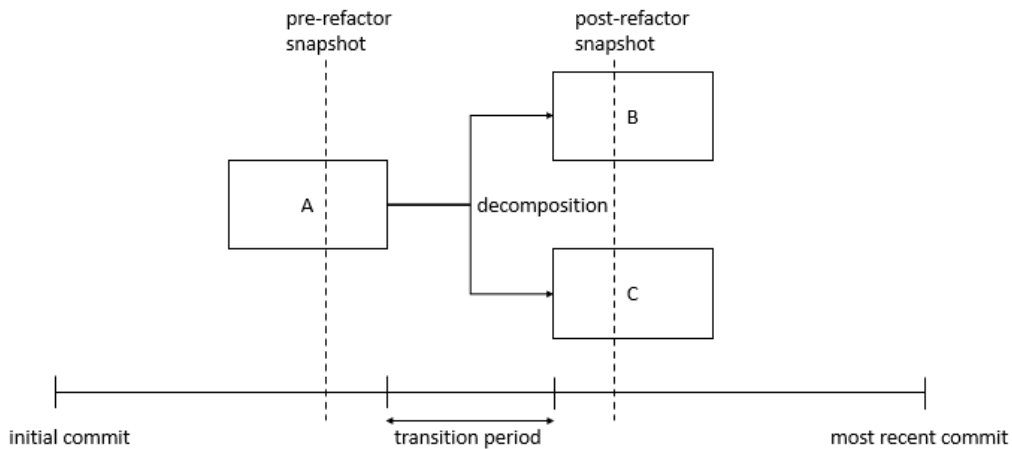


FIGURE 4.4: Illustrative example of a refactor

As all the cases studied in this research use of Git as their version control system, we discuss our data preparation process in a context in which Git is used. This approach could be adapted to use other version control systems, like Mercurial, SVN, Perforce and Team Foundation Server, without too much effort. Some of the instrumentation used, like Code-Maat, is directly compatible with different version control systems.

Determining time intervals for analysis

Of the six metrics that we wanted to measure for each refactor, two of them (change coupling and change frequency) could only be measured over a time interval. This required us to define how to determine the time intervals over which we would measure, to ensure consistency and avoid any biases resulting from an inconsistent determination of time intervals.

We explain how we consistently chose the time intervals to measure in the different refactors based on the example in Figure 4.4. The figure illustrates the decomposition of a service A into services B and C, and the snapshots indicate the points in time to measure the metrics which are not calculated over a time interval (SC, LOC, WSIC, SIDC). The first revision of all branches determines the start of the pre-refactor period. In most cases, this is performed on the master branch. Git is a version control system which allows distributed revisions, over multiple branches. While performing our assessment we do not distinguish the different branches in a project, as this would only be of value in specific use cases, e.g., when a branch which has not been merged with the default branch represents a new release and one would be interested in comparing the maintainability of different branches/versions of the system.

In Figure 4.4, the pre-refactor period is the interval between the initial commit and the start of the transition period. We defined the transition period as the interval between the first revision which was related to the refactor under study and the latest revision which was related to this refactor. In some projects, the start and end of the transition period are represented by the same revision, in case the whole refactor was implemented in a single commit. In other, mainly larger projects, a refactor can be implemented in stages. These revisions done in these periods are irrelevant for our analysis as they would mainly create bias. In Figure 4.4, by including the transition period in our analysis, either service A would seem to have a higher change frequency or service B and C would seem more coupled, depending on if the transition period is included in the pre-refactor or post-refactor period.

Preparing the version control logfiles

Git allows its users to extract the entire version control history of a repository, in which all branches are included, by a single command. Git's CLI also allows the specification of a before or since tag, which lets Git return all revisions before or after a specified date. We use this feature to retrieve log files of both the period before and after the refactor. For the refactors that have not been carried out yet, we cannot measure post-refactor, so in those cases, we create a log file of the entire version control history (from the initial commit to the most recent commit). After extracting the log files of a repository, we clean the data by filtering out several types of commits which are discussed in the sequel.

Bot commits

In the Spinnaker project, numerous revisions present in the version control history were committed by bots. A bot can be in place for several reasons. In Spinnaker's case, there

were two bots for different purposes: one was designed by the Spinnaker team itself, to help manage the different repositories by handling GitHub events and apply policies to issues and pull requests [Spinnaker, b], and another one was provided by Mergify, to help in the process of merging pull requests in different repositories and facilitate the CI/CD process [Mergify, nd]. As revisions committed by bots do not reflect maintenance behaviour, we exclude them from our analysis.

Overloaded commits

The accuracy of a change coupling analysis depends on the behaviour of the developers to a great extent: while some developers might commit their modifications more frequently, others might tend to commit all their changes at once. In the latter case, this can result in overloaded commits in which several unrelated changes are covered [Zimmermann et al., 2003, Oliva and Gerosa, 2015]. Such overloaded commits create artificial change couplings and should be accounted for by removing them from the log files [Oliva and Gerosa, 2015]. In our cases, we only excluded commits affecting more than 100 files from the analysis, as larger commits were often statistical outliers. This mainly removed the commits involving an extremely high number of files, for example, a library which was directly added to the version control system, which was renamed or refactored internally. Choosing a lower threshold is not without risk, as it might accidentally exclude relevant changes from the analysis [Oliva and Gerosa, 2015].

[Oliva and Gerosa, 2015] also mentions incomplete commits, referring to commits which only represent a part of a change as the other parts of the change are represented by separate commits, to be another threat to the accuracy of a change coupling analysis. In general, these related commits should be treated as a single change set, based on a task id, for instance. As each case we studied in this research consists of multiple repositories, and subsequently change coupling between the repositories cannot be derived from an analysis on the level of commits since each repository has its own version control history, we determine our change sets based on a temporal window. By using a temporal window of one day, for instance, three related commits which were committed individually over a day are grouped correctly into the same change set.

Merge commits

[Zimmermann et al., 2004] recommends excluding merge commits. A relevant detail is that Zimmermann et al. performed their work on Concurrent Versions System data, an alternative version control system which was popular in the year of writing [GNU, nd]. They argue that merge commits, in which two branches are being merged, often contain unrelated changes and that changes in branches are ranked higher in a merge commit. This last argument is no longer valid in the context of this work, as in Git, since during a merge conflict in Git, the developer can specify which branch should overrule the other and even manual intervention is allowed [Atlassian, nd].

Addressing renames

A service under analysis could have been renamed or refactored during the time interval of interest. This complicates a change coupling analysis since it can reduce the accuracy of the analysis. We illustrate this problem with an example of a refactor in Figure 4.5. The figure illustrates the decomposition of service A into services B and C, followed by the subsequent decomposition of service B into services D and E. When applying the assessment method to learn about the influence of decomposition 1 on maintainability, the period of interest for this assessment is defined as the interval between the finalisation of

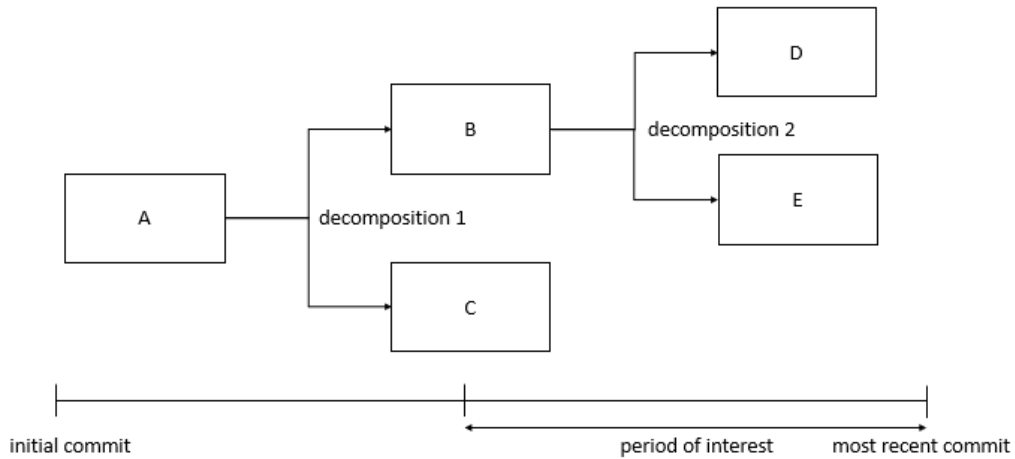


FIGURE 4.5: Fictitious service under analysis (B) evolving into two other services (D,E)

decomposition 1 and the most recent revision. Here, an obstacle is encountered as after decomposition 2, service B ceases to exist as it has been decomposed. Now only the interval between decomposition 1 and 2 can be analysed to learn about the effects of the refactor. However, since services D and E collectively implement the same functionality as B, it is reasonable to treat the pair as a representation of service B during the analysis. This approach allows one to consider the entire time interval after decomposition 1, instead of only the interval between decomposition 1 and 2, enhancing the accuracy of the metrics. In practice, we addressed this issue by implementing a labelling mechanism. We appended a prefix to the lines in the Git logs of services D and E, designating them as service B for the purpose of our assessment. The exact implementation of this labelling depends on the version control system in place. Our GitHub page¹ contains scripts that can be customized or extended to fit a project’s individual needs.

Analysing change coupling

For the calculation of the change couplings, we completely relied on Code-Maat, a command line tool for mining data from version-control systems. Their GitHub page² contains an extensive user manual, so here we only provide a general explanation on the command line options which were relevant for our analysis. As the tool only allows a single log file as input, the last data preparation step is to concatenate the log files of the different repositories. After a concatenation, the information on to which service a commit belongs to might get lost, as filenames are not always conclusive (think of common files such as pom.xml). To address this, before concatenating we labeled the files of each commit with the name of the service. Both concatenation and labelling are only necessary for a multi-repository project. In our analysis, we used the following command line options offered by the tool:

- We set the minimum number of revisions to include an entity in the analysis to 1, just as the minimum number of shared revisions.
- We set the minimum degree of coupling to consider to 1, and the maximum to 100, to be sure to retrieve all couplings.

¹<https://github.com/famkedriessen/quantitative-assessment-method-for-microservices-granularity>

²<https://github.com/adamtornhill/code-maat>

- Just as with the overloaded commits, we set the maximum entities in a change set to 100.
- We used the verbose mode of the tool to get all outputs required to perform a directed coupling calculation afterwards. (Code-Maat does not offer this as a feature itself).
- We specified a temporal period, which lets the tool consider all commits within that time window as a single change set, instead of considering individual commits as change sets.

Temporal window

Specifying an appropriate temporal window is challenging: a temporal window which is too large might result in a higher number of false change couplings, while with a too small temporal window relevant change couplings are missed. Ideally, the temporal window should be based on the developer's behaviour, to find the sweet spot between accuracy and reliability, i.e., retrieving a minimal amount of artificial change couplings while not missing any relevant ones. Developer's behaviour, and specifically developer's commit behaviour, however, is a research area in itself. Recent studies have tried to analyse the characteristics of commit intervals in open-source projects. One study found that commit intervals often follow a power-law distribution, with most intervals being very short and only a few being distinctly long [Ma et al., 2013, Oliva and Gerosa, 2015]. Another study discovered that the number of commits per class and per time unit roughly follows a power-law distribution [Lin et al., 2013, Oliva and Gerosa, 2015]. However, commit practices are influenced by various factors, including the project's development process, the way tasks are defined, and the version control system used. Different commit behaviours may pose challenges for detecting change couplings, and further investigation is needed to determine how to handle these challenges. Relevant questions are whether each commit should be treated the same, and how periods with similar commit properties can be detected. In developer teams, should we perform an individual analysis on the behaviour of each developer or can we rely on the team following the company's commit policies? These questions require further investigation to determine how to effectively perform change coupling analysis.

The commit behaviour of developers cannot be precisely captured yet, which leaves us little support for choosing a suitable temporal window. Therefore, we decided to measure three different temporal windows, of 1, 2 and 3 days. An alternative way of grouping commits into change sets, as recommended both by [Oliva and Gerosa, 2015] and [Tørnhill, 2018] is to consider task ids, corresponding to issues of the issue tracking system in place. As our cases either did not make use of task ids or had an issue tracking system which we could not access, grouping based on temporal windows was our best viable alternative.

Calculating change frequency

Next to change coupling, CF is the only other metric which is calculated over a period rather than at a single point in time. We calculated the CF over the same pre-refactor and post-refactor time intervals that we included in our change coupling analysis. During our calculations, empty commits (e.g., merge commits) are also taken into account as they still represent a change in the system. The scripts we used are available on our GitHub page.

Resetting to older versions of the system

To enable the tool used to count the LOC (cloc), to analyse the previous version of a system, a reset to this older version is required. As LOC is measured on a single point in time, we consistently measure the service sizes right before the refactor under analysis was carried out and just after the transition period ends.

Extracting the API specification

To enable the analysis of the service interfaces using RAMA-CLI, we needed to obtain the API specification files of the services. A complicating factor was the implicit requirement that we had to be able to match an API specification to the service it represented. Two of our cases used an API gateway, a pattern in which one service, serving as the so-called gateway, represents the interface of multiple services. This made it complicated to relate the operations specified in the API specification of the gateway to the services implementing those operations. We analysed the API specifications of the services that had their own API. However, for the services that did not have their own interface and were represented by an API gateway or communicated solely through events, we were unable to calculate the interface-based metrics (SIDC and WSIC) as we did not find a suitable substitute enabling their calculation.

Dockerfiles

MicroDepGraph is the tool we used to derive the structural dependencies from a system in order to calculate the structural coupling. This tool requires Docker-compose files in which those dependencies are specified. The main hurdle of this approach was that in two of our cases, these files were not present. As both the loan eligibility checker and Spinnaker were (partly) event-driven, a central Docker-compose file in which dependencies are explicitly stated is not realistic, as event-driven architecture dictates loose coupling by design, since in this architecture services are agnostic of their consumers and producers.

Data Preparation guidelines

To facilitate the application of our assessment method, in Table 4.2 an overview of the data preparation procedure is provided, specifying the tooling, artefacts and preparation steps required for each metric. These instructions can serve as practical guidance for those interested in applying our assessment method, ensuring accurate and replicable results. By following this procedure, researchers and practitioners can effectively prepare their data and employ our assessment method in their own studies or projects.

