Universidade Estadual de Campinas
Instituto de Computação

Mateus Gabi Moreira

On the Feasibility of Using SOA Metrics to Assess Cohesion during Microservices Evolution

Sobre a Viabilidade de Usar Métricas SOA para Avaliar a Coesão durante a Evolução dos Microsserviços

CAMPINAS
2023

Mateus Gabi Moreira

On the Feasibility of Using SOA Metrics to Assess Cohesion during Microservices Evolution

Sobre a Viabilidade de Usar Métricas SOA para Avaliar a Coesão durante a Evolução dos Microsserviços

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Breno Bernard Nicolau de França

Este exemplar corresponde à versão final da Dissertação defendida por Mateus Gabi Moreira e orientada pelo Prof. Dr. Breno Bernard Nicolau de França.

CAMPINAS
2023

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

Moreira, Mateus Gabi, 1996-
M813o On the feasibility of using SOA metrics to assess cohesion during microservices evolution / Mateus Gabi Moreira. – Campinas, SP : [s.n.], 2023.

Orientador: Breno Bernard Nicolau de França.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de software. 2. Evolução de software. 3. Métricas de coesão. I. França, Breno Bernard Nicolau de, 1983-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações Complementares

Título em outro idioma: Sobre a viabilidade de usar métricas SOA para avaliar a coesão durante a evolução dos microsserviços
Palavras-chave em inglês:
Software architecture
Software evolution
Cohesion metrics
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Breno Bernard Nicolau de França [Orientador]
Paulo Roberto Miranda Meirelles
Bruno Barbieri de Pontes Cafeo
Data de defesa: 10-10-2023
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-5714-318X
- Currículo Lattes do autor: http://lattes.cnpq.br/8133924488381617

Universidade Estadual de Campinas
Instituto de Computação

Mateus Gabi Moreira

On the Feasibility of Using SOA Metrics to Assess Cohesion during Microservices Evolution

Sobre a Viabilidade de Usar Métricas SOA para Avaliar a Coesão durante a Evolução dos Microsserviços

Banca Examinadora:

- Prof. Dr. Breno Bernard Nicolau de França
  IC/UNICAMP
- Prof. Dr. Paulo Roberto Miranda Meirelles
  IME/USP
- Prof. Dr. Bruno Barbieri de Pontes Cafeo
  IC/UNICAMP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 10 de outubro de 2023

Agradecimentos

À medida que concluo este trabalho, é impossível não expressar minha gratidão a todos aqueles que contribuíram de alguma forma para a realização deste trabalho.

Primeiramente, quero expressar minha gratidão ao meu orientador, Prof. Dr. Breno Bernard Nicolau de França, pela orientação, paciência e insights valiosos ao longo deste processo. Sua dedicação e apoio foram fundamentais para o desenvolvimento desta pesquisa.

Agradeço também aos membros da banca examinadora, Prof. Dr. Paulo Roberto Miranda Meirelles e Prof. Dr. Bruno Barbieri de Pontes Cafeo, por sua disponibilidade em revisar este trabalho e por suas sugestões construtivas, que contribuíram significativamente para a qualidade final desta dissertação.

Não posso deixar de agradecer aos colegas de laboratório e aos amigos que compartilharam ideias, conhecimentos e experiências ao longo desta jornada. Suas contribuições foram inestimáveis.

À minha família, agradeço pelo constante apoio, incentivo e compreensão durante os momentos desafiadores. Suas palavras de encorajamento foram super importantes até a conclusão deste projeto. Em especial para Sheila que nos momentos mais difíceis me apoiou a seguir em frente. Ao José Marcelo, meu filho, que virou meu mundo de cabeça para baixo e me mostrou que não temos controle de nada.

Por fim, estendo minha gratidão ao Instituto de Computação, cujo ambiente propício à pesquisa e ao aprendizado foi vital para o desenvolvimento deste trabalho. Além disso, agradeço à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 88887.388238/2019-00, pelo apoio financeiro que possibilitou a realização deste estudo.

Resumo

CONTEXTO: A Arquitetura de Microsserviços (MSA) oferece benefícios como complexidade reduzida, mas também desafios como complexidade de operação e manutenção. O monitoramento da qualidade do software é crucial para evitar aplicações com baixa coesão e altamente acopladas. Métricas podem ajudar a medir a manutenibilidade, mas precisam de ampla validação empírica antes de serem usadas para tomar decisões. Métricas de manutenibilidade específicas para MSA são desconhecidas, e métricas propostas para Arquitetura Orientada a Serviços (SOA) não foram avaliadas para o contexto de MSA. OBJETIVO: Dadas essas questões e a evidência empírica limitada, o problema abordado neste trabalho é a validação empírica das métricas projetadas para SOA aplicadas para medir a coesão de MSA. MÉTODO: Realizamos dois estudos principais. Primeiro, revisamos a literatura para identificar métricas para medir coesão em aplicações com arquitetura MSA. No segundo estudo mineramos repositórios de software, incluindo vários microsserviços e suas versões estáveis. Para isso, selecionamos o repositório a ser minerado; escolhemos uma aplicação MSA de código aberto para avaliar o procedimento de análise proposto. Em seguida, desenvolvemos uma ferramenta de apoio à coleta e análise de dados. Por fim, realizamos uma análise quantitativa (baseada em séries temporais) e qualitativa (baseada em código fonte). RESULTADOS: Nossos resultados sugerem que é possível visualizar a evolução da coesão em arquiteturas de microsserviços usando as métricas SOA selecionadas. No entanto, identificamos que diferentes perspectivas (controladores x microsserviço) usando as mesmas métricas podem apresentar resultados diferentes. O procedimento seguido em nosso método de pesquisa apoiou o monitoramento e a análise da coesão durante a evolução dos microsserviços, que as equipes de software podem usar para rastrear problemas de manutenibilidade durante todo o ciclo de vida do desenvolvimento de software. CONCLUSÃO: Nosso estudo indica que as métricas SOA ($SIDC1$, $LOC_{message}$ e $SIIC$) podem ser usadas para analisar a evolução de aplicações MSA. Quando aplicadas a controladores, essas métricas geralmente exibem um nível estável de coesão, mas a coesão diminui ao avaliar microsserviços inteiros.

Abstract

CONTEXT: Microservices Architecture (MSA) offers benefits like reduced structural complexity and maintainability and operational complexity challenges. Monitoring software quality is crucial to prevent low-cohesion and highly coupled applications. Metrics can help assess maintainability but need wide empirical validation before using them in decision-making. Maintainability metrics specific to MSA are unknown, and metrics proposed for Service-Oriented Architecture (SOA) have not been evaluated for MSA. OBJECTIVE: Given these issues and the limited empirical evidence, the problem addressed in this work is the empirical validation metrics designed for SOA applied to measure MSA cohesion. METHOD: We conducted two major studies. First, we reviewed the literature to identify metrics for assessing cohesion in MSA applications. Second, we mined software repositories, including various microservices and their releases. For that, we selected a repository to be mined; we chose an open-source MSA application to assess the proposed analysis procedure. Then, we developed a tool to support data collection and analysis. Finally, we performed a quantitative (based on time series) and qualitative analysis (based on source code). RESULTS: Our results suggest it is possible to visualize the evolution of cohesion in microservices architectures using the selected SOA metrics. However, we identified that different perspectives (controllers vs. microservice) using the same metrics may present different results. Generally, the procedure followed in our research method supported the monitoring and analysis of cohesion during microservices' evolution, which software teams may use to track maintainability issues during the whole software development life cycle. CONCLUSION: Our study indicates that SOA metrics ($SIDC1$, $LOC_{message}$, and $SIIC$) can be used to analyze the evolution of MSA applications. When applied to Web controllers, these metrics generally exhibit a stable level of cohesion, but cohesion decreases when evaluating entire microservices from a higher-level perspective.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Context

The adoption of Microservices Architecture (MSA) has witnessed significant growth in recent years [19]. This architectural style can reduce application complexity by decomposing functionalities into small, independent units known as microservices [36, 34]. A microservice is an autonomous application with lightweight access interfaces, source code, database, and an independent lifecycle [23, 36, 16]. One of the MSA's claimed benefits is providing highly cohesive modules (i.e., microservices).