TABLE 4.2: Summary of Data Preparation guidelines

Metric	Tooling	Artifacts	Data Preparation steps
CC	CodeMaat, scripts GitHub page	Version control history of services of interest	<ul style="list-style-type: none"> • Determine the time interval to determine the CC over, which should be the same interval as used in the CF calculation. In case the effect of a refactor is to be assessed, exclude the transition period during which the refactor of interest was implemented. • Extract the Git logs of the services of interest for the selected time interval. • Remove bot commits and commits affecting > 100 files (our GitHub contains a script dedicated to this). • In case a service of interest is renamed or refactored during the selected time interval, use our labelling scripts to label this new service(s) as part of the service of interest.
SC	MicroDepGraph	Dockerfiles	<ul style="list-style-type: none"> • Run MicroDepGraph while providing the path to the folder of the service repository. • In the case of an event-driven architecture, where the Dockerfiles do not provide information on service dependencies, an alternative approach is to manually calculate the structural coupling. A condition is that documentation on the dependencies between the services is available and complete. The formulas presented in Section 2.4 can be used to calculate the structural coupling in such cases.
WSIC	RAMA-CLI	API specification (OpenAPI, RAML or WADL)	<ul style="list-style-type: none"> • Find the API specification of the service of interest and provide the file path to RAMA-CLI. • When dealing with architectures that adhere to the API gateway pattern or follow an event-driven approach, it is important to note that not every service may have a dedicated specification. It is advisable to verify whether a specification represents a single service or multiple services, as well as whether it belongs to an actual microservice or to components of the message broker.

Table 4.2 continued from previous page

SIDC	RAMA-CLI	API specification (OpenAPI, RAML or WADL)	<ul style="list-style-type: none"> • Find the API specification of the service of interest and provide the file path to RAMA-CLI. • When dealing with architectures that adhere to the API gateway pattern or follow an event-driven approach, it is important to note that not every service may have a dedicated specification. It is advisable to verify whether a specification represents a single service or multiple services, as well as whether it belongs to an actual microservice or to components of the message broker.
LOC	cloc	Source code	<ul style="list-style-type: none"> • Using the git reset command, revert the system to the point in time of interest (this is superfluous if the goal is to analyse the most recent version of a system).
CF	Scripts GitHub page	Version control history of services of interest	<ul style="list-style-type: none"> • Determine the time interval to determine the CF over, which should be the same interval as used in the CC calculation. In case the effect of a refactor is to be assessed, exclude the transition period during which the refactor of interest was implemented. • Extract the Git logs of the services of interest for the selected time interval. • Remove bot commits and commits affecting > 100 files (our GitHub contains a script dedicated to this). • In case a service of interest is renamed or refactored during the selected time interval, use our labelling scripts to label this new service(s) as part of the service of interest.

Chapter 5

Data Analysis

In this chapter, the results of our assessment are presented for the different types of refactors which were analyzed (merges, decompositions and hybrid refactors). Because we are not aware of any refactors in Spinnaker that affect the granularity, we performed our assessment only on the most recent version of this system at the time of writing and present these results separately. In chapter 6, the assessment results are interpreted and validated with case experts. For each refactor type, we delve into the assessments, examining the results of both pre-refactor and post-refactor assessments. Throughout the chapter we present a detailed discussion of the impact of each refactor on the metric suite, providing a foundation for drawing conclusions regarding the maintainability of the analysed systems.

5.1 Merges

Both R2.2 and R2.3 are refactors which have not been actually implemented, but both have been recommended by experts to be carried out in order to increase maintainability. As they have not been implemented yet, we are only able to show the results of the pre-refactor assessments.

TABLE 5.1: Coupling metrics of R2.2 and R2.3.

refactor	service A	service B	CC	$CC_{A \rightarrow B}$	$CC_{B \rightarrow A}$	support	SC
pre R2.2	S2.2.1	S2.2.2	0.67	0.82	0.58	258	-
pre R2.3	S2.3.1	S2.3.3	0.74	0.68	0.81	78	-
pre R2.3	S2.3.2	S2.3.3	0.48	0.76	0.35	34	-
pre R2.3	S2.3.2	S2.3.1	0.45	0.8	0.32	36	-

Table 5.1 presents the bidirectional change coupling (CC) and directed change coupling ($CC_{A \rightarrow B}$, $CC_{B \rightarrow A}$) between the different service pairs. The structural coupling (SC) could not be measured for any of the service pairs. As can be seen in the table, the services involved in R2.2 exhibit a strong bidirectional change coupling of 0,67. The coupling also has a substantial support consisting of 258 shared revisions. The directed coupling is stronger from S2.2.1 to S2.2.2 than vice-versa, implying that S2.2.1 is more dependent on S2.2.2 than S2.2.2 on S2.2.1.

As for R2.3, the service pair consisting of S2.3.1 and S2.3.3 has a bidirectional change coupling value which we classify as strong based on the confidence intervals proposed by

[Oliva and Gerosa, 2011]. The third service which was pointed out to be a candidate for merging (S2.3.2) has weaker bidirectional coupling values with the other two services (both are within the regular coupling interval of [Oliva and Gerosa, 2011]). However, considering the directed coupling from S2.3.2 to S2.3.1 and S2.3.3, it seems that S2.3.2 is very dependent on both S2.3.1 and S2.3.3, with change coupling values of 0,80 and 0,76, respectively.

Table 5.2 shows the average bidirectional change coupling for all service pairs present in the application. R2.2 and R2.3 take place in the same system context, so the average metric values depicted in Table 5.3 are the same for both refactors. The average bidirectional change coupling is equal to 0,31, which is significantly lower than the coupling between the services involved in both R2.2 and R2.3. We have not been able to assess the structural coupling in either of the refactors, as the loan eligibility checker is event-driven and we do not have a complete overview of the service dependencies at our disposal. We were unable to assess the structural coupling in either of the refactors due to the event-driven nature of the loan eligibility checker, which made it unsuitable for a dependency analysis using MicroDepGraph. Due to the unavailability of a complete overview of all service dependencies, it was neither feasible to calculate the structural coupling manually.

TABLE 5.2: Average metric values for entire application (R2.2 and R2.3)

refactor	avg CC	avg SC	avg LOC	avg WSIC	avg SIDC	avg CF
pre R2.2	0.31	-	34540.90	-	-	27.34
pre R2.3	0.31	-	34540.90	-	-	27.34

Coupling is a relevant metric in the pre-assessment of a merge, as it can justify the merge to some extent. Cohesion and size metrics can complement this justification and for R2.2 and R2.3 we present them in Table 5.3. As the LOC and CF are not normalized, and we do not have any thresholds at our disposal, we discuss these values for the individual services alongside the average values of all services in the application, which are presented in Table 5.3. As we were only able to measure the WSIC and SIDC of a limited set of services in the system, as most services do not expose a RESTful API, Table 5.3 does not present any averages for those metrics. This lack of an average value to compare the WSIC to decreases its relevancy: we cannot classify a service as large based on its WSIC, as we lack insight on the size of other services in the system. For R2.2, both services consist of more lines of code than the average of the services in the application. S2.2.2 is a real outlier here with a LOC of 175942, while the average is 34540.9. The change frequencies of S2.2.1 and S2.2.2 are higher than the average CF of the services in the application, respectively 70% and 219%. Only S2.2.1 contained an analysable interface, exposing 24 distinct operations which have almost no parameters in common, resulting in a SIDC smaller than 0,01. As for R2.3, the opposite is true: all services had an under-average LOC value and CF. The LOC value of the largest service (S2.3.1), is not even half the average LOC value. Furthermore, the SIDC of S2.3.2 is 0,5, which means that the two operations defined in its interface (WSIC = 2) have half of their parameters in common.

TABLE 5.3: Cohesion and size metrics of R2.2 and R2.3

refactor	service	LOC	WSIC	SIDC	CF
pre R2.2	S2.2.1	36745	24	0.0054	46.36
pre R2.2	S2.2.2	175942	-	-	87.21
pre R2.3	S2.3.1	14021	-	-	15.93
pre R2.3	S2.3.2	1155	2.00	0.5	4.52
pre R2.3	S2.3.3	2167	-	-	11.37

The average LOC value is greatly influenced by whether all types of code lines are included in the count or only the Java or Python code in which the business logic of the service is implemented. Table 5.4 shows this difference for the services involved in R2.2 and R2.3 (which were all Java services rather than Python). The difference is significant, as by only considering the Java code, S2.2.2 is only 67 percent bigger than the average of all services, instead of the initial 409 percent.

TABLE 5.4: LOC values of the services involved in R2.2 and R2.3 (total and only Java)

service	LOC	LOC Java
average all services	34.541	7.415
S2.2.1	36.745	8.865
S2.3.1	14.021	3.729
S2.3.2	1.155	110
S2.3.3	2.167	893

5.2 Decompositions

In contrast to the aforementioned merge refactors, R1.1 and R2.4, which both entail a decomposition, were actually implemented. This enabled us to assess the application not only pre-refactor, but also post-refactor. This post-refactor assessment provides us with insights into the effect of the refactor on our metric suite.

In Table 5.5, the coupling values between the services involved in R1.1 and R2.4 are presented. The change couplings of the pre-refactor services (S1.1.1 for R1.1 and S2.4.1 for R2.4) cannot be calculated as a change coupling calculation requires two artefacts. As for the post-refactor assessments, we see bidirectional change couplings varying from weak to regular in strength. The only coupling that we classify as strong is the directed coupling from S1.1.3 to S1.1.2, implying that the former is strongly dependent on the latter. We emphasize that the support for each of these change couplings is not impressive, due to R1.1 and R2.4 being implemented quite recently, which leaves us only a small time interval to calculate the change coupling and change frequency over. Only for the service pair post-R1.1 we were able to calculate the structural coupling, which was 0.75.

With respect to the entire application, most of the bidirectional change couplings between the services resulting from the refactors are lower than the average change coupling in the systems before the refactors. In Metadata, before R1.1 was implemented the average

TABLE 5.5: Coupling metrics of R1.1 and R2.4

refactor	service A	service B	CC	CC _{A→B}	CC _{B→A}	support	SC
pre R1.1	-	-	-	-	-	-	-
post R1.1	S1.1.3	S1.1.2	0,60	0,93	0,45	14	0,75
pre R2.4	-	-	-	-	-	-	-
post R2.4	S2.4.2	S2.4.3	0,3	0,52	0,21	12	-
post R2.4	S2.4.1	S2.4.3	0,25	0,56	0,16	9	-
post R2.4	S2.4.2	S2.4.1	0,15	0,13	0,19	3	-

change coupling was 1. This average was calculated over the single pair of services that was available. As only a selection of the services is present in the application after R2.4, we are not able to calculate mean values for the different metrics of all services of the application post-refactor.

TABLE 5.6: Average metric values for the entire application (R1.1 and R2.4)

refactor	avg CC	avg SC	avg LOC	avg WSIC	avg SIDC	avg CF
pre R1.1	1.00	-	632.50	-	-	1.00
post R1.1	0.60	-	399.00	-	-	1.93
pre R2.4	0.31	-	37696.39	-	-	34.24
post R2.4	-	-	-	-	-	-

The metrics for the individual services are shown in Table 5.7. Since a high change frequency and service size in combination with a low interface cohesion can form an indication for decomposition, we first discuss the values of these metrics pre-refactor. For the LOC and CF we do this in the context of the other services of the system, by considering the averages of all services in the application as presented in Table 5.6.

For R1.1, the SIDC of 0,5 shows that the 4 operations exposed by S1.1.1 share 50% of their parameters. The LOC and CF of S1.1.1 both were lower than the average. The service was 0.66 times smaller compared to the average service, and the CF was 0.90 times lower. Since only one other service was present in the pre-refactor application, the average value is calculated only over two services. Considering R2.4, we observed that S2.4.1 scored far above average for both the LOC and the CF, being 4,43 times the average service size and having a CF which is 4,52 times the average. Furthermore, the LOC of S2.4.1 is high mainly because of the lines of JSON code, which form 81% of the total number of LOC. When ignoring the lines of JSON, the size is still larger than average. The post-refactor metric values show that each service resulting from the decomposition is smaller in terms of LOC than its ancestor, except in the case of S2.4.1, which increased in size by 3%. Only considering lines of Java code, the same service decreased in size during the refactor by 10,7%. The change frequencies of the post-refactor services are all lower than the CF of their ancestor, except in the case of S2.4.3 resulting from R2.4. As S1.1.2 inherits the complete interface of S1.1.1, their WSIC and SIDC are equal.

TABLE 5.7: Cohesion and size metrics of R1.1 and R2.4

refactor	service	LOC	WSIC	SIDC	CF
pre R1.1	S1.1.1	417.00	4.00	0.5	9
post R1.1	S1.1.3	93.00	-	-	1.07
post R1.1	S1.1.2	279.00	4.00	0.5	2.8
pre R2.4	S2.4.1	166981.00	-	-	154.36
post R2.4	S2.4.1	172167.00	-	-	84
post R2.4	S2.4.2	4396.00	-	-	82
post R2.4	S2.4.3	19417	-	-	201

5.3 Hybrid refactors

In the three hybrid refactors discussed here (R1.2, R2.1 and R2.5) a merge as well as a decomposition took place. During these refactors, the functionality of different services was extracted and combined into (a) new service(s). R1.2 has never been implemented and is mainly a recommendation from the engineer who developed Metadata, so only the pre-refactor metrics are calculated for this individual refactor. For R2.1 the engineering team switched from a peer-to-peer communication style to an orchestration, which affected a lot of services so we only analysed the eight services that we could match to each other pre- and post-refactor. Since presenting the change couplings between all possible service pairs requires a lot of space without adding much value, the coupling values of those pairs are omitted here but can be found in our GitHub repository.