Microservices offer several advantages, including technology diversity, improved scalability, enhanced productivity, and simplified deployment [1, 25]. However, they also bring challenges such as maintainability and operational complexity during their evolution [36, 16, 7].

One of these challenges was experienced in the Industry by Amazon Prime Video. The Amazon Prime Video Team published a white paper with an interesting case this year[1] that describes how they achieved a significant cost reduction of 90% in their monitoring service. They improved this by transitioning from a microservice-based architecture to a monolithic application. Such a shift allowed them to achieve greater scale and resilience. This case highlights the classic Brook's statement "*no silver bullet*" [9], clarifying that no software technology, including microservices, fits all solutions. Hence, it is still challenging to study and understand the various software attributes in microservices and how they interact to support business goals.

The increasing adoption of Microservices Architecture brings both benefits and challenges. It is crucial to conduct research and analysis to characterize the software attributes within microservices better, enabling organizations to harness the full potential of this architectural style while effectively managing its complexities. Organizations in the Industry can make informed decisions and effectively address the challenges associated with MSA.

[1] Available online: `https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90`

1.2 Research Problem

Software architects often rely on manual practices such as code reviews to ensure the evolvability of their microservices [6]. However, these practices can be error-prone and reliant on decentralized teams exchanging knowledge [6], potentially impeding the practical evolution of microservices and leading to maintenance issues. The demand for swift and dependable software product evolution has intensified. Hence, it becomes imperative to consistently monitor maintainability throughout the process to avert rising maintenance costs and to deliver valuable software products.

Software maintenance constitutes a significant portion of the overall software development effort. Furthermore, the inability to achieve rapid and dependable software evolution might result in missed business opportunities [5]. Therefore, it is essential to measure maintainability throughout the software evolution. To assess and monitor software maintainability, metrics such as size, complexity, coupling, and cohesion can be employed [7].

This dissertation assesses the cohesion property to evaluate microservices' proper architectural design and evolution. Cohesion, a crucial software attribute, significantly impacts software maintainability [8, 15]. t quantifies the degree to which the elements within a module are interconnected and logically belong together [32]. n the literature, there are service-level cohesion metrics [7, 27, 28, 3]. However, none of these proposed metrics identified in the literature consider microservices principles and characteristics or even evaluate them empirically in the context of MSA [7]. This lack of specific metrics for microservices makes tracking software maintainability evolution in MSA applications more difficult.

The effectiveness of software metrics depends on their proper characterization and validation. According to Pressman and Maxim [30], certain principles should be followed when creating metrics:

- A metric should have desirable mathematical properties (theoretical validity).
- When a metric represents a software characteristic that increases when positive traits occur or decreases when undesirable traits are encountered, the metric's value should increase or decrease similarly.
- *Each metric should be validated empirically in many contexts before being published or used to make decisions.*

Numerous software metrics have been proposed, but not all are useful for software engineers [30]. Some require overly complicated measurement procedures, while others are too esoteric for real-world professionals to learn. Additionally, some metrics go against the basic intuitive principles of what constitutes high-quality software. Given these issues, the literature limitation, and their unresolved problems, the problem addressed in this work is the empirical validation metrics designed for service-oriented architecture applied to microservices cohesion and how to assess cohesion metrics during the software evolution.

1.3 Research Goals and Question

Our main goal is to provide empirical evidence that cohesion metrics, initially proposed for SOA applications, can be used in MSA applications, considering an evolution perspective. Reaching this primary goal, software practitioners can track and analyze maintainability issues in microservices during their evolution, avoiding those before deployment. More specifically, our study aims to empirically explore the practical applicability of well-known cohesion metrics for Service-Oriented Architecture (SOA) in real-world MSA applications in the context of their evolution. We propose the following research question: *Can SOA cohesion metrics be used to track maintainability issues during the evolution of MSA applications?*

1.4 Contributions

We have selected and evaluated a set of cohesion metrics through empirical research, which can be used to measure cohesion in MSA. Additionally, we have devised a method[2] to evaluate cohesion throughout the evolution of a microservice.

Our research focused on cohesion metrics for Service-Oriented Architecture (SOA) in the Microservices Architecture (MSA) context. To accomplish this, we created an empirical method that tracks and analyzes cohesion as microservices progress.

Our analysis concentrated on a specific group of cohesion metrics (found in Section 2.4). These metrics included *SIDC1*, $LoC_{message}$, and *SIIC*. We then assessed their practical effectiveness in a real-world scenario by studying how microservices' cohesion changed over time. We assessed the metrics in Spinnaker - an open-source project.

Our methodology is grounded in Mining Software Repositories (MSR), a widely acknowledged empirical technique in Software Engineering research [17]. We have also created a tool to support this methodology that automatically evaluates metrics from REST controllers. This tool can help developers identify areas for enhancement and improve cohesion.

The first results of our analysis were published in: *Analysis of microservice evolution using cohesion metrics* by Mateus Gabi Moreira and Breno Bernard Nicolau De França. In Proceedings of the 16th Brazilian Symposium on Software Components, Architectures, and Reuse, SBCARS '22, page 40–49, Uberlândia, MG, Brazil, 2022.

Another contribution is the software that supports the methodology of this work:

- **Aegeus**: library to measure cohesion for a microservice using the metrics selected in this paper. This tool supports Spring applications with Rest Controllers, gRPC applications, and a generic microservice descriptor. Writing *plug-ins* to translate from source code to the generic descriptor can support other languages and frameworks. Available in: `https://github.com/MateusGabi/Aegeus`

- **Aegeus Scripts**: library with a set of scripts automatically measures the cohesion of a microservice for the n latest versions. This library comprises the download of a

[2]In this dissertation, the 'method' is used in our research for exploratory investigation purposes and needs additional studies and tailoring to be transferred to practitioners.

repository (main branch); downloading *nth* latest versions using git tags; assessing cohesion metrics for each version (using Aegeus library); reading logs and storing metric assessments on CSV file; generating graphics using R Script and Python. Available in `https://github.com/MateusGabi/Aegeus-scripts`

These contributions can be used in Academia and, with some limitations, in Industry. This study expands our understanding of cohesion measurement in modern software architectures by evaluating the applicability of service-oriented architecture (SOA) metrics in real-world MSA applications. The study also highlights the need for further research and refinement of cohesion metrics tailored to MSA applications. The selected SOA metrics (SIDC1, $LoC_{message}$, and SIIC showed potential and provided insights into the cohesion dynamics of microservices. This opens up future research to refine existing metrics and propose new ones that better address MSA applications' specific challenges and characteristics.

Furthermore, the study offers practical importance for software development teams working with microservices. The methodology developed in this study allows software engineers to analyze the cohesion of their microservices and identify potential improvement areas. The study supports the maintenance of microservices to ensure the long-term quality of the software. Software teams can proactively track and manage cohesion by using the metrics in this work and addressing the identified problems, leading to more maintainable microservices.

1.5 Organization

This thesis is structured as follows. Chapter 2 delves into the theoretical foundation necessary to understand the main concepts used within this dissertation, such as Service-Oriented Architecture (SOA), Microservices Architecture (MSA), software quality attributes, and the relationship between cohesion in Microservices and cohesion metrics for SOA, as well as it discusses related works. Chapter 3 outlines the research method used to analyze the applicability of SOA cohesion metrics in the context of an MSA application evolution. Chapter 4 presents the results found by this work, including insights gained from the results and how we mitigated threats to validity. Chapter 5 summarizes the findings of this work, its impact on Academia and the Industry, and future work.

Chapter 2

Background and Related Work

Bass *et al.* [4] define software architecture as a realization of early-design decisions made regarding decomposing the system into parts (i.e., modules, services, and components.), allowing organizations to meet their business goals. Concrete or actual software architecture also results from conscious and unconscious decisions made throughout the application life cycle. One of the leading architectural styles in the literature and industry is Service-Oriented Architecture (SOA), and in recent years, Microservices have increased relevance as an extension of SOA. This chapter discusses the primary concepts and associated literature. It discusses SOA and MSA pros and issues, the quality attributes related to this architectural style, the cohesion property in the context of SOA, and its metrics found in the literature.

2.1 Service-Oriented Architecture (SOA)

Service-Oriented Architecture (SOA) is a software system architecture style that enables providing services to client users or other services through a public interface [26]. SOA offers a collection of independent and reusable functionalities packaged into services. These services work together to create a more complex system, reducing complexity and costs while increasing flexibility and operational efficiency [26]. However, the implementation of SOA can be challenging and introduces some disadvantages as the system grows [26].

One of the significant drawbacks is the large size of the services' code base and their undesired complexity. Debugging, fixing issues, and developing new functionalities require considerable effort through the code base, making maintenance difficult [16]. The second challenge is the "dependency hell" problem when adding or updating services or external libraries, resulting in inconsistent systems [16] due to many internal or external dependencies.

2.2 Microservices Architecture (MSA)

Microservices Architecture (MSA) is an architectural style that provides a set of independently deployable applications called microservice [36, 34, 7]. Each microservice is loosely coupled and isolated in a small coherent and autonomous unit, running in its process [23]

and communicates through lightweight interfaces [23, 16]. As microservices are independently deployable [23, 36, 16], they can be written by different and distributed teams and using different technologies [23]. There are some discussions in academia and industry about the relationship of microservices with SOA [16, 23]. Bogner et al. [7] state that MSA is a specialization of SOA. We discuss the following typical characteristics that most microservice architectures exhibit.

- **Componentization** [23]. The primary way of componentization is decomposing the application into (micro)services. The main reason for using services is that services are independently deployable. It means that many single changes require redeploying of just a few services. Nevertheless, some changes will impact the service interface, but an exemplary microservice implementation minimizes these cases through cohesive boundaries and service contracts. Most programming languages do not have an efficient mechanism for defining component interfaces. Thus, only documentation and discipline avoid tight coupling between components. Services prevent these problems by explicit remote call mechanisms, but remote calls are more expensive than in-process calls.

- **Business Capabilities** [23]. Microservice teams are organized around business capability. This service implementation requires a broad-stack implementation of software. Consequently, the teams are cross-functional, including a broad range of skills: user experience, database, development, and project management. Such teams are responsible for building and operating their products. Monolith applications can always be decomposed around business capabilities, which is not typical.

- **Products fit to user needs** [23]. The project-oriented development aims to deliver some piece of software to users, then considered to be solved. With microservices, a team should own a product over its lifetime, which means that a development team takes full responsibility for the software in production. Besides, the main goal of a software team is to assist its users in enhancing business capability. The smaller granularity of services makes the personal relationship between developers and their users easier.

- **Smart endpoints and dumb pipes** [23, 16]. The biggest issue in breaking down a monolith application into a set of microservices is changing the communication pattern. In monolith applications, components communicate via in-memory method invocation or function call. An application with microservice architecture aims to be as decoupled and cohesive as possible. A microservice receives a request, applies business logic, and produces a response. Two protocols are most common for this: (i) HTTP request-response and (ii) messaging over a lightweight message bus like RabbitMQ and ZeroMQ. The smart endpoints are because the endpoints in the services produce and consume messages, and dumb pipes are because the messaging bus works as a message router to the microservices.

- **Decentralization** [23]. By breaking the monolith into services, we can choose the stack that solves the problem better. A monolithic team generally writes down its

standards on a document, but microservices teams prefer to produce useful tools that other developers can use to solve similar problems. A company shares these tools via open-source repositories. For instance, Netflix follows this internal open-source philosophy of sharing internal tools as libraries on its GitHub repositories. These tools commonly focus on data storage, inter-process communication, and infrastructure automation.

- **Intense Automation** [23] [18]. Many software products have introduced infrastructure automation to reduce the operational complexity of the building, testing, and deployment to production. Besides that, automation leads teams to reduce the delivery time and the time to collect feedback from users. Typically, a monolithic application is built, tested, and deployed to production quickly for each new version. Conversely, as we commented on the Business Capabilities characteristic, a microservice application is quite different since each may have a broad stack. Nevertheless, each microservice is built, tested, and deployed to production in isolation.

2.3 Software Quality Attributes

Software quality attributes help the architecture to achieve the requirements [4]. Some quality attributes can be observed through the system execution (Performance, Security, Availability, Functionality, Usability). Still, others may need to keep their internal structures (Maintainability, Portability, Reusability, Integrability, Testability) [4].

- **Performance** refers to the system's responsiveness, For example, the time required to respond to events or the number of events processed in some time interval.
- **Security** measures the system's ability to resist unauthorized attempts to service while still providing its services to authorized users.
- **Availability** measures the time the system is up and available to users. It is measured by the time between failures and how quickly the system can resume operation in the event of failure.
- **Functionality** is the ability of the system to do the work for which it was intended.
- **Usability** refers to the ability of the system to make it quick and easy for a user to learn how to use it while the system responds fast and prevents user errors.
- **Portability** is the ability of the system to run in different environments.
- **Reusability** relates to software architecture in that architectural components are units of reuse. How reusable a component is depends on how tightly coupled it is with other components.
- **Integrability** is the ability to make the separately developed components of the system work correctly together.

- **Testability** refers to the ease with which software can demonstrate its faults through testing.
- **Modifiability** is making changes quickly and cost-effectively. It may be the attribute most closely aligned with the architecture of a system. A widespread change is more costly than making a change into a single component. Sometimes, it is called *maintainability*, but the ISO/IEC SQuaRE [21] defines it as a sub-characteristic for Maintainability.

2.4 Software Evolution, Maintainability and Cohesion

Over the years, software practitioners have referred to software maintenance as Swanson's [33] typology based on three classifications: *perfective maintenance*, where the system in terms of its performance, processing efficiency, or maintainability, *adaptive maintenance* to adapt the system to changes in its data environment or processing environment and *corrective maintenance* to correct processing, performance or implementation failures of the system.

Chapin et al. [10] proposed a more detailed classification of Software Evolution and Software Maintenance. Software Maintenance applies activities and processes to existing software to modify its operations or contribute to the system's business. It includes quality assurance activities and management and is often done in the context of software evolution [10]. Software Evolution refers to applying software maintenance activities and processes to create a new version of operational software with modified functionality or properties experienced by the customer [10]. The term is sometimes used interchangeably with software maintenance.

Maintainability, as defined by the ISO/IEC SQuaRE standard, signifies the degree of effectiveness and efficiency for maintainers responsible for modifying a product or system [21]. The impact of low maintainability on costs and business opportunities has been well-documented [5], rendering its management a significant challenge for software engineers. There are several metrics used for controlling maintainability during the software evolution. One strategy to define maintainability metrics is measuring the software attributes. In this context, software *cohesion* emerges as a pivotal software attribute, contributing to a comprehension of internal quality and maintainability [8, 15]. Stevens *et al.* [32] defined cohesion as *the degree of the elements that belong together* ***inside*** *a module.*

2.4.1 Cohesion in Microservices

Cohesion refers to how components within a microservice are related and focused on a single responsibility or specific functionality. With high cohesion, a microservice has a more well-defined and isolated scope. So the benefits of using microservices increase, such as the separation of functionalities, ease of maintenance since the size of the code is generally smaller, facilitates modification or the implementation of new functionalities,

and scalability as the demand for the functionalities increases, which reduces the cost of hardware.

The high level of componentization is an achievement when a Microservice Application is well designed and aligned with the cross-functional teams distributed around business capabilities. This way, teams are responsible for the entire lifestyle of their microservices. They commonly work in only one set of microservices and keep them healthy in maintainability, improving product and business requirements agility.

Thus, using microservices favors the development of independent functionalities with low coupling and high cohesion. However, the natural increase of cohesion does not inhibit the creation of highly coupled and low cohesion applications if software practitioners do not have the interest and tools that help them design, implement, and evolve their applications. The team should track their quality continuously because of the 7th Lehman's Law: *"programs will be perceived as declining quality unless rigorously maintained and adapted to a changing operational environment"* [22].

Currently, some metrics are available in the literature but not empirically evaluated, limiting their practical application in the industry. Meanwhile, in the industry, several architecture problems are addressed as technical debts without specializing and applying a solution to the issue, specifically, making the solutions experience-based or *ad-hoc* from the software practitioners themselves [6].

Cohesion is crucial to maintainability and design quality in software architectures. While several studies have explored cohesion in the context of Service-Oriented Architecture (SOA), exploring cohesion in Microservices remains scarce. Many researchers have studied cohesion metrics in SOA applications to predict maintainability issues and assess system qualities. For instance, Perepletchikov et al. [27] developed a set of cohesion metrics for early maintainability issue prediction, while Shim et al. [31] proposed a design quality model for SOA systems. However, their chosen metric was deemed unsuitable for cohesion assessment. Athanasopoulos et al. [3] also introduced metrics to assess cohesion in service interfaces, enabling service decomposition for SOA applications. However, these studies do not explicitly address cohesion in Microservices.