Table 5.8 shows that the pre-refactor pairs in both R1.2 and R2.5 have strong change coupling values, which are equal to (R1.2) or higher than (R2.5) the average change coupling for all service pairs in the system (see Table 5.9 for the system’s averages per refactor phase). The SC of the service-pair pre-R1.2 is zero since these services are never active within the same system instance. For R1.2, we are not able to tell how the metrics evolve in reaction to the refactor, but for R2.5 we observe strong coupling values. Each of these post-refactor couplings has a support of six, which is significantly less than the support of the pre-refactor coupling of S2.5.1 and S2.5.2. The directed couplings from S2.5.2 to S2.5.3 and S2.5.1 are even 1, implying that S2.5.2 is completely dependent on the other two services. Next to the low support, it is also of relevance that the period over which the post-refactor metrics of R2.5 were calculated is relatively short. As for R2.1, we only discuss the averages of the services, which are presented in Table 5.9. Pre-refactor, we observe an average change coupling of 0.14. After re-implementing the system as an orchestration, the average change coupling increased to 0.31. In Table 5.9, we also see that the change coupling for the service pair in R1.2 is equal to the average change coupling of the system. This is not a coincidence, as this service pair is the only service pair present in the system of R1.2. As for R2.5, we observe that the pre-refactor change coupling between S2.5.1 and S2.5.2 is 2,87 times higher than the average change coupling in the system pre-refactor. Post-refactor we are not able to provide any averages as only the services involved in the refactor were at our disposal rather than all services the application is composed of. Since R1.2, R2.1 and R2.5 entail not only a merge but also a decomposition, the size and cohesion metrics depicted in Table 5.10 become of extra relevance. We discuss the LOC and CF values of the services with respect to the corresponding system’s averages shown in Table 5.9. As we were not able to calculate the interface-based metrics for all services

TABLE 5.8: Coupling metrics for R1.2, R2.1 and R2.5

refactor	service A	service B	CC	CC _{A→B}	CC _{B→A}	support	SC
pre R1.2	S1.2.2	S1.2.1	0.66	0.64	0.7	7	0
pre/post R2.1	see GitHub page	see GitHub page	see GitHub page	see GitHub page	see GitHub page	see GitHub page	-
pre R2.5	S2.5.1	S2.5.2	0.89	0.91	0.89	201	-
post R2.5	S2.5.1	S2.5.2	0.92	0.86	1	6	-
post R2.5	S2.5.2	S2.5.3	0.92	1	0.86	6	-
post R2.5	S2.5.1	S2.5.3	0.85	0.86	0.86	6	-

TABLE 5.9: Average metric values for entire application (R1.2, R2.1 and R2.5)

refactor	avg CC	avg SC	avg LOC	avg WSIC	avg SIDC	avg CF
pre R1.2	0.66	0	951.50	-	-	2.05
pre R2.1	0.14	-	37.349.75	-	-	24.86
post R2.1	0.31	-	34.540.90	-	-	27.34
pre R2.5	0.31	-	34.540.90	-	-	27.34
post R2.5	-	-	-	-	-	-

in the applications, we do not provide any averages for those metrics.

Considering R1.2, the average values do not provide us with new information, as they were calculated over the only two services in the system, narrowing down to S1.2.2 and S1.2.1. For R2.1, we see how the average service size decreases by 8,1%, while the average CF increases by 9,9% post-refactor. Pre-refactor the lowest CF (0.44 revisions per month) belongs to S2.1.5, a service that solely handles the communication with the API gateway in place. As for the interface-based metrics, we are not able to observe their evolution during R2.1, as most services did not expose a RESTful interface pre-refactor. Only for the S2.1.4, post-refactor merged with S2.1.5 into S2.1.10, we observe an increase of 13 operations in the WSIC, which simultaneously reduces the SIDC by 79,5%.

The entries corresponding to R2.5 show that S2.5.1 and S2.5.2 were both significantly larger than the average service pre-refactor, 509,6% and 218,8%, respectively. As mentioned earlier, we are not able to compare their post-refactor sizes to an average value, as we were not able to assess all services in the system post-refactor. S2.5.1 increased slightly in size, whereas the LOC of S2.5.2 decreased. S2.5.3 was extracted during this refactor and had the highest change-frequency post-refactor. The change frequencies of S2.5.1 increased by 93,3%, while that of S2.5.2 decreased by 6,6%. The interface-based metrics were not affected by the refactor, in the sense that the WSIC and SIDC of S2.5.1 are equal pre-refactor and post-refactor. The SIDC shows that this interface is weakly cohesive, with a value of 0,2875.

TABLE 5.10: Cohesion and size metrics for R1.2, R2.1 and R2.5

refactor	service	LOC	WSIC	SIDC	CF
pre R1.2	S1.2.2	1189.00	-	-	1.3
pre R1.2	S1.2.1	714.00	4.00	0.5	2.8
pre R2.1	S2.1.1	1510.00	-	-	6.61
pre R2.1	S2.1.2	38910.00	-	-	69.83
pre R2.1	S2.1.3	1.584.00	-	-	2.33
pre R2.1	S2.1.4	13.241.00	6	0.5	26.44
pre R2.1	S2.1.5	319	-	-	0.44
pre R2.1	S2.1.6	11345	8	0	1.11
pre R2.1	S2.1.7	74673	-	-	47.28
pre R2.1	S2.1.8	157216	-	-	44.83
post R2.1	S2.1.9	175942	-	-	87.21
post R2.1	S2.1.1	2544	-	-	26.5
post R2.1	S2.1.2	3370	-	-	91
post R2.1	S2.1.3	2640	18	0.5621	14.44
post R2.1	S2.1.10	2231	19	0.1023	16.25
post R2.1	S2.1.6	3533	24	0.0054	46.36
post R2.1	S2.1.7	2497	-	-	10.04
post R2.1	S2.1.8	3611	16	0.55	37.5
pre R2.5	S2.5.1	75592	29	0.2875	172
pre R2.5	S2.5.2	176040	-	-	170.73
post R2.5	S2.5.1	75943	29	0.2875	332.14
post R2.5	S2.5.2	171476	-	-	159.43
post R2.5	S2.5.3	383006	6	0.5	469.43

5.4 Spinnaker

All results of our assessment on Spinnaker can be found in our GitHub repository¹. The assessment reveals that a certain subset of services, consisting of S3.0.6, S3.0.9, S3.0.2, S3.0.7 and S3.0.10, are frequently modified on the same day as a set of three services consisting of S3.0.3, S3.0.4 and S3.0.1, for >86% of the revisions of these services. This trio also has the highest bidirectional change couplings present in the system, which we each classify as strong. Table 5.11 shows these couplings together with the next highest coupling degrees in which neither of the services is a member of the trio.

¹<https://github.com/famkedriessen/quantitative-assessment-of-microservices-granularity>

TABLE 5.11: Coupling metrics for a selection of the service pairs of Spinnaker

service A	service B	CC	CC _{A→B}	CC _{B→A}	support
S3.0.4	S3.0.1	0,79	0,82	0,77	1582
S3.0.1	S3.0.3	0,77	0,69	0,87	1424
S3.0.4	S3.0.3	0,76	0,70	0,83	1355
S3.0.10	S3.0.2	0,42	0,50	0,36	288
S3.0.6	S3.0.9	0,42	0,38	0,47	223

Analyzing the trio further, we observe that these services each have a higher CF compared to the average, varying from 87,4% to 324,6% more, as can be seen in Tables 5.12 and 5.13. The LOC value of these services is also higher than the average, i.e., the services are 64,1-213,3% larger than an average service in Spinnaker.

TABLE 5.12: Cohesion and size metrics for the services S3.0.4, S3.0.1 and S3.0.3

service	LOC	WSIC	SIDC	CF
S3.0.4	302131.00	-	-	67.13
S3.0.1	277895.00	-	-	143.72
S3.0.3	158256.00	-	-	63.44

TABLE 5.13: Average metric values for entire Spinnaker application

avg CC	avg SC	avg LOC	avg WSIC	avg SIDC	avg CF
35.89	-	96424.36	-	-	33.85

In this chapter, we presented the results of the assessments. Where CC, LOC and CF could be calculated in every context, we were not able to calculate the SC and the interface-based metrics (SIDC and WSIC) for every case, as these had limited applicability in event-driven architectures. The assessments discussed in this chapter form the basis for the validation of our assessment method, which is discussed in Chapter 6.

Chapter 6

Validation

In this chapter, the validation of our assessment method is discussed. As described in Chapter 3, our validation strategy is based on how the experts perceived the impact of each refactor on the system’s maintainability. Hence, insight into these experiences and into the initial intentions of the experts to implement the refactors was crucial for our validation. Such insight was gained by conducting an interview with each case expert, of which the results are described in Chapter 4. In Section 6.1, we first explain how we interpreted the results from our assessment, i.e., how we drew conclusions on the evolution of the systems’ maintainability from the raw metric values, which allowed for a comparison with the observations of the experts. Subsequently, we compare the observations of our assessment with those of the experts and discuss the extent to which the former matches the latter.

A second round of interviews was conducted after performing our analysis, during which we discussed the results of our assessment with the experts. These validation interviews complemented the initial interviews, as we were now able to discuss the discrepancies we discovered between our assessment and the experiences of the expert. Discussing these discrepancies in detail allowed us to learn about system-specific details which could influence our assessment. The results from these interviews are discussed in Section 6.2.

Finally, in Section 6.3, we discuss the overall validity, generalisability and limitations of our research and objectively describe potential research biases.

6.1 Assessment Observations

In order to validate our assessment using the results of the expert interviews, our initial step was to extract the conclusions regarding maintainability from the assessment outcomes. To enable a consistent interpretation of our results, we summarised how the metrics should be interpreted in different contexts. The resulting interpretation framework can be seen in Table 6.1. Subsequently, we compared and contrasted the resulting observations with those of the experts, to determine to what degree they align. In order to generalize the discussion on the degree to which the assessment captured the actual maintainability, in this section we discuss the validation process per type of refactor: merges, decompositions and hybrid refactors.

TABLE 6.1: Framework supporting the interpretation of the metric values in different refactor contexts.

Metric	Merge	Decomposition
CC	pre: if the CC value between two services was 0.66 or more, this was regarded as evidence in favour of merging these services.	post: a CC value was considered to suggest that the decomposition of service A into services B and C had been beneficial for maintainability if the CC value between service B and C was 0.33 or less.
SC	pre: due to the lack of thresholds proposed in the literature to base the classification of SC values on, we can only reason about the alignment of the evolution of SC during a refactor and the evolution in maintainability experienced by the expert. In the case of a merge, one would expect the average SC to decrease as the number of entities to contribute to coupling decreases. An SC which is higher than the system average was regarded as evidence in favour of merging the involved services.	post: due to the lack of thresholds proposed in the literature to base the classification of SC values on, we can only reason about the alignment of the evolution of SC during a refactor and the evolution in maintainability experienced by the expert. In case of a decomposition, if the resulting services had an under-average SC, the refactor was regarded as beneficial for the maintainability of the system.
WSIC	pre: a WSIC was considered to contradict the suggestion of merging services A and B to be beneficial for maintainability if the WSIC of either service A or B fell within the lower 50% intervals as proposed by [Bogner et al., 2020], i.e., if the WSIC was higher than 15.	pre: considering the thresholds calculated by [Bogner et al., 2020], we regarded WSICs higher than 15 as supporting evidence for decomposing a service. post: services resulting from a decomposition were expected to have lower WSICs than their ancestor.
SIDC	pre: a SIDC value was considered to contradict the suggestion of merging services A and B to be beneficial for maintainability if the SIDC of either service A or B fell within the lower 50% intervals as proposed by [Bogner et al., 2020], i.e., if the SIDC was lower than 0.64.	pre: considering the thresholds calculated by [Bogner et al., 2020], we regarded SIDC values lower than 0.64 as supporting evidence for decomposing a service. post: services resulting from a decomposition were expected to have higher SIDCs than their ancestor.
LOC	pre: a LOC value was considered to contradict the suggestion of merging services A and B to be beneficial for maintainability if the LOC value of either service A or B was higher than the average LOC value of all services in the system.	pre: a LOC value of a service which was higher than the LOC of an average service in the system was considered as support for the decomposition of that service. post: services resulting from a decomposition were expected to have lower LOC values than their ancestor.

Table 6.1 continued from previous page

CF	pre: a CF was considered to contradict the suggestion of merging services A and B to be beneficial for maintainability if the CF of either service A or B was higher than the average CF of all services in the system.	pre: if the CF of a service was higher than the average CF of all services in the system, this was considered as support for the decomposition of that service. post: services resulting from a decomposition were expected to have lower CFs than their ancestor.
----	---	---

Table 6.1 provides an overview of the interpretive guidance by which we derived conclusions on maintainability from the metric values. Per metric, a summary is provided, outlining the relationship between its value and maintainability. This is done in the context of different refactor types. Hybrid refactors are not represented in this overview, as they essentially encompass a combination of a decomposition and a merge. The overview also specifies whether the pre-refactor value or post-refactor value of the metric is relevant. For each refactor type, we first formulated the expected assessment outcomes based on the expert’s observations, as discussed earlier in Chapter 4. Subsequently, guided by the overview in Table 6.1, we extracted concrete conclusions on maintainability from our assessment results. In the sequel, we discuss to which extent these conclusions are in line with the observations of the experts.

6.1.1 Merges

As none of the refactors that involved a merge was actually implemented, we are not able to reason about the effect of these refactors on maintainability. We can still discuss however the extent to which the results of our assessment were in line with the opinion of the expert who deemed a refactor to be beneficial for the maintainability of the application. In both of the planned refactors, R2.2 and R2.3, the involved services have a similar relation: the API is implemented in an additional, separate service, next to the service implementing the business logic. In R2.3 there is also a third service, which handles the communication with the API gateway of the system. Both service groups have been pointed out by the involved experts to be prominent candidates for merging, to improve the maintainability in a modular sense: the experts describe how in R2.2 the two services constantly exchange complex state events, as they have to be in the same state for every process instance, and how R2.3 is a textbook example of a bounded context divided over three services.

To assess the degree of alignment between our assessment and the expert’s observations, we began by formulating the expected assessment outcomes, guided by the overview earlier presented in Table 6.1. These expected outcomes are based on the expert’s observations, and form the most-right column in Table 6.2. Subsequently, we summarised the assessment results per metric for each refactor, which resulted in the second column of Table 6.2. This table offers a way to gain a rapid understanding of the alignment between the expert’s observations and the assessment outcomes.

TABLE 6.2: Relation between the assessment outcomes and the expert’s observations for the analysed merges.

Metric	Assessment observations	Expectations based on expert’s experiences
CC	Pre-R2.3: two of the services had a strong bidirectional change coupling, while the service pairs which included the third service had a lower change coupling which we classified as of regular strength. Pre-R2.2: a strong change coupling was observed for the service pair involved in this refactor.	Strong bidirectional change couplings were expected between the services.
SC	Pre-R2.3: SC could not be measured due to the event-driven nature of the system. Pre-R2.2: SC could not be measured due to the event-driven nature of the system.	Above-average SC values were expected between the services.
WSIC	Pre-R2.3: only S2.3.2 offered an interface, which had a WSIC of 2. Pre-R2.2: only S2.2.1 offered an interface, which had a WSIC of 24.	The WSIC of each service was expected to be lower than 15.
SIDC	Pre-R2.3: only S2.3.2 offered an interface, which had a SIDC of 0.5. Pre-R2.2: only S2.2.1 offered an interface, which had a SIDC of 0.0054.	The SIDC value of each service was expected to be higher than 0.64.
LOC	Pre-R2.3: each service had an under-average LOC value. Pre-R2.2: both services had a LOC value which was higher than the average of the system.	The LOC value of each service was expected to be under average.
CF	Pre-R2.3: each service had an under-average CF. Pre-R2.2: both services had a CF which was higher than the average of the system.	The CF of each service was expected to be under average.