A literature review by Bogner et al. [7] encompassed metrics for maintainability in both SOA and Microservices Architecture (MSA). Notably, the authors encountered challenges in studying cohesion and suggested the possibility of employing metrics defined by Perepletchikov et al. after empirical studies in real-world applications. Despite these efforts, the explicit investigation of cohesion in Microservices remains largely unexplored.

Given the lack of metrics specifically created for Microservices, it is possible that metrics used in Service-Oriented Architecture can be adapted and applied to assess the cohesion in Microservices. The premise is that, although the characteristics and challenges of Microservices are distinct from other architectures, classic metrics may provide relevant insights into the cohesion of the potential of services in a Microservices-based architecture. However, conducting empirical studies and careful evaluations are essential to verify the effectiveness and relevance of these classic metrics in a Microservices context. Using these metrics may offer an initial approach to assess the cohesion of Microservices until more specialized metrics are developed and validated for this architecture in future works. For a more detailed understanding of the related work in the literature, Section 2.4.3 discusses

the studies as mentioned earlier and their implications.

2.4.2 Cohesion Metrics

Chidamber and Kemerer's [12] work on software metrics is considered prominent. They developed a suite of metrics based on theoretical evaluation. One of the metrics developed is Lack of Cohesion in Methods (LCOM1), which calculates the number of pairs of methods in a class that do not use any attribute in common. Later on, Chidamber and Kemerer [13] developed and implemented LCOM2. LCOM2 is calculated as the number of pairs of methods in a class that do not use any attributes in common minus the number of pairs of methods that do. If this difference is negative, LCOM2 is set to zero.

Hitz and Montazeri [20] proposed two metrics: LCOM3 and LCOM4. They defined an undirected graph G, where the vertices represent the methods of a class. An edge is created between two vertices if the corresponding methods use at least one attribute in common. LCOM3 is the count of the connected components of G. LCOM4 is similar to LCOM3. Still, additionally, the graph G has an edge between vertices representing methods m and n if method m invokes n or vice versa.

As we move forward, this section will explore cohesion metrics in microservices. In this context, it's worth noting that we can gain valuable insights from the works of Chidamber and Kemerer as well as Hitz and Montazeri in the context of object orientation.

Bogner *et al.* [7] presents a literature review that includes the metrics presented in this section. We excluded the cohesion metric in [31] as, from our point of view, it is a metric for *coupling* instead of *cohesion* since it focuses on the interaction between services. Every metric below goes from 0 to 1.

Perepletchikov et al. [27] expanded the notions of cohesion to SOA and developed a set of design-level cohesion metrics to assess different aspects of cohesion at the service level, as follows:

- **Service Interface Data Cohesion (SIDC1)**: this metric quantifies the cohesion of a service based on the cohesiveness of the operations exposed in its interface, which means operations are sharing the same type of input parameter [27]. SIDC2 evolves from SIDC1 by considering both in/out parameter types. Formal definition:

$$SIDC(s) = (Common(Param(SOp(si_s))) \\ + Common(returnType(SOp(si_s)))) / \\ (Total(SOp(si_s)) * 2),\ where$$

  - $SOp(si_s)$ is the set of all operations exposed in the interface si_s of service s.
  - $(Common(Param(SOp(si_s)))$ returns the number of service operations pairs that have at least one input parameter type in common.
  - $Common(returnType(SOp(si_s))))$ returns the number of service operations pairs that have the same return type.
  - $Total(SOp(si_s))$ returns the number of combinations of operation pairs for the service interface si_s.

- **Service Interface Usage Cohesion (SIUC)**: This metric quantifies the client usage patterns of service operations when a client invokes a service. Formal definition:

$$SIUC(s) = Invoked(clients,\ SOp(si_s))/ \\ (|clients| * |SOp(si_s)|),\ where$$

  - $clients$ is the set of all service clients s.
  - $Invoked(clients,\ SOp(si_s))$ returns the number of used operations per client.

- **Service Sequential Usage Cohesion (SSUC)**: This metric quantifies the cohesion of a service based on the cohesiveness of the operations exposed in its interface client usage, taking into consideration the dependencies between service operations. Formal definition:

$$SSUC(s) = InvokedSequence(clients, SOp(si_s))/ \\ (number(clients) * |SOp(si_s)|),\ where$$

  - $InvokedSequence(clients, SOp(si_s))$ returns the sum of all the sequentially used operations per client.
  - $number(clients)$ is the number for clients of service s.

- **Service Interface Sequential Cohesion (SISC)**: SISC evolves from SSUC [27]. It quantifies sequential properties of dependencies between service operations in the client usage patterns [28]. Formal definition:

$$SISC(s) = SeqConnected(SOp(si_s))/ \\ Total(SOp(si_s)),\ where$$

  - $SeqConnected(SOp(si_s))$ returns the number of service operations pairs with sequential operations calls.
  - $Total(SOp(si_s))$ returns the number of all possible combinations operation pairs.

- **Strict Service Implementation Cohesion (SSIC)**: This metric quantifies the cohesion of a service based on the cohesiveness in the implementation of the operations. Formal definition:

$$SSIC(s) = |IC(s)|\ / \\ (|(C_s \bigcup I_s \bigcup P_s \bigcup H_s \bigcup BPS_s)| * |SO(si_s)|),\ where$$

  - $C_s \bigcup I_s \bigcup P_s \bigcup H_s \bigcup BPS_s$ is the set of all implementation elements of service s (classes, interfaces, procedural packages, package headers, and business process scripts).

  - $IC(s)$ is the set of all common service implementation elements shared by the service operations.

- **Loose Service Implementation Cohesion (LSIC)**: This metric is a variation of SSIC, but the same implementation elements can implement services or have been indirectly connected via implementation elements. SSIC and LSIC were revised and became *SIIC - Service Interface Implementation Cohesion* where SIIC is **exactly** the same as SSIC. Formal definition:

$$LSIC(s) = |CC(s)| \ / \\ (|(C_s \bigcup I_s \bigcup P_s \bigcup H_s \bigcup BPS_s)| * |SO(si_s)|)$$

$$SIIC(s) = |IC(s)| \ / \\ (|(C_s \bigcup I_s \bigcup P_s \bigcup H_s \bigcup BPS_s)| * |SO(si_s)|), \ where$$

  - $C_s \bigcup I_s \bigcup P_s \bigcup H_s \bigcup BPS_s$ is the set of all implementation elements of service s (classes, interfaces, procedural packages, package headers, and business process scripts).
  - $IC(s)$ is the set of all common service implementation elements shared by the service operations.
  - $CC(s)$ is the set of all, directly and indirectly, service implementation elements shared by the service operations.

- **Total Interface Cohesion of a Service (TICS)** This metric is a normalized sum of all values of the previous cohesion metrics. Formal definition:

$$TICS(s) = (SIDC(s) + SIUC(s) + SIIC(s) + SISC(s))/4$$

Consider an example service named *EnrollStudent* with three operations:

- *getLibraryClearance(int libraryMemberId): bool*
- *checkPrerequisiteCourses(string studentId, Course): bool*
- *enrollStudentIntoCourse(string studentId, Course): bool*

The service consumer *Student* will call operations in a strict sequence:

getLibraryClearance → *checkPrerequisiteCourses* → *enrollStudentIntoCourse*

In this way, we have the following results:

- SIDC (EnrollStudent) = 2/3 = 0.67;
- SIUC (EnrollStudent) = 3/3 = 1;
- SSUC (EnrollStudent) = 3/3 = 1;

- SSIC (EnrollStudent) = 6/(3*9) = 0.22;
- The value of TICS (EnrollStudent) will then be equal to: (0.67+1+1+0.22)/4 = 0.72 indicating stronger then average overall cohesion of a service.

Another set of metrics is presented in [3], in which the authors propose an approach that enables the cohesion-driven decomposition of web services interfaces without analyzing the interface implementation. This approach defines the following cohesion metrics:

- **Lack of Domain-level Cohesion** (LoC_{dom}): it measures the cohesion in a service interface given the terms of its operation that correspond to concepts of the targeted domain [3], i.e., service is highly cohesive when its operations share common domain-level terms. Formal definition:

  $G^*_{si} = (V_{si}, E_{si}, OpS_*)$

  - G^*_{si} is an interface-level graph for a service interface, si, and a similarity function, OpS_*, which reflects the degree to which the operations of si are related. Nodes are the operations of si, V_{si}; the edges are the interface-level relations between pairs of operation, E_{si}, if: $OpS_*(op_i, op_j) > 0$;

  $LoC_{dom}(si) = LoC_*(si, OpS_{dom})$, *where*

  - OpS_{dom} is a similarity function of two operation names, $op_i, op_j \in si$, where T_{op_i} and T_{op_j} are the sets of the domain terms, as follow:

    $OpS_{dom}(op_i, op_j) = \frac{|T_{op_i} \cap T_{op_j}|}{|T_{op_i} \cup T_{op_j}|}$

  - $LoC_*(si, OpS_*)$ is the relative difference between the ideal and the interface-level graph, as follows:

    $LoC_*(si, OpS_*) = \frac{|E_{ideal}| - \sum_{(op_i, op_j) \in E_{si}}^{OpS_*(op_i, op_j)}}{|E_{ideal}|}$, and
    $|E_{ideal}| = \frac{|V_{si}| * (|V_{si}| - 1)}{2}$

- **Lack of Conversation-level Cohesion** (LoC_{conv}): it measures the cohesion in a service interface given two sequential messages. Two operations are similar (high cohesion) when the latter operation input is similar (has common data types) to the output of the first operation. So, this metric analyzes the behavior of client usage. Formal definition:

  $LoC_{conv}(si) = LoC_*(si, OpS_{conv})$, *where*

  - OpS_{conv} is a similarity function of two operations, $op_i, op_j \in si$, where is the average of the similarity between the input of op_i and the output of op_j; and the similarity between the output of op_i and the input of op_j, as follows:

    $MsgS(m_i, m_j) = \frac{|V_{m_i} \cap m_j|}{|V_{m_i} \cup m_j|}$

    $OpS_{conv}(op_i, op_j) = \frac{MsgS(op_i.in, op_j.out)}{2} + \frac{MsgS(op_i.out, op_j.in)}{2}$

- **Lack of Message-level Cohesion** ($LoC_{message}$): it measures the cohesion of an interface given its input and output. An interface is highly cohesive when its operations share standard parameters and return types. $LoC_{message}$ uses the similarity between two operations (input and output). The similarity is the common data types between two operations inputs (and output). Formal definition:

  $LoC_{message}(si) = LoC_*(si, OpS_{message})$, *where*

  - $OpS_{message}$ is a similarity function of two operations, $op_i, op_j \in si$, being the average of the similarity between the input of op_i and the input of op_j; plus the similarity between the output of op_i and the output of op_j, as follows:

    $OpS_{message}(op_i, op_j) = \frac{MsgS(op_i.in, op_j.in)}{2} + \frac{MsgS(op_i.out, op_j.out)}{2}$

Example on *AWS MessageQueue service*, where have thirteen operations:

- *DeleteQueue(string ForceDeletion): ResponseStatus*
- *SetQueueAttributes(AttributedValue): ResponseStatus*
- *GetQueueAttributes(string Attribute): ResponseStatus, AttributedValue*
- *SendMessage(string MessageBody): ResponseStatus, string MessageId*
- *ReceiveMessage(int NumberOfMessages, int VisibilityTimeout): ResponseStatus, Message*
- *PeekMessage(string MessageId): ResponseStatus, Message*
- *DeleteMessage(string MessageId): ResponseStatus*
- *ChangeMessageVisibility(string MessageId, int VisibilityTimeout): ResponseStatus*
- *AddGrant(Grantee, string Permission): ResponseStatus*
- *RemoveGrant(Grantee, string Permission): ResponseStatus*
- *ListGrants(Grantee, string Permission): ResponseStatus, GrantList*
- *GetVisibilityTimeout(int VisibilityTimeout): ResponseStatus, int VisibilityTimeout*
- *SetVisibilityTimeout(int VisibilityTimeout): ResponseStatus*

In this way, we have the following results:

- $LoC_{message} = 0.98$
- $LoC_{conv} = 0.98$
- $LoC_{dom} = 0.81$

2.4.3 Related Work

In recent years, different authors have defined and explored the cohesion of SOA and MSA [27, 31, 3, 7, 2, 24]. In their study, Perepletchikov et al. [27] expanded upon existing concepts of cohesion in object-oriented systems and procedural design to suit the unique qualities of SOA better, thus enabling early prediction of the maintainability of SOA applications. To assess the level of cohesion in service-oriented design constructs, metrics were created. However, these metrics have not been tested empirically to determine their accuracy in predicting service cohesion levels and, more importantly, their ability to forecast maintainability. Our research addresses this gap in empirical validation by examining the feasibility of these metrics for microservices.

Shim et al. [31] established a model for assessing the quality of SOA systems in their early stages. They defined desirable quality attributes and identified the metrics necessary to measure them, creating an assessment model for identifying metrics at different levels of abstraction. The proposed model was tested by applying it to an information management system to verify its ability to deduce the correct set of metrics for estimating changes to system design. However, the authors acknowledge that the metrics derived from the model are in their initial stages and require further testing to validate their accuracy across a broader range of systems.

The authors defined *cohesion* as the degree of relationships between operations defined in a service. They also created a metric called the *average used message factor*, which is the inverse number of service operations. However, we believe this metric is unsuitable for measuring cohesion, as it does not consider the internal relationships between the elements of a module (service). Instead, it is closer to measuring the *coupling* property.

In their study, Athanasopoulos et al. [3] created metrics to evaluate the cohesion of service interfaces to suggest service decomposition. They defined service cohesion as the extent to which service interface operations are related. Using this approach, the authors devised a technique to gradually break down a particular interface without addressing the implementation. The authors did not directly evaluate maintainability but concentrated on clustering interface operations. As a result, the metrics they developed were designed for a different objective, and it may not be feasible to use them for assessing maintainability. Our research investigates the applicability of these metrics for evaluating maintainability in the context of microservices evolution.

Bogner et al. [7] performed a literature review to find metrics for maintainability for SOA and MSA. The authors mapped those metrics into four design properties: size, complexity, coupling, and cohesion. In the cohesion property, the authors note the difficulty of measuring cohesion automatically because of its subjective nature and *the literature not*

widely covered it. The authors highlight the metrics outlined by Perepletchikov et al. [27] to shed light on the insufficient cohesion studies in the literature. Bogner et al. have identified a need for more research on cohesion studies for microservices. There is a lack of sufficient research on this topic, emphasizing the importance of further exploration. Our study aims to provide valuable insights and contribute to a better understanding of cohesion for microservices.

Apolinario and França [2] developed a method to monitor the evolution of MSA coupling. They conducted an experiment to study the behavior of coupling metrics using artificially generated data. The experiment revealed the metric's behavior in different scenarios, providing insights into identifying architectural degradation. The method collects coupling metrics during runtime and monitors them during software evolution. An upward trend in coupling metrics could indicate architectural degradation. This work is similar to ours, but we focus on cohesion instead of coupling. We believe that both coupling and cohesion are important for maintaining software quality.

Table 2.1: Literature on MSA and SOA cohesion metrics.

Reference	Metrics for		Context
	Interface	Implement.	
Perepletchikov et al. [27]	X	X	Early prediction of maintainability issues
Shim et al. [31]	X		Design of quality model to assess SOA systems
Athanasopoulos et. al. [3]	X		Service decomposition
Bogner et al. [7].	X	X	Literature Review

Table 2.1 summarizes the literature on MSA and SOA metrics for cohesion. In this way, this study does not consider the metrics defined by Apolinario and França [2]. Section 2.4.2 defines the metrics from works in Table 2.1. Due to a lack of research on the cohesion property in MSA [7], one may wonder if SOA metrics can be used to evaluate MSA. To address this, we conducted a non-systematic literature review and analyzed cohesion metrics in a real-world MSA application as part of this Master's dissertation [24]. We identified metrics that can automatically assess cohesion and aid software practitioners in detecting maintainability issues early on.