While comparing the expected and actual outcomes, a few things stood out. In both refactors, the involved group of services contained a service pair with a strong bidirectional change coupling, implying that our assessment correctly identified these services as candidates for merging. As one of the services involved in R2.3 had a lower bidirectional coupling to the other two, this made it difficult to recognise this third service as an additional merge candidate. Its directed change coupling to the other two services was strong however, so using the directed change coupling we would be able to also identify this third service as a candidate for merging.

In R2.2, based on the size and cohesion measures one would not suggest a merge, as both

services were larger than an average service in the system and changed more often. In addition, S2.2.1 has a SIDC approaching zero, indicating a service with extremely low cohesion. As S2.2.1 serves as the API gateway for the entire application, however, this low cohesion is as expected and should not form the basis for any granularity decisions. Based solely on these facts the merge of R2.2 would not be recommended, as it would result in a service with an even larger size.

Our size and cohesion measures barely formed a counterargument for carrying out R2.3: each service had a LOC and CF which was lower than the average. The only metric contradicting a merge was the SIDC, which had only been measured for S2.3.2 as this was the only service offering an interface. The SIDC of this service implied that the interface was not cohesive, which is undesirable in this scenario, as a merge with another service will make the service only less cohesive. However, the small size of the interface (WSIC=2) and the fact that the service acts as an API for two other services undermine the relevance of the SIDC in this context.

As we look at the service sizes at a more granular level, however, at which we distinguish between lines of code of different programming languages and only consider lines of Java code, the services of R2.2 are not such outliers in size anymore. It is justifiable to some extent to solely consider Java lines of code, as Java is the only programming language among the top three code types that constitute the service size for both services in R2.2. The other main contributors, JSON and CSV, are rather data formats which are not used to implement any logic. When reasoning about service complexity as a sub-characteristic of maintainability, one could argue that lines of data contribute less to the complexity of a service than lines of code which actually represent the functionality of the service.

Overall, our assessment was able to identify the services which were pointed out by the experts as candidates for merging based on the change coupling values. As for the size and cohesion metrics, the assessment results did not oppose a merge in the case of R2.3. However, for R2.2, the size and cohesion metrics indicated that the involved services were relatively large and frequently altered compared to the rest of the system. This contradicts the suggestion of a merge to enhance maintainability.

6.1.2 Decompositions

In both the refactors entailing a decomposition (R1.1 and R2.4) we were able to perform our assessment both before and after the refactor. This allowed us to reason about the evolution of maintainability in reaction to the refactors.

To evaluate the alignment between our assessment and the expert's observations, we initially derived the expected assessment outcomes using the guidance provided in Table 6.1. These expected outcomes, shown in the rightmost column of Table 6.3, were formulated based on the expert's observations. Subsequently, we summarised the assessment results for each refactor per metric, resulting in the second column of Table 6.3.

TABLE 6.3: Relation between the assessment outcomes and the expert’s observations for the analysed decompositions.

Metric	Assessment observations	Expectations based on expert’s experiences
CC	<p>Post-R1.1: the CC value of the service pair resulting from the decomposition is 0.6, which we classified as of regular strength rather than weak.</p> <p>Post-R2.4: the CC values of the service pairs resulting from the decomposition could all be classified as weak.</p>	<p>Post-refactor, weak bidirectional change couplings were expected between the services.</p>
SC	<p>Post-R1.1: the SC between the resulting service pair was 0.75, but as we lacked averages due to this being the only service pair in the system, this value was not conclusive.</p> <p>Post-R2.4: SC could not be measured due to the event-driven nature of the system.</p>	<p>Post-refactor, under-average SC values were expected between the services.</p>
WSIC	<p>Pre-R1.1: the service had a WSIC of 4.</p> <p>Post-R1.1: only one of the services has an interface, also with a WSIC of 4.</p> <p>Pre- and post-R2.4: neither of the involved services offered an interface, so no WSIC could be determined.</p>	<p>Pre-refactor, a WSIC higher than 15 was considered an indication for decomposing.</p> <p>Post-refactor, the WSICs of the services resulting from the refactor were expected to be lower than the WSIC of the pre-refactor service.</p>
SIDC	<p>Pre-R1.1: the service had a SIDC of 0.5.</p> <p>Post-R1.1: only one of the services has an interface, also with a SIDC of 0.5.</p> <p>Pre- and post-R2.4: neither of the involved services offered an interface, so no SIDC could be determined.</p>	<p>Pre-refactor, a SIDC lower than 0.64 was considered an indication for decomposing.</p> <p>Post-refactor, the SIDC values of the services resulting from the refactor were expected to be higher than the SIDC of the pre-refactor service.</p>

Table 6.3 continued from previous page

LOC	<p>Pre-R1.1: the service had an under-average LOC.</p> <p>Post-R1.1: the LOC values of the services resulting from the decomposition were lower than the LOC of the pre-refactor service.</p> <p>Pre-R2.4: the service had a LOC value which was above average.</p> <p>Post-R2.4: the LOCs of the services resulting from the decomposition were lower than the LOC of the pre-refactor service, except for the LOC of S2.4.1, which is only lower when only considering lines of Java.</p>	<p>Pre-refactor, an above-average LOC was considered an indication for decomposing.</p> <p>Post-refactor, the LOC values of the services resulting from the refactor were expected to be lower than the LOC value of the pre-refactor service.</p>
CF	<p>Pre-R1.1: the service had an under-average CF.</p> <p>Post-R1.1: the CFs of the services resulting from the decomposition were lower than the CF of the pre-refactor service.</p> <p>Pre-R2.4: the service had a CF which was above average.</p> <p>Post-R2.4: the CFs of the services resulting from the decomposition were lower than the CF of the pre-refactor service, except for the CF of the source-docs-policy.</p>	<p>Pre-refactor, an above-average CF was considered an indication for decomposing.</p> <p>Post-refactor, the CFs of the services resulting from the refactor were expected to be lower than the CF of their ancestor.</p>

Our pre-refactor assessment did not identify the service decomposed in R1.1 to be a candidate for decomposition: both the CF and size of the service were lower than the average. A side note here is that this average is calculated over only two artefacts, making it vulnerable for extreme values, and less helpful in determining appropriate size and CF ranges for services in this system. The SIDC of 0.5 gave the only indication for decomposing. Regarding the service which was decomposed in R2.4, our assessment supported the decision of the expert to decompose the service: the CF and size were both more than 4 times above average.

Post-refactor, we observed a decrease in both CF and size for each of the two services resulting from R1.1, compared to the metric values of the pre-refactor service. This indication for an improved maintainability is in line with the experiences of the expert. Another observation was that the WSIC and SIDC were exactly the same for the pre-refactor service and one of the post-refactor services, which could indicate that the latter inherited the entire interface of the former.

The assessment of R2.4 showed a decrease in LOC and CF in the post-refactor services, which is an indication of more cohesive services. Two exceptions were S2.4.1, which only decreased in size when considering lines of Java, and S2.4.3, which CF was higher than that of S2.4.1 pre-refactor. These observations were discussed with the expert during the validation interviews, to gain a better understanding of their cause.

As a refactor should increase the modularity in order to increase maintainability, successful decompositions are expected to result in services which are not tightly coupled. Our assessment showed that there are no strong bidirectional change couplings post-refactor, which suggests that the modules which were extracted into separate services are not (strongly) dependent on each other, which is in line with the experiences of the experts of both cases.

The structural coupling observed between the service pair resulting from R1.1 (0,75) should not be interpreted as strong either, as we lack thresholds to justifiably reason about its implications: currently, there is a lack of empirical research that validates structural coupling intervals as indicators of maintainability.

Our assessment demonstrated a high degree of consistency with the maintainability trend observed by the experts during the refactors, whereas its ability to identify candidates for decomposition differed strongly per refactor. The inadequate identification of decomposition candidates in R1.1 may be attributed to the limited number of services in the system. This makes the average size and cohesion values susceptible to extremes, providing less reliable guidance in granularity decisions.

6.1.3 Hybrid Refactors

In the hybrid refactors, during which functionality was both extracted and merged simultaneously, certain services existed both pre- and post-refactor. This provided us with an opportunity to investigate the extent to which the individual metrics reflect the evolution of maintainability as perceived by the engineer involved in the refactor. This was not possible in the previously discussed merges and decompositions, as services never persisted throughout these refactors.

We first discuss R1.2 and R2.5 in parallel as they are considered comparable, in the sense that they both encompass a partial merge of two services. The corresponding assessment outcomes and expectations based on the expert's experiences can be found in Table 6.4. R2.1 is addressed separately due to its unique nature of transitioning from a choreography to an orchestration. This change in architectural style, which aimed at improving maintainability, is discussed individually since it is not directly comparable to the other refactors.

TABLE 6.4: Relation between the assessment outcomes and the expert’s observations for the analysed hybrid refactors.

Metric	Assessment observations		Expectations based on expert’s experiences
CC	Pre-R1.2: the two involved services exhibited strong bidirectional change couplings. Pre-R2.5: a strong change coupling was observed for the service pair involved in this refactor.	Post-R1.2: this measurement was not feasible as R1.2 was never implemented. Post-R2.5: strong change coupling values, higher than the change coupling of the pre-refactor service-pair, were observed between the services.	Pre-refactor, regular to strong bidirectional change couplings were expected between the services. Post-refactor, the bidirectional change couplings between the services were expected to decrease.
SC	Pre-R1.2: the SC of 0 of the service-pair is equal to the average SC, as the average is only based on this single service pair. Pre-R2.5: SC could not be measured due to the event-driven nature of the system.	Post-R1.2: - Post-R2.5: -	Pre-refactor, above-average SC values were expected between the services. . Post-refactor, the SC values between the services were expected to decrease.
WSIC	Pre-R1.2: the single service that did offer an interface, had a WSIC of 4. Pre-R2.5: only S2.5.1 offered an interface, which had a WSIC of 29. Post-R2.5: S2.5.1 still had a WSIC of 29, and the new service (S2.5.3) had a WSIC of 6.	Post-R1.2: - Post-R2.5: the S2.5.1 still had a WSIC of 29, and the new service (S2.5.3) had a WSIC of 6.	Pre-refactor, a WSIC higher than 15 was considered an indication for decomposing. Post-refactor, the WSIC of each service was expected to decrease.
SIDC	Pre-R1.2: the single service that did offer an interface, had a SIDC of 0,5. Pre-R2.5: only the S2.5.1 offered an interface, which had a SIDC of 0,2875.	Post-R1.2: - Post-R2.5: the S2.5.1 still had a SIDC of 0,2875, and the new service (S2.5.3) had a SIDC of 0,5.	Pre-refactor, a SIDC lower than 0,64 was considered an indication for decomposing. Post-refactor, the SIDC value of each service was expected to increase.

Table 6.4 continued from previous page

LOC	R1.2: one service had an under-average LOC and one an above-average LOC. Pre-R2.5: both services had a LOC value which was higher than the average of the system.	Post-R1.2: - Post-R2.5: only the LOC of S2.5.2 decreased, while the LOC of S2.5.1 increased. The new service (S2.5.3) had a LOC larger than each of the pre-refactor services.	Pre-refactor, an above-average LOC was considered an indication for decomposing. Post-refactor, the LOC value of each service was expected to decrease.
CF	Pre-R1.2: as the system-average is calculated over the two involved services, naturally one service had an under-average CF while the other one had a CF higher than the average. Pre-R2.5: both services had a CF which was higher than the average of the system.	Post-R1.2: - Post-R2.5: the CF of S2.5.2 decreased, while the CF of S2.5.1 increased. The new service (S2.5.3) exhibited a CF higher than each of the pre-refactor services.	Pre-refactor, an above-average CF was considered an indication for decomposing. Post-refactor, the LOC value of each service was expected to decrease.

The pre-refactor assessments of R1.2 and R2.5 each showed strong change coupling values between the pre-refactor services, which matched the experiences of the experts who described these services as tightly coupled. Structural coupling measurements could not provide additional support, as this could not be measured in the event-driven context of R1.2 and R2.5.

In addition to the change coupling values, we are also able to identify the service pair involved in R2.5 as candidates for extracting functionality based on size and cohesion-related metrics. Their LOC and CF values are significantly larger than the average service in the pre-refactor system. The high WSIC and low SIDC of the involved API-gateway service are additional indicators supporting the decomposing of this service, as these values imply a large interface with a low cohesiveness, both indicators for a lowly cohesive service. In the case of R1.2, we measured less prominent indications for extracting functionality. One service exhibited an above-average CF and LOC, while the other service scored below average in these metrics. This is partly due to the system-average, being based solely on the service pair involved in the refactor, which makes this average unreliable.

While we could not reason about the impact of R1.2 as this was never carried out, for R2.5 we observed that the three services resulting from R2.5 have change coupling values even stronger than the pre-refactor pair. The goal of the expert to reduce the coupling between the initial two services does not align with this observation. As the support for these post-refactor change coupling values is relatively low however, the extent to which this calculation reflects the actual change coupling can be questioned. The support is low as the change coupling was calculated over the short time interval which was available post-refactor. Calculating only over the interval just after the refactor can give biased results,

as the revisions which were made in this period could also be related to the aftermath of the refactor instead of maintenance activity. These observations were discussed during the validation interviews, which are described in Section 6.2.

We did not formulate specific expectations for the assessment results of R2.1. Due to the unique nature of this refactor, involving a large-scale switch in architectural style, we considered it more useful to observe and learn how our assessment would respond to such a change. A main observation was that the average change coupling between service pairs increased during the refactor. This implies that in the post-refactor architecture, the services are more dependent on each other than in the initial architecture. Intuitively this is correct, as an event-driven choreography allows a producer and consumer to operate independently, without being dependent on a central orchestrator. Services in such an architecture are by design already more independent than in an orchestration, as having an orchestrator in place creates extra interdependencies [Singhal et al., 2019]. The evolution in change coupling observed by our assessment aligns with theoretical expectations. The average service size decreased, which can be explained by the flow logic of each service being refactored into the orchestrator. As the interface-based metrics and SC could not be calculated for many of the involved services, these are less conclusive.

Due to the different natures of the hybrid refactors it is hard to draw any general conclusions, but we now summarise our main observations. For refactors involving a partial merge (R1.2 and R2.5), the assessment accurately identified tightly coupled services and supported the expert’s recommendations for partly merging the services. Our assessment is not able to distinguish candidates for a merge from candidates from a partial merge though, since this would require a finer-grained analysis in which internal components of the services are assessed. The assessment’s ability to identify the involved services as candidates for decomposition was weaker in the case of R1.2 due to the small number of services in the system, making average size and cohesion values less reliable references. Overall, the assessment demonstrated to be able to identify the candidates for a (partial) merge, and showed good alignment with the experts’ observations on the evolution of maintainability. Limitations were found in the identification of candidates for decomposition.

6.2 Validation Interviews

During the post-analysis interviews, we presented the results of our assessment to the case experts. Subsequently, we interviewed the experts on how they explained the results based on their extensive knowledge of the system under analysis. In this section, we discuss any additional information provided by the experts during these interviews. We discuss this on a case-by-case basis, discussing first the two refactors in Metadata, followed by the five refactors in the loan eligibility checker, and concluding with the Spinnaker project.

6.2.1 R1.1

Based solely on our pre-refactor assessment, the functionality that is provided as binary code, core, would be a more likely candidate for a decomposition than S1.1.1, as it has a slightly higher LOC value and CF. We inquired with the expert regarding the level of cohesion they attributed to the binary code and the reasons behind choosing to refactor the S1.1.1 instead of this binary code. The expert explained that the binary code is less cohesive than S1.1.1, which matches the observations from our assessment, but adds that as

this code is distributed as a jar to the end-user, and not as a highly-configurable application such as, for instance, S1.2.1, it was more convenient to let all binary code remain to be a single unit. After the refactor, we observed a strong directed coupling from S1.1.3 to S1.1.2, while vice-versa, the coupling degree was of regular strength. The expert acknowledged that S1.1.3 was indeed more dependent on S1.1.2: revisions to S1.1.3 are often related to adding or renaming a column in the database schema, and these changes are often required due to a change to S1.1.2. In addition, S1.1.2 evolves more independently as changes which do not involve the persistence layer and thus a change in the database schema, such as a renaming or internal refactor, does not influence S1.1.3 in any way. The expert also explains why the WSIC and SIDC values of S1.1.1 (pre-refactor) and S1.1.2 (post-refactor) are identical: S1.1.2 inherited the entire interface of S1.1.1. As S1.1.3 is only responsible for communicating with the database, this service did not require its own REST API.

6.2.2 R1.2

Regarding R1.2, we observed a strong change coupling between S1.2.2 and S1.2.1. We consulted the expert to determine if they were aware of this coupling. The expert emphasized that structurally, the two services are not related, since end-users typically use either S1.2.1 (REST) or S1.2.2 (GraphQL), but not both simultaneously. However, the expert recognised the change coupling. The expert aimed to offer a similar user experience in terms of functionality to every user of Metadata, regardless of whether they use the GraphQL or REST version. To achieve this goal, both services were maintained synchronously with regard to functionality. This explains the strong change coupling between these structurally unrelated services.

6.2.3 R2.1

One of the primary observations derived from our assessment was the increase in average change coupling of the system of 0,17 during the refactor. The expert suspects S2.1.9 to be the main cause for this. Per definition, an orchestrator introduces additional coupling between all services, and according to the expert that is an important cause of the increase in the average change coupling. Another observation was a decrease in average service size of 8,1%. The expert suspected this to be the flow logic of the individual services which was refactored into the orchestrator (S2.1.9) during R2.1.

6.2.4 R2.2

The services which are candidates for R2.2 (S2.2.1 and S2.2.2) are outliers in the system considering LOC and CF. Their extreme size is caused mainly by JSON code, which in S2.2.2 solely serves as stubs for integration tests. These services are larger compared to other sets of API gateways and orchestrators involved in different customer journeys, mainly because S2.2.1 and S2.2.2 enable seven distinct journeys instead of just one. This high number of dependent journeys is the reason why they are behind on maintenance: in other journeys, it was less complex to extract functionality from similar services over time. These extractions in other journeys have proven to be beneficial for maintainability, which forms an extra indication for the refactoring of S2.2.1 and S2.2.2 to be improvements as well regarding maintainability.

6.2.5 R2.3

Based on the high change coupling and relatively low CF and size of S2.3.1, S2.3.2 and S2.3.3, compared to the system averages, these services would all be candidates for merging, which is in line with the opinion of the expert. To gain more insight into what can be the cause of the low change frequencies, we asked the expert whether there was a concrete reason for the change frequencies of these services to be under average. The expert explained that although the functional scope of the services is not that elaborate, which explains the relatively low LOC values, the testing of the services is quite complex. This service group is responsible for the connection with the endpoints of the customer's bank in order to retrieve data following the guidelines from the PSD2 directive [Rijksoverheid, nd]. This directive prescribes a range of security standards which should be adhered to during data exchange. To enable proper testing of all the hashing procedures, elaborate sandboxes are required. The expert points out that the offer of PSD2 sandboxes is limited and that the sandboxes are often not representative. This forces the team to test any changes to this group of services directly in a production environment, which in its turn causes the team to modify these services the least possible. Another factor contributing to the relatively low CF is the integration of SaltEdge, a third-party module, which provides comprehensive functionality for handling integrations with other banks in adherence to the PSD2 directive [SaltEdge, nd]. The bank's policy aims to eventually transition all systems to exclusively use SaltEdge for such integrations. As the team was aware of the impending replacement of the PSD2 services by SaltEdge in the near future, actively maintaining the services received a low priority.

6.2.6 R2.4

Considering our assessment of R2.4, S2.4.1 was a high outlier in size with regard to the other services of the system. JSON code formed 81% of the total service size, so the service size did not map one-to-one with the functional scope of the service. During the interview, we asked the expert whether this JSON code was mainly static or also a main contributor to the high CF of the service. The expert explained that most of the JSON code resides in the test folder, and serves as stubs to enable integration testing of the service. The central position of S2.4.1 as an orchestrator requires a large number of stubs.

This also explains our observation that S2.4.1 slightly increased in size during the refactor, despite it being decomposed: the creation of two new services introduces additional JSON stubs that are added to S2.4.1. Apparently, the inclusion of these stubs generates more lines of code than the number of lines removed by extracting functionality into the two new services. As a result, the overall size of S2.4.1 slightly increased.

The decrease in functionality is only reflected when considering lines of Java code: during the refactor, this number decreased by 10,7%. The expert explained the high CF of S2.4.3 as a symptom of a new service: his expectation is that CF decreases over time.

6.2.7 R2.5

We asked the expert about the strong change coupling values which we observed post-refactor. The expert explained that these coupling values represent the actual coupling and are not artificial, as S2.5.3 is still dependent on the other two services when considering the workflow: the system supports multiple journeys, among others the revision journey. Although the workflow of this journey is handled by S2.5.3 since the refactor, the start of the journey is still handled by S2.5.2 and S2.5.1, creating a change coupling which the

expert expects to decrease over time. In S2.5.1 we noticed a slight increase in size during the refactor, which was counter-intuitive considering that functionality from this service was extracted into S2.5.3. The expert clarified that this increase is due to the fact that the original code in S2.5.1, which has become redundant, still needs to be removed from the service. Thus, the increase in size is a temporary effect caused by a delay in removing the now unnecessary code in S2.5.1. In addition, to ensure a smooth transition, the team implemented a switch in S2.5.1 which either instructed S2.5.3 or used the original internal code. The code related to this switch explains the small increase. This also explains why WSIC and SIDC are identical pre- and post-refactor, as the operations which are now implemented by S2.5.3 are simply not removed yet from the interface of S2.5.1. The service which was extracted during this refactor (S2.5.3) had the highest CF in the post-refactor period which was analysed. The expert described how in his experience newer services are often more prone to changes, and how he expects the CF to decrease when the service matures and becomes more stable.

6.2.8 Spinnaker

The assessment revealed that a varying range of services, among which S3.0.6, S3.0.9, S3.0.2, S3.0.7 and S3.0.10, have a directed change coupling of more than 0,86 to S3.0.3, S3.0.4 and S3.0.1, which we classify as strong. These three services also have a higher CF compared to the system average, which could be the cause of the high number of services showing a strong directed change coupling towards the three: as S3.0.3, S3.0.4 and S3.0.1 are modified very frequently, the chance is high that if a modification is made to another service, these three services were modified on this same day. This could possibly create false change couplings. We asked two experts who are actively contributing to Spinnaker about our observation and asked them whether they recognised the high directed coupling to these three services from their experiences, or whether they suspected most of it to be false couplings.

As Spinnaker is such a large system, contributors often are involved in the development of a number of services, but rarely in all services. The experts we were able to interview were familiar with S3.0.4, S3.0.2, S3.0.7, S3.0.3, S3.0.11, S3.0.5 and S3.0.6. One of the experts recognises the strong dependency of several services on S3.0.3, as S3.0.3 has a central position in the application: it is the orchestrator of the system and coordinates the other services. The expert remarks that he would expect S3.0.2 to also come forward as a service on which a high number of services depend, as S3.0.2 fulfils a central role in the system as the API gateway. Another observation from our assessment was the strong bidirectional change coupling between the three services (S3.0.1, S3.0.4 and S3.0.3). Furthermore, the three services each have a size which is above average in terms of lines of code. We inquired whether the experts would consider it to be an improvement to extract some artefacts out of (one of) these services, to increase their cohesiveness and ultimately prevent them from being maintainability bottlenecks. One of the experts gave a nuanced answer to this question: he explained that when considering refactors that cross service boundaries, first a thorough assessment is required to check whether the refactor would not result in other issues such as increased complexity or a decrease in performance. The other expert mentioned that an often-proposed refactor is to extract the artefact functionality out of S3.0.4 and create an artefact service. In the context of Spinnaker, an artefact is a JSON object which refers to an external resource, such as a Docker image or a file in GitHub. As this artefact functionality only constitutes 2% of the total size of S3.0.4, we asked the expert why this refactor would improve the maintainability of the system. The expert

explains that currently, an artefact has to be stored in multiple services, among others S3.0.4, S3.0.5 and S3.0.6, and it would be beneficial for the maintainability to centralise this functionality. A final remark of the experts is that the Spinnaker team is currently working on the transition from multiple repositories to one central mono-repository. They see the value of monitoring change coupling and want to create the possibility of calculating change coupling based on atomic commits instead of logical change sets.

6.3 Discussion

In this section, we discuss the implications of our assessment method, while covering its validity, reliability and generalizability. We focus on these properties specifically, as they determine the rigour of our research [Heale2015]. We also discuss the ability of our assessment method to identify services which are candidates for refactoring. Additionally, we reason about how our assessment method could be improved to be better equipped to serve as decision support for microservices granularity. Subsequently, research design choices are pointed out which may have influenced our results and we explain the limitations of our research.

6.3.1 Implications

Validity

The validity of the assessment, i.e., the extent to which our measurements correspond to the real world [Parveen and Showkat, 2017], is supported by several components of our research, amongst them the alignment of the assessment observations and the experiences of the experts, as discussed in Section 6.1. Considering the merge refactors, our assessment was in line with the opinions of the involved experts: the assessment enabled the identification of the services which were pointed out by the experts as candidates for merging. As for the analysed decompositions, we were able to measure the same evolution in maintainability as described by the experts: both the CF and size decreased for each service resulting from the decomposition, and the decompositions did not lead to services with high change coupling values. Finally, for the hybrid refactors, our assessment was able to identify the same refactor candidates as pointed out by the experts. Based on the assessment it was not possible to come to the same conclusions as the involved engineers did, as this would have required a finer-grained analysis: our assessment did not provide insight into the maintainability metrics of the different service components, which is necessary in order to recommend the extraction of an artefact into a separate service. The assessment did however indicate an evolution in maintainability similar to how this was experienced by the involved engineers.

Effect of temporal window

The tooling used to calculate change coupling values from the version control data (CodeMaat) allows for the specification of a temporal window. All revisions which were committed within this logical window are considered as one change set. This temporal window is specified in days, and decimals are not allowed. This implies that commits cannot be grouped based on an interval smaller than a day, while these groupings could be more accurate. Finding the most accurate temporal window is a research field on its own, but the effect of an increased temporal window is straightforward: a larger temporal window always results in change couplings equal or stronger than with a smaller temporal window. An example of how the change coupling between two services increases while enlarging the

temporal window can be seen in Table 6.5. We did not have validated approaches at our disposal to find the most accurate temporal window for each project but were still able to minimise biases by opting for the safest approach which yields the least false coupling, by selecting a temporal window of 1 day. While it is possible that this approach led to weaker couplings appearing less significant, thereby potentially undermining the perceived performance of our assessment, it ensures that we do not overestimate the performance of our assessment.

TABLE 6.5: Effect of different temporal windows on change coupling. Legend: CC represents the bidirectional change coupling, $CC_{A \rightarrow B}$ represents the directed coupling from artefact A to artefact B, $CC_{B \rightarrow A}$ represents the directed coupling from artefact B to artefact A and TW specifies the temporal window used for grouping revisions.

refactor	service A	service B	CC	shared-revisions	$CC_{A \rightarrow B}$	$CC_{B \rightarrow A}$	TW
pre-R2.2	S2.2.1	S2.2.2	67	258	0,82	0,58	1
pre-R2.2	S2.2.1	S2.2.2	81	344	0,91	0,74	2
pre-R2.2	S2.2.1	S2.2.2	85	376	0,95	0,78	3

Lack of revision context

In the change coupling analysis and CF calculations we performed as part of our assessment, the assumption was made that every revision included in the calculations of the metrics was maintenance-related. In literature, several types of maintenance activities are distinguished [Gomez, 2022], where *development* (i.e., the initial implementation of an application based on the requirements) is defined as a distinct phase in most software design life cycles [Kumar and Rashid, 2018]. The accidental inclusion of revisions part of this development phase in our analysis could have influenced the accuracy of our metrics negatively, as they do not represent maintenance activity and the commit behaviour of the developers might be different in the development phase compared to the maintenance phase. We expect this to have been of minimal impact since we explicitly consulted with the experts to determine the periods during which the refactors were implemented.

With the exception of R1.1, all the analysed refactors involved multiple Git repositories. To calculate change couplings between services involved in these refactors, revisions made within a temporal window of a single day were treated as part of the same change set. This grouping approach helps to identify logical change sets across repositories, but it does not eliminate the possibility of false change couplings. These false couplings occur when revisions that are completely independent of each other are grouped into the same change set. The presence of an issue tracking system that allows grouping revisions based on related issues, or the consistent mentioning of task IDs in commit messages, would have reduced the likelihood of creating artificial couplings. However, neither of these conditions were met in the selected projects.