Working in software development, it can be challenging to maintain cohesion across a myriad of microservices. This is primarily due to the complex nature of software systems, which often comprise multiple microservices, technologies, and their dependencies. Assessing cohesion can be subjective, but several resources are available in the literature to guide practitioners in addressing cohesion concerns for SOA, such as metrics, methods, quality models, tools, and frameworks. However, currently, there are no equivalent resources for MSA.

Our Master's is dedicated to assisting software practitioners in overcoming their challenges. Specifically, we aim to explore the feasibility of utilizing SOA Metrics to evaluate cohesion throughout the evolution of microservices. Through our research, both academia and industry can yield benefits.

Chapter 3

Research Method

This chapter describes the research method adopted to analyze the applicability of Service-Oriented Architecture (SOA) cohesion metrics in Microservice Architecture (MSA) applications, focusing on software evolution. Our method is divided into two major studies. First, we reviewed the literature to find metrics used to assess cohesion in Microservice Architecture (MSA) applications. Second, we mined software repositories, including various microservices and their releases. For that, we selected the repository to be mined; we chose an open-source MSA application to assess the proposed analysis procedure. Then, we developed a tool to support data collection and analysis. Finally, we performed a time-series quantitative and qualitative analysis.

3.1 Literature Review

Our *ad-hoc* literature review aims to identify metrics to assess cohesion in Microservices. For that, we followed a process similar to a systematic mapping study, and we searched for works using the following keywords: *software cohesion*, *software maintainability*, *metrics*, *monitoring*, *microservices*, *microservice architecture*, and *service-oriented architecture*. We included service-oriented architecture considering the possibility to test whether SOA metrics can be applied to MSA applications, which was an *a priori* assumption.

As sources, we adopted three search engines: IEEE Xplore, ACM Digital Library, and Scopus. We only included peer-reviewed papers published in English. In addition, we excluded papers unrelated to the research goals, such as those focused solely on non-SOA or non-MSA applications.

Later, we extracted metrics definitions from the selected studies. After the extraction, we ended up with twelve (12) service cohesion metrics (cf. Section 2.4). All of those metrics were proposed in the context of services-oriented approaches. Then, we analyzed all the definitions to understand them in full. Therefore, we choose to work only with *SIDC1*, $LOC_{message}$, and *SIIC*, as shown in Figure 3.1. In the following, we explain the criteria used to select these metrics.

First, we excluded metrics assuming an external perspective about client service usage that contradicts our notion of cohesion as a module characteristic based on its **internal** properties. As a result, we did not analyze the following metrics: SIUC, SSUC, SISC,

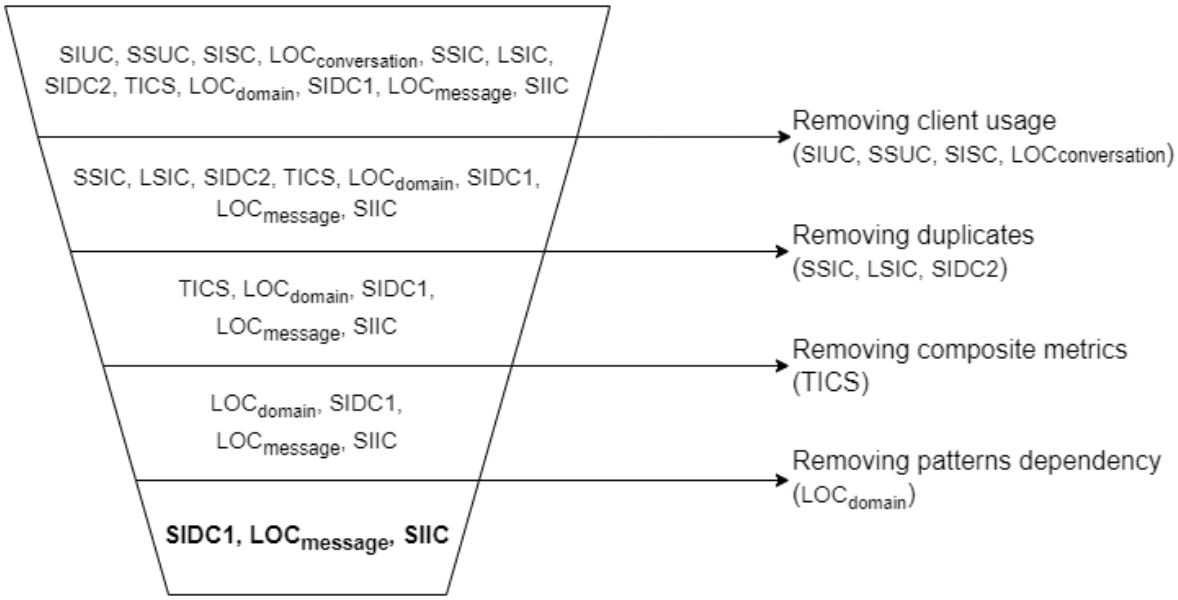


Figure 3.1: Selection Criteria and Resulting Metrics

and $LOC_{conversation}$. These metrics try to infer cohesion characteristics based on external information, e.g., service methods usage patterns or sequence by its clients. We consider these characteristics may even be correlated with cohesion but they are not measuring cohesion. Second, we excluded duplicate or obsolete metrics, keeping only the most recent versions recommended by the authors. Thus, we excluded the following metrics from our analysis: SSIC, LSIC, and SIDC2. Third, we did not analyze the TICS metric because it is a composite metric that aggregates other metrics. So, we decided to keep only the base metrics and, if necessary, to start working with derived metrics. Fourth, we excluded metrics that require knowing developer's patterns, such as naming conventions. Our study is an exploratory analysis conducted independently of the authors who proposed the metrics and developed the case application. Therefore, we cannot anticipate the specific patterns implemented in the analyzed case. Consequently, we excluded the LOC_{domain} metric because it depends on patterns for operation names, parameters, and results.

3.2 Repository Mining

This section describes the procedure for mining Spinnaker repositories using our supporting tool Ageus (presented in Section 4.1). This section follows the guidelines proposed in [35]. As previously stated, the objective of this study is to verify the practical applicability of the selected metrics in the literature review (*SIDC1*, $LOC_{message}$, and *SIIC*) in a real-world application.

3.2.1 Repository Selection

As our study concerns software evolution, we needed to find a microservices application with a history of its version releases. Thus, we searched for active repositories hosting

open-source microservices applications with many contributors and contributions. Regarding the source, GitHub is recognized for its many open-source software development projects.

We searched for repositories using the GitHub Search API and two keywords (*microservices* and *Java*). As a result, we found 716 repositories. We only included repositories developed with Java or JVM support (which also includes Kotlin projects). Furthermore, we excluded demos/examples or applications without stable releases and sorted them by stars and the number of commits.

Finally, we chose the Spinnaker project as our case because of its significant code base of thousands of historical changes and releases. Additionally, a large community actively supports Spinnaker, with about eight thousand stars on GitHub. Furthermore, its technical oversight committee includes reputable companies such as Salesforce, Armory, Google, AWS, and Netflix.

Spinnaker is a multi-cloud continuous delivery platform. Its organization on GitHub has over 50 repositories, but not all contain microservices. Therefore, we filtered the repositories to identify microservices. We included microservices written in Java or JVM languages. We excluded archived repositories, demo code, microservices with no releases, documentation repositories, installation scripts, tools, front-end-only repositories, and those developed using non-JVM languages. We also excluded microservices with recent unstable releases.

Spinnaker architecture adopts the API Gateway Design Pattern and comprises a set of twelve microservices. Figure 3.2 illustrates the relationship between these microservices.

- Deck is the browser-based UI. It is developed using Nodejs[1]. We did not analyze the Deck microservice because it uses Nodejs, which is not a JVM-based programming language.
- Gate is the API Gateway that establishes communication between the front-end application, Deck, and other Spinnaker's internal microservices.
- Orca is an orchestration service that coordinates the pipeline stages, tasks, and executions.
- Clouddriver microservice connects to Cloud Providers (such as AWS, CloudFoundry, and Azure) and indexes all deployed resources.
- Front50 is used to persist the metadata of applications, pipelines, projects, and notifications.
- Rosco produces immutable VM images (or image templates) for various cloud providers. It produces machine images (for example, GCE images, AWS AMIs, and Azure VM images). It currently wraps the packer but will be expanded to support additional image production mechanisms.