As we were not able to tell to which maintenance task a revision was related, determining the optimal points in time within the system’s history to conduct the post-refactor assessment was challenging. The challenge lies in identifying when a refactor is considered fully implemented. In our cases, there were a few examples of functionality being extracted into a new service. As in industrial projects, downtime should often be minimised, such refactors entail a transition process. During this process, in the service to be decomposed a toggle

is implemented, which can be used to indicate whether the new service (implementing the extracted functionality) should be called or the internal version of this functionality should be used. Once the new service is completed, the toggle can be switched to exclusively use the new service. However, removing the duplicated old functionality from the decomposed service may not be considered a top priority. As a result, the team may begin working on other maintenance tasks, postponing the actual removal of the duplicate code until a later stage. This mainly influences the accuracy of the metrics that are not measured over a period but at a specific point in time, like SC, LOC, WSIC and SIDC. Using the commit messages as a reference, we made an effort to identify clean-up commits, which served as indicators for the end of the measured transition periods.

Developer behaviour

Another threat to the validity of our assessment is the potential variability in developer behaviour, more specifically in commit behaviour. Both change coupling and CF are metrics strongly influenced by this factor. One commit practice that can introduce bias is that of incomplete commits, where an engineer commits revisions that do not represent a complete atomic maintenance task, but only a fraction of it [Oliva and Gerosa, 2015]. In such cases, the entire maintenance task is typically completed through multiple commits. This results in an increased CF and lower change coupling values which, if these dependent commits are committed over multiple days, are lower than the actual change coupling.

Developer behaviour is a research field itself: several studies tried to analyse the commit behaviour of developers. One of those studies analysed commit intervals in four open-source projects, in an attempt to fit the intervals into statistical distributions [Ma et al., 2018, Oliva and Gerosa, 2015]. Their findings suggested that the majority of intervals between consecutive commits followed a power-law distribution, indicating that most of them were short, with only a few being notably long. The study also focused on the size of commits, which varies a lot and is further influenced by factors such as the project's development process, issue tracking systems, and the version control system being used. The use of a multi- or mono repository is also of influence here: developers make smaller commits in distributed repositories compared to centralised ones, and a change is split over multiple commits more frequently [Brindescu et al., 2014, Oliva and Gerosa, 2015].

The presence of a commit policy within an engineering team can significantly reduce the described biases, as they enforce more consistent commit behaviour among the different developers. Such a commit policy was in place in both Spinnaker and the loan eligibility checker, ensuring a more aligned committing approach. As Metadata was developed by an engineer working solo, we assumed a constant commit behaviour throughout the life-cycle of the project. We do not consider the influence of developer behaviour as a significant threat to the validity of our research, as our assessment primarily focuses on the impact of refactors on maintainability metrics. Under the assumption that developer behaviour before and after a refactor does not differ extremely, we expect the evolution of maintainability, as measured by our assessment, not to be substantially affected by it. However, we acknowledge that we cannot fully prevent developer behaviour and commit practices from still introducing some level of bias and variability in our assessment.

Reliability

In research, reliability refers to the degree of consistency in measurement [Gidron, 2013]. In this research, consistency was ensured by extensively describing the methodology used, covering the instrumentation used and other prerequisites and criteria, which has been discussed in Chapter 3. Rather than providing case-specific details such as the exact revisions involved in a refactor or the precise period over which a metric was calculated, we describe in detail our approaches to address the hurdles we encountered, as these challenges are not unique to our cases but can be encountered in general scenarios. An example is the exclusion of bot commits in the Spinnaker case: while we do not mention the specific commits which were excluded from the analysis, we explain what kind of revisions should be excluded and provide a potential approach in Chapter 4.

Generalizability

We discuss the generalizability of our research on two levels: the level of the analysed refactors and the level of the software projects in which these refactors were carried out. We analysed a total of seven refactors during our research, comprising two merges, two decompositions, and three hybrid refactors. These refactors each involved a range of two to seven microservices. The diversity of refactor types in our study provides the initial support for the generalizability of our research. Considering the software projects in which the refactors were analysed, we included three projects where each one represents a different type of project:

- Metadata is a small, solo-developer project consisting of two microservices.
- The loan eligibility checker is a large industrial project consisting of >17 microservices.
- Spinnaker is a large, open-source project consisting of >10 microservices.

By including projects of different sizes, implemented in different contexts (industrial and open-source), a wide range of projects is represented. This diversity is essential as we aimed to develop an assessment method with a broad applicability.

6.3.2 Prioritising refactoring candidates

Our current research objective is to enable a quantitative granularity assessment with regard to maintainability. The ultimate goal is to realise a tool which can serve as actual decision support for microservices granularity choices by identifying candidates for different types of refactors. Using both our pre-refactor assessments and expert input on why certain refactors would increase the maintainability, we were able to effectively evaluate the ability of our assessment to function as decision support.

Our assessment method exhibited a strong capability of identifying candidates for a merge. Services that the experts pointed out as candidates for merging were consistently recognized by our assessment through strong change coupling values. The assessment method's strength in recognising candidates for a decomposition varied, which we attribute to the system averages, crucial for the interpretation of size and cohesion metrics, being less reliable in systems with a small number of services.

In addition to identifying refactor candidates, useful decision support should also be able to prioritise them. Box plot analyses could be valuable in this scenario, as they enable the identification of outliers. In Figure 6.1 a box plot is depicted which was constructed using the change coupling values of the system of R2.3. The bottom and top lines of the box represent the start of the second and fourth quartile, respectively. In this example, the service pair which was pointed out as a candidate for merging by the expert is an outlier in terms of change coupling, represented by the dot. This suggests that our assessment method was able to prioritise the refactor of these services over those of other refactor candidates.



FIGURE 6.1: Boxplot calculated over the change coupling values pre-R2.3.

Finer-grained analysis

While our assessment method was able to identify candidates for a merge, the metrics which can support decompositions (size and cohesion metrics) offer less guidance, as our assessment was only performed on a service level. By not performing our assessment on a more fine-grained level, in which we calculated the metrics for individual modules within a service, we are not able to reason about the extent to which a module belongs in a service. At such a finer-grained level, change coupling could be useful to identify modules with extremely low change couplings, which could indicate that these modules are not cohesive. Next to performing the assessment on a module level, we could perform it on a file level as it could help to pinpoint the roots of change couplings or high change frequencies. Such fine-grained analyses can be of additional use, but only when interpreted by someone with in-depth knowledge of the analysed system. As this research focuses on granularity on a service level these finer-grained analyses were not performed.

6.3.3 Research limitations

Acquiring a data set

While in this research eventually three diverse projects were used as cases, the challenge of finding suitable cases is a limitation that needs to be addressed. A first remark is that there are only a few open-sourced microservice-based projects available. [Rahman et al., 2019] collected a data set of projects implementing an MSA, but most of these projects are sample projects, demonstrating a design pattern or the use of a framework. These projects are expected to not be under active maintenance and consequently are no proper representation of reality. This is a limitation which we expect to confine research to maintainability in MSAs in general. An additional limiting factor in our research was the requirement to have access to an expert who could provide essential information to enable the analysis of the refactors and in order to validate our findings. While [Rahman et al., 2019] does mention 11 industrial projects, the requirement of having contact with an expert involved in the project forced us to exclude most of them.

Interface-based metrics

As for the interface-based metric we measured (WSIC and SIDC) we identified two limitations:

1. Since not every service has an interface, these metrics are not always able to reflect the maintainability evolution in reaction to a refactor. Due to this limited applicability, we were often not able to calculate any meaningful system-wide averages, and as we do not have any other thresholds at our disposal, the metrics are not conclusive in most cases.
2. An interface does not map one-to-one to the functionality of a service, which sometimes caused a misalignment between our assessment and the expert's experience.

We encountered this second limitation in, for instance, S1.1.1 during R1.1: this service was decomposed into S1.1.3 and S1.1.2, and S1.1.2 inherited its entire interface. S1.1.3 implements functionality which had always been present in S1.1.1 but was not exposed in its interface. We observed a similar problem in Spinnaker and the loan eligibility checker: in both the event-driven architectures, some services do expose an API. As both systems follow the API gateway pattern however, the metrics calculated from an API specification are not always representative of the functionality implemented by the corresponding service. The API gateway pattern is a design pattern in which one service (the API gateway) serves as the single entry point to the system [Richardson, 2020], and routes incoming requests to the appropriate services. This pattern comes with a few advantages, one being that an API gateway hides the actual partitioning of the API into the different services from clients, removing the need for clients to interact with multiple APIs of individual microservices. This obstructs us from calculating the interface-based metrics for the individual services that do expose an API, as we can no longer relate the operations specified in the API specification to the corresponding service. The loan eligibility checker has several services adjacent to the API gateway that provide an API. However, these services, in turn, expose an API that represents the functionality of another group of services, so these API specifications do not allow us to calculate interface-based metrics corresponding to the functionality of a single service.

Measuring structural coupling

Structural coupling has proven to be a metric which is more complicated to measure than change coupling, as the ease of measuring and its accuracy are both strongly dependent on the available documentation on dependencies. The tool which was introduced in Chapter 3 (MicroDepGraph) calculates the structural coupling for each pair of services defined in the Docker compose file of the project under analysis. The availability of a Docker compose file was not an inclusion criterion during our case selection, to preserve generalizability, and consequently in both Spinnaker and the loan eligibility checker there was no such file. The absence of this Dockerfile in projects with an event-driven architecture is explainable, as in such architectures, per definition, services are agnostic of their producers and consumers. Manual calculation of the systems' structural coupling based on the available documentation on service dependencies was omitted as the engineers remarked that several dependencies were abstracted away in the documentation. Consequently, we were only able to measure the structural coupling for two refactors. This limited number of measured structural couplings reduces the robustness of our validation regarding the usefulness of structural coupling as an addition to our assessment method.

Chapter 7

Final remarks

In this chapter we give our final remarks by discussing related work, the conclusions we drew based on our research and the contributions this research makes, and finally by making recommendations for future research.

7.1 Related Work

In this section, we provide an overview of other relevant studies in the field that focused on quantitatively assessing microservice granularity. We discuss how our research relates to these works and highlight the contributions of our approach.

Inherent to the popularity of microservices is the high research activity in the field. Although we found only a few works which focused directly on assessing granularity in microservices, many papers describe approaches for decomposing monoliths, or to scope bounded contexts at design time in a green field application [Santos and Paula, 2021, Baresi et al., 2017, Li et al., 2019, Jin et al., 2018, Ahmadvand and Ibrahim, 2017]. Some of these papers only propose an approach for identifying microservice candidates, while others apply their approach to real software projects and provide a validation strategy which involves quantitatively assessing the quality of the resulting MSA. We consider the latter, despite their main focus being on identifying microservices candidates, as related work, as they do suggest evaluation strategies to assess the quality of a microservice architecture.

For instance, [Santos and Paula, 2021] applied the silhouette coefficient to identify microservices in a monolithic architecture, which is a quality measure related to the extent of cohesion and coupling. They define microservices as clusters and calculate the silhouette coefficient to assess cohesion within clusters and coupling between clusters. These notions of cohesion and coupling are based on the number of change sets in which A and B co-changed. While their approach was not validated specifically in the context of microservices, and they do not provide this validation themselves, it provides insights into assessing cohesion and coupling from version control data. As the focus of their research is decomposing a monolith however, their research does not provide insights on how to assess a green field MSA: their approach requires the version control data of a monolithic repository. Different in our work is the wide applicability to green field and brown field MSAs. Furthermore, we strongly focus on validation, and the use of a larger metric suite which considers more aspects than cohesion and coupling allows for a more comprehensive assessment of maintainability.

[Vural and Koyuncu, 2021] validated their decomposition approach by performing a quality assessment on the resulting granularity. Their assessment also focuses on the notions of cohesion and coupling, but measures these in a different way: in the paper, the metrics are not derived from the source code or version control data, but from the UML diagrams representing the microservice-based system. They measure the efferent and afferent coupling between the different microservices, and the relational cohesion within the microservices, concepts which were presented in [Atole and Kale, 2006]. These metrics were originally meant for object-oriented systems, and Vural et al. do not give any support for the validity of the metrics in an MSA context. In contrast to our work, their assessment approach has limited applicability as it is completely dependent on the availability of UML diagrams. The accuracy of the assessment in its turn depends on the completeness and correctness of the diagrams. As our assessment method only involves language-independent, automatically derivable metrics, we expect the applicability of our method to be significantly larger.

[Ntentos et al., 2020] proposes to assess the quality of MSAs by measuring the conformance of the architecture to a range of well-established design patterns and specifically coupling patterns, as these can affect the quality of an MSA strongly according to the authors. Ntentos et al. aimed to create a foundation for automatically assessing this conformance, but in order to calculate the different metrics the authors first manually analysed the code bases of the cases to derive representative model diagrams. Subsequently, the metrics were calculated based on these models. While their approach offers a foundation for assessing conformance, our research expands on this by introducing an approach with a maintainability focus which requires minimal manual intervention, increasing the feasibility of assessing large systems.

[Cardarelli et al., 2019] presents a method based on Model-Driven Engineering techniques, which provides insight into the evolution of different quality attributes of an MSA in reaction to architectural changes. Coincidentally, the paper uses the maintainability quality attribute as an example of an attribute to assess, but emphasises that their approach could be tailored for measuring other quality attributes. An obstacle to using their approach is that a model of the architecture is required: although there are techniques to automatically recover architectures from MSAs [Granchelli et al., 2017a, Granchelli et al., 2017b, Alshuqayran et al., 2018], these techniques require specific input data which is not always available. As an example, the recovery technique used in their case study requires Dockerfiles, and if those are not available the alternative of deriving the model manually can be challenging. In contrast, our work focuses primarily on maintainability and uses metrics automatically derivable from the software and version control data.

The research of Apolinario et al. focuses specifically on monitoring the evolution of maintainability in an MSA [Apolinário and de França, 2021]. They developed a method called SYMBIOTE, which collects coupling metrics at runtime, and measured their metrics in every release in the history of Spinnaker, which also was their (only) case. Although their results are promising, as the evolution in metrics seems to correspond with the architectural evolution of the system, their method does not assess cohesion and requires a test suite covering the complete system, which decreases the accessibility of their method as such a test suite is not always at hand. Our research addresses these limitations by introducing metrics that capture cohesion as well as coupling and size, providing a different perspective for evaluating maintainability in relation to granularity.

We conclude that the existing work on microservices granularity assessments and measurement of quality with regard to maintainability mainly focuses on coupling metrics. These coupling metrics can be derived either statically or dynamically, but do not mine version control data, which to our knowledge is a novel way of assessing the maintainability of MSAs. An exception here is the work of [Santos and Paula, 2021], but their approach is only applicable to brownfield MSAs for which the version control data of the monolithic architecture is still available. Aside from the coupling-based metrics, [Bogner et al., 2020] is one of the few works that propose an approach to assess maintainability in MSAs based on cohesion metrics, which is why we used their tool (RAMA-CLI) in our work. The assessment methods presented in the related works often require assets which can only be obtained with a substantial effort, such as very accurate UML diagrams or elaborate test suites. In contrast, one of the contributions of our research is to provide an assessment method which can be applied with minimal manual intervention.

In summary, while there are related works that address aspects of granularity assessment in microservices, our research provides a unique contribution by introducing an assessment method focused on measuring maintainability. Our research complements existing works by offering a different methodology, using a larger metric suite, which enhances the understanding and evaluation of microservice architecture quality.

7.2 Conclusion

In this section, we revisit our initial research questions and provide answers based on our research findings. Subsequently, we summarise the contributions of this research to both science and the industry.

1. How can granularity be assessed from a maintainability perspective?

(a) Which metrics enable a quantitative assessment of maintainability in MSAs?

(b) How can these metrics be derived from existing projects?

To assess granularity from a maintainability perspective, we constructed a quantitative method to assess the quality of a microservice granularity. This assessment method focused on maintainability, as this system property is crucial for the long-term success of a software project. We selected a set of six maintainability metrics tailored for microservice-based systems: change coupling (CC), structural coupling (SC), lines of code (LOC), weighted service interface count (WSIC), service interface data cohesion (SIDC) and change frequency (CF). These metrics were derived automatically, by utilising a set of tools which could be combined into a single assessment tool. We learned that it is not possible to measure our complete set of metrics in every type of project. For instance, measuring the SC using the MicroDepGraph tool required Dockerfiles in which the dependencies between services were explicitly stated. In event-driven architectures, services are agnostic of their producers and consumers, and the SC could not be determined. Similarly, WSIC and SIDC, both interface-based metrics, have limitations in their applicability. These metrics can only be derived from RESTful APIs, which are always available. Additionally, in the case of the API gateway pattern, where a single service represents a set of other services through its API, these metrics can be influenced in an inaccurate manner. Hence, the calculation of WSIC and SIDC may not always be feasible or may provide misleading results in certain scenarios. Nevertheless, the complementary nature of our metric set ensured that the unavailability of certain metric calculations did not undermine the usefulness of

our assessment method. We applied our method to the selected projects, both before and after refactors, to evaluate the impact of granularity on maintainability.

2. How does this assessment method obtained from our results relate to the intuitive understanding of the experts?

(a) In what context can the performance of our assessment method be compared to the intuitive understanding of the experts?

In order to compare the performance of our assessment method with the observations of the experts, we selected three microservice-based projects that met our criteria: the source code and version control data of the projects were accessible, the systems addressed real-life business needs and were not sample projects which demonstrated a certain design pattern for instance, at least one refactor had been carried out which affected the granularity of the system, and we were able to contact an involved expert about the intentions behind the refactor and its aftermath regarding the maintainability of the system. We applied our assessment method to each of the selected projects and subsequently compared the results to the observations of the experts who were involved in the selected projects. Beforehand, we conducted interviews with the experts to gather their insights on the refactors and how they experienced the refactors to have impacted maintainability. We found out that our assessments aligned with the experts' experiences in most cases, indicating a correlation between our quantitative assessment method and their intuitive understanding. However, there were some exceptions, particularly when our metrics were measured over a short interval of time or compared to system averages based on a small number of services. These factors disturbed the alignment between our method and the experts' observations and should be taken into account as criteria when evaluating the usability of our assessment method in future studies.

How can the quality of the granularity of a microservice architecture be improved with regards to the application's maintainability?

Based on our research findings, the quality of the granularity of a microservice architecture can be improved with respect to maintainability by applying our assessment method to the system. Our assessment method showed to be of different strength when identifying a merge beneficial for maintainability compared to a decomposition but nevertheless proved to be useful in both cases. As for the merge refactors, we observed strong change coupling values between the pre-refactor service pairs which were all above average, implying that our assessment method was able to recognise these services as candidates for merging. In the case of decompositions, our pre-refactor assessment was only able to identify one of the two refactors as a candidate for decomposing. The service which was not identified was part of a project with a small number of services, which we assume to be of influence on the performance of our assessment method: when identifying candidates for decomposition, the method relies on cohesion and size metrics, which are interpreted in relation to the system averages. However, in smaller systems, these averages may be less reliable. Our post-refactor assessments were able to recognise the refactors as beneficial for maintainability, as we did not observe any strong change coupling values between the services resulting from the decompositions. In the three hybrid refactors we analyzed (combinations of merges and decompositions), our assessment method was able to recognise the same trend in maintainability as experienced by the experts. We observed strong pre-refactor change coupling values, between the services about to be partially merged. These services also

had a higher LOC and change frequency than an average service in the system, advocating for the extraction of functionality. Our method was not able to consistently distinguish candidates for a complete merge from candidates from a partial merge, as this would require a finer-grained analysis, on the service-component level rather than the service-level. Overall, we found our the results of applying our assessment method to align well with the experiences of the experts, providing valuable insights into improving maintainability through appropriate granularity decisions. However, some exceptions were observed when measuring over a short time interval or comparing metrics to system averages based on a small number of services. Therefore, these and other context-specific factors need to be considered when applying our assessment method to enhance the maintainability of a microservice architecture.

7.2.1 Contributions to Research

Several contributions are made by this research to the research fields of microservice granularity and maintainability in MSAs:

1. A quantitative assessment method for microservice granularity: the quantitative assessment method developed during this research provides a systematic approach to evaluate the quality of the granularity level in microservice architectures with respect to maintainability. This method constitutes a basis for future decision support to help determine appropriate granularity levels.
2. Integration of multiple metrics: by integrating multiple metrics, change coupling, structural coupling, LOC, WSIC, SIDC and change frequency, the assessment method provides a complementary evaluation of microservice granularity. The multi-dimensional approach increases the reliability of the assessment method.
3. Identification of refactor candidates: the assessment method demonstrated to be proficient in identifying refactor candidates, specifically in recognizing services as candidates for merging. The method's ability to accurately identify candidates for merging based on change coupling values can serve as an inspiration for future work aiming to provide objective decision support in microservice granularity decisions. The results and methodology presented in this report can serve as a basis for such future work, enabling researchers to build upon this work and increase the potential of microservice architecture by developing automated and objective decision-making tools, which can assess microservice architectures and give recommendations to improve their granularity.

7.2.2 Contributions to Industry

This research also offers valuable contributions and insights to the industry, especially in the context of microservice architectures:

1. Decision support for refactoring: our assessment method provides decision support which helps to identify refactor candidates in microservice-based applications. By identifying and to a certain extent also prioritizing refactor candidates, the assessment method can guide software teams in making informed decisions about the granularity level of their systems, ultimately improving maintainability.

2. Validation strategy: this research presented a methodology for validating metric-based results through expert input in the context of real-world refactors. This methodology increases the applicability of the assessment method in industrial software development settings.

7.3 Future Work

As mentioned earlier, one of the threats to the validity of our research is the lack of empirical evidence showing that grouping revisions based on a temporal window reflects the actual change sets to an appropriate extent. Research to empirically validate this correlation would be valuable, as many microservice-based projects are implemented in a multi-repository style, which does not allow for a change coupling analysis on the level of commits.

To circumvent the use of temporal windows for the grouping of change sets, a promising approach would be to make use of Integrated Development Environments (IDEs) that are well-instrumented for monitoring change evolution. The Eclipse plug-in proposed by Negara et al. is an example of such instrumentation, and it allows to collect finer-grained data than solely the data saved by the version control system, which can be summarized as the changes to files and their corresponding metadata [Negara et al., 2012]. Their plug-in is able to measure other data related to the evolution of an application, such as the invocation of tests. This could form another basis for grouping commits into change sets, and the accuracy of the resulting change coupling should be investigated.

Another research direction that is quite critical to enable the assessment of the granularity of microservice architectures, is the investigation of approaches to automatically identify dependencies between microservices in existing systems. The existing techniques for deriving dependencies, such as the MicroDepGraph tool which considers Docker dependencies, are helpful but have a low applicability as they are not suited for event-driven architectures, for instance, as in such architectures services are often agnostic of their consumers and producers. Furthermore, the detection of dependencies at the level of Docker dependencies abstracts away the information relevant to assign weights to the coupling relations so that it is not possible to distinguish a service pair between which only one small transaction takes place from time-to-time, from a pair which constantly exchanges a high volume of data. This limitation highlights the need for more fine-grained approaches, which can provide a deeper understanding of the dependencies between microservices. A potential approach would be to leverage machine learning techniques in order to derive more detailed and accurate dependency information from existing systems. This requires models that can be used to analyse transaction types, data volumes, and other relevant factors to assign weights to the coupling relations between microservices.

Finally, for our assessment method to ultimately serve as a decision support tool for microservice granularity, our main recommendation for future research is to empirically validate our findings on larger data sets. Given that a larger data set of refactors could be acquired, it would be interesting to categorise the experiences of experts systematically, which would enable the application of statistical tests to investigate the correlation between the results of applying our assessment method and the system maintainability quantitatively.

Bibliography

- [Ahmadvand and Ibrahim, 2017] Ahmadvand, M. and Ibrahim, A. (2017). Requirements reconciliation for scalable and secure microservice (de)composition. pages 68–73. Institute of Electrical and Electronics Engineers Inc.
- [Al-Debagy and Martinek, 2021] Al-Debagy, O. and Martinek, P. (2021). Microservices identification methods and quality metrics.
- [Alshuqayran et al., 2018] Alshuqayran, N., Ali, N., and Evans, R. (2018). Towards micro service architecture recovery: An empirical study. *Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018*, pages 47–56.
- [Apolinário and de França, 2021] Apolinário, D. R. and de França, B. B. (2021). A method for monitoring the coupling evolution of microservice-based architectures. *Journal of the Brazilian Computer Society*, 27:1–35.
- [Athanasopoulos et al., 2015] Athanasopoulos, D., Zarras, A. V., Miskos, G., Issarny, V., and Vassiliadis, P. (2015). Cohesion-driven decomposition of service interfaces without access to source code. *IEEE Transactions on Services Computing*, 8:550–5532.
- [Atlassian, nd] Atlassian (n.d.). Git merge conflicts.
- [Atole and Kale, 2006] Atole, C. S. and Kale, K. V. (2006). Assessment of package cohesion and coupling principles for predicting the quality of object oriented design. *2006 1st International Conference on Digital Information Management, ICDIM*, pages 1–5.
- [Auslander, 2017a] Auslander, D. (2017a). Getting the balance right in microservices development.
- [Auslander, 2017b] Auslander, D. (2017b). Getting the balance right in microservices development.
- [Awati and Wigmore, 2022] Awati, R. and Wigmore, I. (2022). What is monolithic architecture in software?
- [Ball et al., 1997] Ball, T., Kim, J. M., Porter, A., and Siy, H. P. (1997). If your version control system could talk.
- [Baresi et al., 2017] Baresi, L., Garriga, M., and Renzis, A. D. (2017). Microservices identification through interface analysis. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10465 LNCS:19–33.

- [Beckman et al., 2021] Beckman, M. D., Çetinkaya Rundel, M., Horton, N. J., Rundel, C. W., Sullivan, A. J., and Tackett, M. (2021). Implementing version control with git and github as a learning objective in statistics and data science courses. *Journal of Statistics and Data Science Education*, 29:S132–S144.
- [Bogner et al., 2019] Bogner, J., Fritzsich, J., Wagner, S., and Zimmermann, A. (2019). Assuring the evolvability of microservices: Insights into industry practices and challenges. *Proceedings - 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*, pages 546–556.
- [Bogner et al., 2017a] Bogner, J., Wagner, S., and Zimmermann, A. (2017a). Automatically measuring the maintainability of service- and microservice-based systems - a literature review. volume Part F131936, pages 107–115. Association for Computing Machinery.
- [Bogner et al., 2017b] Bogner, J., Wagner, S., and Zimmermann, A. (2017b). Towards a practical maintainability quality model for serviceand microservice-based systems. *ACM International Conference Proceeding Series*, Part F130530:195–198.
- [Bogner et al., 2020] Bogner, J., Wagner, S., and Zimmermann, A. (2020). Collecting service-based maintainability metrics from restful api descriptions: Static analysis and threshold derivation. volume 1269 CCIS, pages 215–227. Springer Science and Business Media Deutschland GmbH.
- [Brindescu et al., 2014] Brindescu, C., Codoban, M., Shmarkatiuk, S., and Dig, D. (2014). How do centralized and distributed version control systems impact software changes?
- [Brown, 2020a] Brown, K. G. (2020a). What’s the right size for a microservice?
- [Brown, 2020b] Brown, K. G. (2020b). What’s the right size for a microservice?
- [Cardarelli et al., 2019] Cardarelli, M., Salle, A. D., Iovino, L., Malavolta, I., Francesco, P. D., and Lago, P. (2019). An extensible data-driven approach for evaluating the quality of microservice architectures. *Proceedings of the ACM Symposium on Applied Computing*, Part F147772:1225–1234.
- [Cataldo et al., 2009] Cataldo, M., Mockus, A., Roberts, J. A., and Herbsleb, J. D. (2009). Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35:864–878.
- [Chidamber and Kemerer, 1994] Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20:476–493.
- [Cojocararu et al., 2019] Cojocararu, M. D., Oprescu, A., and Uta, A. (2019). Attributes assessing the quality of microservices automatically decomposed from monolithic applications. *Proceedings - 2019 18th International Symposium on Parallel and Distributed Computing, ISPDC 2019*, pages 84–93.
- [D’Ambros et al., 2009] D’Ambros, M., Lanza, M., and Robbes, R. (2009). On the relationship between change coupling and software defects. *Proceedings - Working Conference on Reverse Engineering, WCRE*, pages 135–144.
- [Dhanushka, 2021] Dhanushka, D. (2021). Event-driven apis — understanding the principles.