[1] `https://nodejs.org`

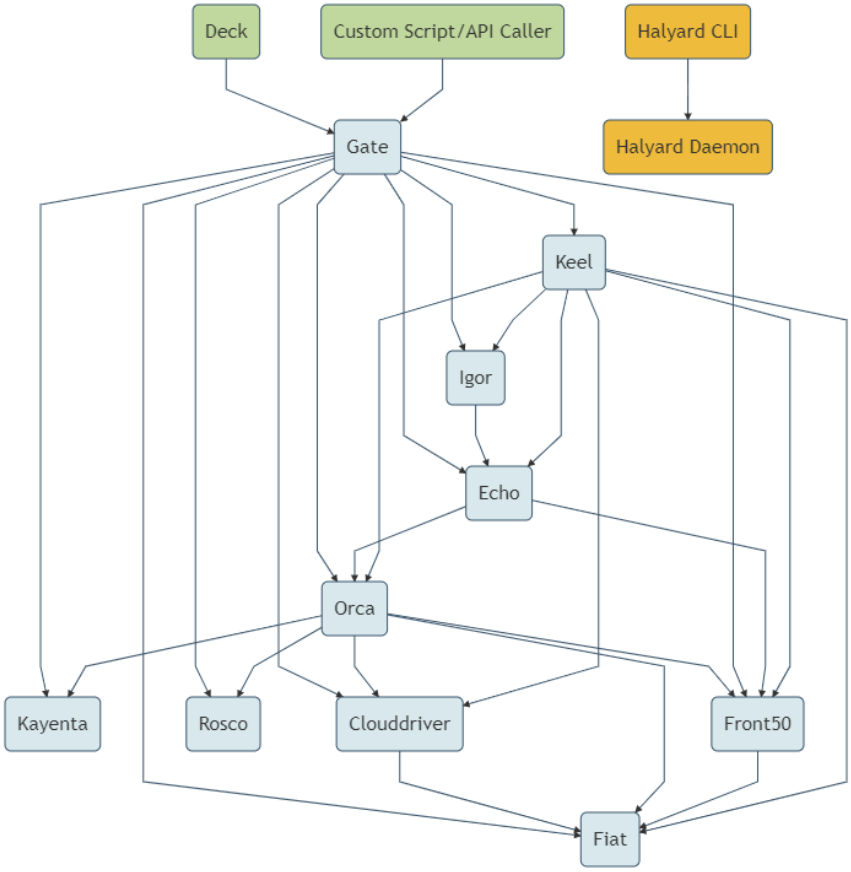


Figure 3.2: Spinnaker's system dependencies.

- Igor triggers pipelines via continuous integration jobs. Igor is used to triggering pipelines via continuous integration jobs in systems like Jenkins and Travis CI, and it allows Jenkins/Travis stages to be used in pipelines.
- Echo is the Spinnaker's event bus. It supports sending notifications (e.g., Slack, email, SMS) and acts on incoming webhooks from services like GitHub.
- Fiat is Spinnaker's authorization service. It is used to query a user's access permissions for accounts, applications, and service accounts.
- Kayenta provides automated canary analysis for Spinnaker.
- Keel powers managed delivery by giving software practitioners shared configurations in the delivery process. It was still under development during the analysis; thus, we excluded it.
- Halyard is Spinnaker's configuration service.

We ultimately selected Clouddriver, Echo, Fiat, Front50, Gate, Halyard, Igor, Kayenta, Orca, and Rosco from all the microservices. These microservices have more than four thousand releases together, and each release is labeled using the Semantic Versioning pattern[2].

We selected only the *major* and *minor* releases and excluded *patch* releases as they usually only fix bugs, which is unlikely to impact service cohesion. Furthermore, the Spinnaker project adopts the approach of one repository per microservice, where each has its own life cycle. Therefore, the versions of the microservices may differ at the same time. For instance, during the software evolution, one microservice may be in version 10 while the other is in version 8 or 3. Hence, we collected the latest 100 versions of each microservice as a sample—Table 3.1 shows which microservice versions we mined.

Table 3.1: Target versions for each microservice.

Microservice	**version**	
	from	to
Gate	4.59.1	6.52
Orca	7.43	8.19
Clouddriver	5.23.4	5.75.0
Igor	2.1	4.5
Echo	1.528	2.29
Fiat	0.52	1.28
Front50	1.124	2.24
Halyard	0.0.1	0.37
Rosco	0.105	1.7

[2]`https://semver.org`

3.2.2 Execution

As mentioned in Section 3.2.1, Spinnaker has an extensive source code base, including pull requests, commits, and issues, which makes mining the repository a challenging task. Therefore, we have created a tool to assist with repository mining.

We develop the Aegeus[3] tool to extract data from the microservices repository for each version (Table 3.1) that the interface must be annotated with Spring `@RestController`. Aegeus finds these classes and extracts: URL, HTTP Method, interface objects, parameters, implementation elements (objects, functions, etc.), and output objects. Section 4.1 describes in more detail Aegeus.

3.2.3 Analysis

This section describes the procedure for analyzing and the results. First, we examined two perspectives: *controller* and *microservice*. This separation of perspectives was important as services are usually defined as a collection of methods or end-points, which are defined in the context of controllers. So, ultimately, we can understand a service as a collection of exposed methods (operations) organized into controllers. Then, for each perspective, we performed both quantitative and qualitative analyses. Finally, we separated those perspectives to understand the difference between analyzing the microservice and its controller.

Controller Perspective

To quantitatively analyze controllers, we sequentially analyzed the versions of each controller for each metric. Then, we removed controllers with no changes in the target metrics during the observed frame interval from the analysis. Finally, we plotted a sequence of run charts with the cohesion metrics across the releases of each controller. We visually inspected each chart for peaks and valleys to identify potential degradation or improvement. Figure 3.3 illustrates a *peak* and a *valley*.

To qualitatively analyze controllers, we manually examined the source code for those identified changes to validate and explain the behavior of the metrics. For example, if the metric SIDC1 has a value of 1, and after a new version, its value changes to 0.5, we looked into this release to determine what sort of changes were implemented.

Microservice Perspective

This analysis is based on the premise that a microservice can be viewed as a collection of interconnected and semantically cohesive controllers where each controller exposes a set of objects and operations. By considering the overall set of controllers and their associated entities and operations, it is possible to evaluate the cohesion of the microservice as a

[3]Aegeus, the King of Athens, is noted for being the father (or not) of Theseus, the fearless hero who defeated the Minotaur in the Labyrinth of Crete. Upon Theseus' return to Athens, Aegeus believed he had lost his son and threw himself into the sea in sorrow. This event gave the name to the Aegean Sea. Aegeus' tale is marked by themes of destiny, courage, and the disastrous consequences of misinterpretation.

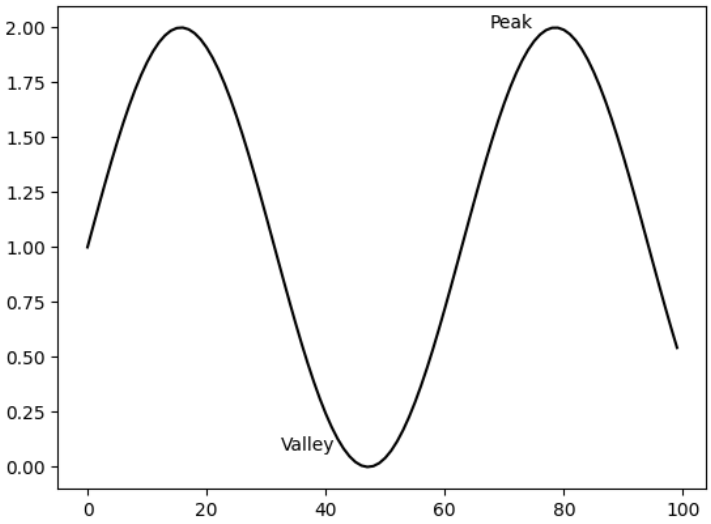


Figure 3.3: Peak and Valley Example

single module. We used this perspective to assess how objects and operations within a microservice are interrelated and work together toward a common goal.

This analysis aggregates the operations and objects of all publicly available controllers for each metric. For instance, let us consider a microservice denoted as m, comprising two controllers, namely c_1 and c_2. In turn, c_1 contains an object o_1 and an operation p_1, while c_2 has the objects o_1, o_2, and o_3, and the operation p_2. In the analysis of the microservice m, the objects o_1, o_2, and o_3, as well as the operations p_1 and p_2, are evaluated altogether, and only one value for each metric is calculated.