- [Docker, nd] Docker (n.d.). Compose file version 3 reference.
- [Engelbertink and Vogt, 2014] Engelbertink, F. and Vogt, H. (2014). How to save on software maintenance costs: an omnext white paper on software quality.
- [Erl, 2005] Erl, T. (2005). *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, USA.
- [Erl, 2017] Erl, T. (2017). *Service-Oriented Architecture: Analysis and Design for Services and Microservices*.
- [Ferguson, 2017] Ferguson, P. (2017). What size should microservices be?
- [Fluri et al., 2005] Fluri, B., Gall, H. C., and Pinzger, M. (2005). Fine-grained analysis of change couplings. *Proceedings - Fifth IEEE International Workshop on Source Code Analysis and Manipulation, SCAM 2005*, pages 66–74.
- [Fritzsich et al., 2019] Fritzsich, J., Bogner, J., Wagner, S., and Zimmermann, A. (2019). Microservices migration in industry: Intentions, strategies, and challenges. pages 481–490. Institute of Electrical and Electronics Engineers Inc.
- [Gall et al., 2003] Gall, H., Jazayeri, M., and Krajewski, J. (2003). Cvs release history data for detecting logical couplings. *International Workshop on Principles of Software Evolution (IWPSE)*, 2003-January:13–23.
- [Ghofrani and Lübke, 2018] Ghofrani, J. and Lübke, D. (2018). Challenges of microservices architecture: A survey on the state of the practice.
- [Gidron, 2013] Gidron, Y. (2013). Reliability and validity.
- [Glöckner et al., 2016] Glöckner, M., Ludwig, A., and Franczyk, B. (2016). How low should you go? - conceptualization of the service granularity framework. In *European Conference on Information Systems*.
- [GNU, nd] GNU (n.d.). Cvs - open source version control.
- [Goebelbecker, 2022] Goebelbecker, E. (2022). Java vs python: Code examples and comparison.
- [Gomez, 2022] Gomez, J. (2022). The four types of software maintenance.
- [Granchelli et al., 2017a] Granchelli, G., Cardarelli, M., Francesco, P. D., Malavolta, I., Iovino, L., and Salle, A. D. (2017a). Microart: A software architecture recovery tool for maintaining microservice-based systems. *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings*, pages 298–302.
- [Granchelli et al., 2017b] Granchelli, G., Cardarelli, M., Francesco, P. D., Malavolta, I., Iovino, L., and Salle, A. D. (2017b). Towards recovering the software architecture of microservice-based systems. *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings*, pages 46–53.
- [Halstead, 1977] Halstead, M. H. (1977). *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., USA.

- [Harsh, 2022] Harsh, K. (2022). Soa vs. microservices: A head-to-head comparison | scout apm blog.
- [Hassan and Holt, 2004] Hassan, A. E. and Holt, R. C. (2004). Predicting change propagation in software systems.
- [Hassan and Holt, 2006] Hassan, A. E. and Holt, R. C. (2006). Replaying development history to assess the effectiveness of change propagation tools. *Empirical Software Engineering*, 11:335–367.
- [Hassan et al., 2020] Hassan, S., Bahsoon, R., and Kazman, R. (2020). Microservice transition and its granularity problem: A systematic mapping study. *Software - Practice and Experience*, 50:1651–1681.
- [Heitlager et al., 2007] Heitlager, I., Kuipers, T., Visser, J., Heitlager, I., Kuipers, T., and Visser, J. (2007). Software evolution open universiteit 20 artikel 2 a practical model for measuring maintainability artikel 2 a practical model for measuring maintainability a practical model for measuring maintainability-a preliminary report.
- [Hirzalla et al., 2009] Hirzalla, M., Cleland-Huang, J., and Arsanjani, A. (2009). A metrics suite for evaluating flexibility and complexity in service oriented architectures. volume 5472 LNCS, pages 41–52.
- [Homay et al., 2020] Homay, A., de Sousa, M., Zoitl, A., and Wollschlaeger, M. (2020). Service granularity in industrial automation and control systems. volume 1, pages 132–139.
- [Homay et al., 2019] Homay, A., Zoitl, A., de Sousa, M., Wollschlaeger, M., and Chrysoulas, C. (2019). Granularity cost analysis for function block as a service. volume 1, pages 1199–1204.
- [IBM, 2020] IBM (2020). Event storming.
- [IBM, 2021a] IBM (2021a). Microservices in the enterprise, 2021: Real benefits, worth the challenges.
- [IBM, 2021b] IBM (2021b). Soa vs. microservices: What’s the difference? | ibm.
- [IBM, nd] IBM (n.d.). Event-driven architecture.
- [Interquartile range, nd] Interquartile range (n.d.). Median and interquartile range-nonparametric univariate statistics for quantitative variables.
- [ISO25000, nd] ISO25000 (n.d.). Iso25000.
- [Jain et al., 2021] Jain, G., Thakar, U., Tewari, V., and Varma, S. (2021). A survey on trending topics of microservices. *International Journal of Emerging Trends in Engineering Research*, 9:1091.
- [Jin et al., 2018] Jin, W., Liu, T., Zheng, Q., Cui, D., and Cai, Y. (2018). Functionality-oriented microservice extraction based on execution trace clustering. pages 211–218. Institute of Electrical and Electronics Engineers Inc.
- [Khoshnevis, 2023] Khoshnevis, S. (2023). A search-based identification of variable microservices for enterprise saas. *Frontiers of Computer Science*, 17.

- [Knoche and Hasselbring, 2019] Knoche, H. and Hasselbring, W. (2019). Drivers and barriers for microservice adoption-a survey among professionals in germany 1 drivers and barriers for microservice adoption-a survey among professionals in germany. 14.
- [Krishna, 2021] Krishna, H. (2021). 5 microservices examples: Amazon, netflix, uber, spotify etsy.
- [Kulkarni and Dwivedi, 2008] Kulkarni, N. and Dwivedi, V. (2008). The role of service granularity in a successful soa realization - a case study. volume PART 1, pages 423–430.
- [Kumar and Rashid, 2018] Kumar, M. and Rashid, E. (2018). An efficient software development life cycle model for developing software project. *International Journal of Education and Management Engineering*, 8:59–68.
- [Laskowski, 2019] Laskowski, D. (2019). Moving to microservices: How granular should my services be? - technology insights blog.
- [Lewis and Fowler, 2014] Lewis, J. and Fowler, M. (2014). Microservices.
- [Li et al., 2019] Li, S., Zhang, H., Jia, Z., Li, Z., Zhang, C., Li, J., Gao, Q., Ge, J., and Shan, Z. (2019). A dataflow-driven approach to identifying microservices from monolithic applications. *Journal of Systems and Software*, 157.
- [Li et al., 2020] Li, Y., Wang, C. Z., Li, Y. C., Su, J., and Chen, C. H. (2020). Granularity decision of microservice splitting in view of maintainability and its innovation effect in government data sharing. *Discrete Dynamics in Nature and Society*, 2020.
- [Lin et al., 2013] Lin, S., Ma, Y., and Chen, J. (2013). Empirical evidence on developer’s commit activity for open-source software projects. In *International Conference on Software Engineering and Knowledge Engineering*.
- [Lindvall et al., 2003] Lindvall, M., Tvedt, R. T., and Costa, P. (2003). An empirically-based process for software architecture evaluation. *Empirical Software Engineering*, 8:83–108.
- [Ma et al., 2018] Ma, S. P., Fan, C. Y., Chuang, Y., Lee, W. T., Lee, S. J., and Hsueh, N. L. (2018). Using service dependency graph to analyze and test microservices. volume 2, pages 81–86. IEEE Computer Society.
- [Ma et al., 2013] Ma, Y., Wu, Y., and Xu, Y. (2013). Dynamics of open-source software developer’s commit behavior: An empirical investigation of subversion.
- [Mathews, 2021] Mathews, S. (2021). How many repositories do you need for a microservices project?
- [McCabe, 1976] McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.
- [Mella et al., 2019] Mella, F. P., Márquez, G., and Astudillo, H. (2019). Migrating from monolithic architecture to microservices: A rapid review.
- [Mergify, nd] Mergify (n.d.). Mergify - ci/cd pipeline optimizer.

- [Munialo et al., 2019] Munialo, S. W., Muketha, G. M., and Omieno, K. K. (2019). Size metrics for service-oriented architecture. *International Journal of Software Engineering Applications (IJSEA)*, 10.
- [Negara et al., 2012] Negara, S., Vakilian, M., Chen, N., Johnson, R. E., and Dig, D. (2012). Is it dangerous to use version control histories to study source code evolution? *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7313 LNCS:79–103.
- [Netflix, 2015] Netflix (2015). Global continuous delivery with spinnaker.
- [Newman, 2021] Newman, S. (2021). *Building microservices : designing fine-grained systems*.
- [Ntentos et al., 2020] Ntentos, E., Zdun, U., Plakidas, K., Meixner, S., and Geiger, S. (2020). Assessing architecture conformance to coupling-related patterns and practices in microservices. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 12292 LNCS:3–20.
- [Oliva and Gerosa, 2011] Oliva, G. A. and Gerosa, M. A. (2011). On the interplay between structural and logical dependencies in open-source software. pages 144–153.
- [Oliva and Gerosa, 2015] Oliva, G. A. and Gerosa, M. A. (2015). Change coupling between software artifacts: Learning from past changes. *The Art and Science of Analyzing Software Data*, pages 285–323.
- [Panichella et al., 2021] Panichella, S., Rahman, M. I., and Taibi, D. (2021). Structural coupling for microservices. *International Conference on Cloud Computing and Services Science, CLOSER - Proceedings, 2021-April*:280–287.
- [Parveen and Showkat, 2017] Parveen, H. and Showkat, N. (2017). Review view project mass communication theory view project.
- [Perepletchikov and Ryan, 2011] Perepletchikov, M. and Ryan, C. (2011). A controlled experiment for evaluating the impact of coupling on the maintainability of service-oriented software. *IEEE Transactions on Software Engineering*, 37:449–465.
- [Perepletchikov et al., 2007] Perepletchikov, M., Ryan, C., Frampton, K., and Tari, Z. (2007). Coupling metrics for predicting maintainability in service-oriented designs. *Proceedings of the Australian Software Engineering Conference, ASWEC*, pages 329–338.
- [Rahman et al., 2019] Rahman, M. I., Panichella, S., and Taibi, D. (2019). A curated dataset of microservices-based systems. *CoRR*, abs/1909.03249.
- [Rahman and Taibi, nd] Rahman, M. I. and Taibi, D. (n.d.). Microdegraph.
- [Rama-Cli, nd] Rama-Cli (n.d.). rama-cli: Restful api metric analyzer cli.
- [Richardson, 2020] Richardson, C. (2020). Decompose by business capability.
- [Rijksoverheid, nd] Rijksoverheid (n.d.). Payment service directive 2 (psd2).
- [RubyGarage, 2019] RubyGarage (2019). Monolith vs soa vs microservices vs serverless architecture.
- [SaltEdge, nd] SaltEdge (n.d.). Salt edge | open banking for every business.

- [Santos and Paula, 2021] Santos, A. and Paula, H. (2021). Microservice decomposition and evaluation using dependency graph and silhouette coefficient. *ACM International Conference Proceeding Series*, pages 51–60.
- [Sellami et al., 2022] Sellami, K., Ouni, A., Saied, M. A., Bouktif, S., and Mkaouer, M. W. (2022). Improving microservices extraction using evolutionary search. *Information and Software Technology*, 151.
- [Seroukhov, 2020] Seroukhov, S. (2020). What’s the right size for a microservice?
- [Shadija et al., 2017] Shadija, D., Rezai, M., and Hill, R. (2017). Microservices: Granularity vs. performance. pages 215–220. Association for Computing Machinery, Inc.
- [Siket et al., 2014] Siket, I., Beszédés, Á., and Taylor, J. (2014). Differences in the definition and calculation of the loc metric in free tools.
- [Silva et al., 2014] Silva, L. L., Valente, M. T., and Maia, M. D. A. (2014). Assessing modularity using co-change clusters. *MODULARITY 2014 - Proceedings of the 13th International Conference on Modularity (Formerly AOSD)*, pages 49–60.
- [Singh and Huhns, 2005] Singh, M. P. M. P. and Huhns, M. N. (2005). Service-oriented computing : semantics, processes, agents. page 549.
- [Singhal et al., 2019] Singhal, N., Sakthivel, U., and Raj, P. (2019). Selection mechanism of micro-services orchestration vs. choreography. *International Journal of Web Semantic Technology (IJWesT)*, 10.
- [Spinnaker, a] Spinnaker. Spinnaker architecture overview.
- [Spinnaker, b] Spinnaker. Spinnaker bot: A github bot for managing spinnaker’s repos.
- [Spinnaker, c] Spinnaker. Spinnaker project.
- [Statista, 2022] Statista (2022). Microservices adoption level worldwide 2021.
- [Tornhill, 2015] Tornhill, A. (2015). *Your Code as a Crime Scene*.
- [Tornhill, 2018] Tornhill, A. (2018). *Software Design X-Rays: Fix Technical Debt with Behavioral Code Analysis*.
- [Tornhill, nd] Tornhill, A. (n.d.). Code maat: A command line tool to mine and analyze data from version-control systems.
- [Ulander, 2017] Ulander, D. (2017). Software architectural metrics for the scania internet of things platform : From a microservice perspectiv. Beschrijft metrics voor:
- complexity
- importance of a service
- absolute criticality of a service.
- [Vera-Rivera et al., 2021] Vera-Rivera, F. H., Gaona, C., and Astudillo, H. (2021). Defining and measuring microservice granularity—a literature overview. *PeerJ Computer Science*, 7:e695.
- [Visockis, nd] Visockis, S. (n.d.). Metadata.
- [Vural and Koyuncu, 2021] Vural, H. and Koyuncu, M. (2021). Does domain-driven design lead to finding the optimal modularity of a microservice? *IEEE Access*, 9:32721–32733.

- [Wang, 2009] Wang, X. J. (2009). Metrics for evaluating coupling and service granularity in service oriented architecture. *Proceedings - 2009 International Conference on Information Engineering and Computer Science, ICIECS 2009*.
- [Wang et al., 2021] Wang, Y., Kadiyala, H., and Rubin, J. (2021). Promises and challenges of microservices: an exploratory study. *Empirical Software Engineering*, 26.
- [Waseem et al., 2021] Waseem, M., Liang, P., Shahin, M., Salle, A. D., and Márquez, G. (2021). Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software*, 182.
- [Yourdon and Constantine, 1979] Yourdon, E. and Constantine, L. L. (1979). *Fundamentals of a Discipline of Computer Program and Systems Design*.
- [Zimmermann, 2017] Zimmermann, O. (2017). Microservices tenets: Agile approach to service development and deployment. *Computer Science - Research and Development*, 32:301–310.
- [Zimmermann et al., 2003] Zimmermann, T., Diehl, S., and Zeller, A. (2003). How history justifies system architecture (or not). *International Workshop on Principles of Software Evolution (IWPSSE)*, 2003-January:73–83.
- [Zimmermann et al., 2004] Zimmermann, T., Weißgerber, P., Diehl, S., and Zeller, A. (2004). Mining version histories to guide software changes. *Proceedings - International Conference on Software Engineering*, 26:563–572.
- [Zimmermann et al., 2005] Zimmermann, T., Weißgerber, P., Diehl, S., and Zeller, A. (2005). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31:429–445.
- [Zórnio, 2020] Zórnio, L. (2020). How to choose wisely when defining microservices granularity.

Appendices

A Research Planning