Chapter 4

Results and Discussion

4.1 Supporting Tool

This section describes the first result of this work: a tool for evaluating cohesion metrics in repositories. Our tool, Aegeus, allows developers to analyze microservices and provides them indicators of the degree of cohesion in their microservices using the selected metrics (*SIDC1*, $LOC_{message}$, and *SIIC*) for REST controllers in a Git repository.

To use Aegeus, users must provide four input parameters: i) the Git repository address of the microservice, ii) the path address to service controllers, iii) the technology (currently limited to Java Spring MVC framework, annotated with *@RestController*), and iv) the number of releases to analyze (with the option to analyze only the latest version by providing a value of 1).

Figure 4.1 presents the Aegeus architecture, based on the Pipes-and-Filters style, in which a component is triggered immediately after the previous one finishes. The architecture is divided into two major components: *ControllerRetriever* and *MetricCalculator*.

The *ControllerRetriever* detects and retrieves controllers and is composed of two subcomponents: *ReleaseRetriever* and *ControllerFinder*. *ReleaseRetriever* downloads releases from the Git repository and sorts them using the semantic versioning pattern commonly used with Git tags. Then, *ControllerFinder* identifies REST Controllers for each version.

The *MetricCalculator* calculates the metrics for each controller identified by the previous component. It comprises three components: the *Reader*, the *Calculator*, and the *Exporter*. The *Reader* reads a controller file and parses it into an *IServiceDescriptor* instance. The *Calculator* receives the instance and calculates the metrics, *IMetricResult*. Finally, the *Exporter* writes the results into a CSV file.

The tool is available on GitHub[1] and is implemented using Java, Python, and Shell to support the select metrics. It supports Spring MVC applications with an extensible interface for other application frameworks.

[1] `https://github.com/mateusgabi/aegeus`

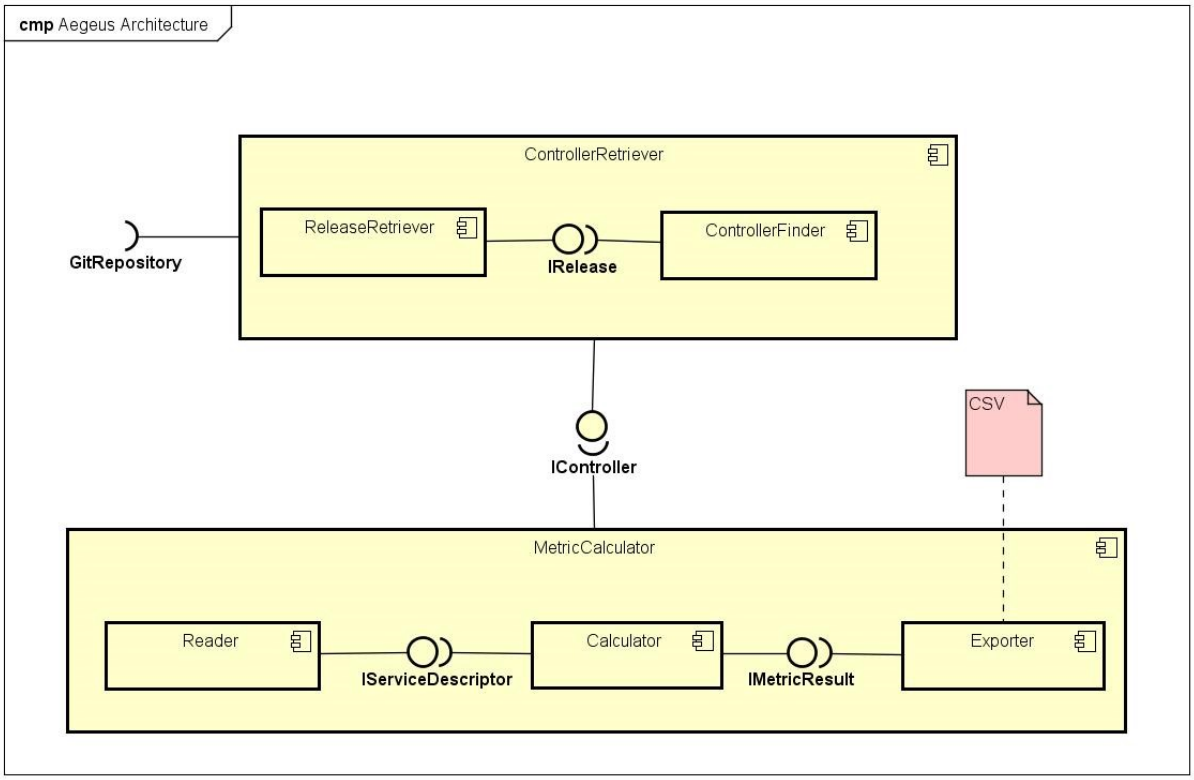


Figure 4.1: Aegeus Architecture

4.2 The Controller Perspective

This section provides an overview of the outcomes of each microservice from the controllers' perspective. Our study focused on nine Spinnaker application microservices, including 138 controllers and a total of 776 releases. To assess the cohesion of each microservice, we conducted both quantitative and qualitative analyses from both the controller and the microservice viewpoint.

From Table 4.1 to Table 4.8, we present an overview of the quantitative results obtained for main controllers for each microservices. All these controllers are present from the first measured release until the latest one.

We have observed a decrease in cohesion for all three metrics in the *Concourse* controller of the *Gate* microservice and in the *Artifact* controller of the *Igor* microservice. However, other controllers exhibited varying results in the metrics for interface (SIDC1 and $LoC_{message}$) and for implementation (SIIC). For instance, the *GoogleCloudBuild* controller's interface metrics showed increased cohesion for the *Igor* microservice, while the implementation metric suggested reduced cohesion. This outcome does not imply a decrease in the cohesion of the entire controller but only in their interface implementation, which can be improved through refactoring to enhance cohesion and maintain the controller's sustainability.

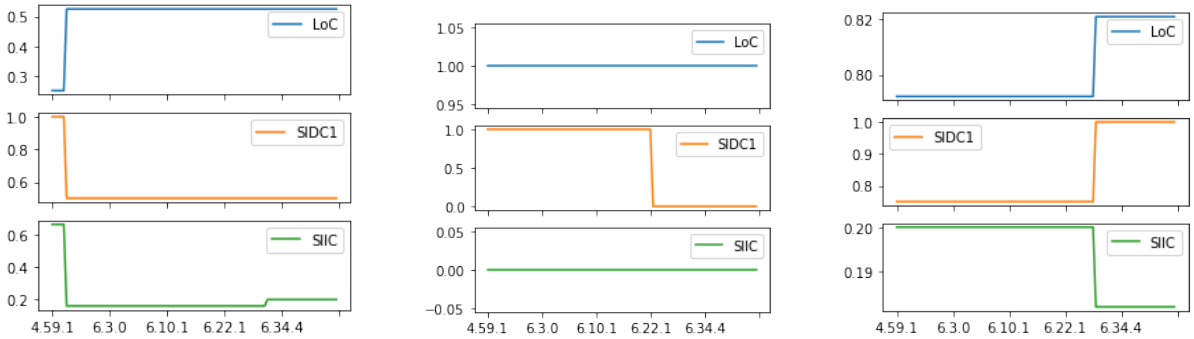
- **Controllers of the Gate Microservice**. The evolution of the Concourse, GoogleCloudBuild, and V2PipelineTemplates controllers from version 4.59 to 6.51 is illustrated in Figure 4.2. The $LoC_{message}$ metric indicates a lack of cohesion when its

value is close to 1 and high cohesion when it is close to 0. In the case of SIDC1 and SIIC, a value closer to 1 indicates lower cohesion of the microservice.

All metrics in the *Concourse* controller exhibited a decrease in cohesion. This could be attributed to the creation of several operations that lacked common output types and architectural elements, such as classes and functions in their implementation.

Regarding the *GoogleCloudBuild* controller, we observed that the metrics $LoC_{message}$ and SIIC remained stable, indicating low cohesion. However, there was a considerable decrease in the SIDC metric, suggesting the new operations added did not share common input types.

In the case of the *V2PipelineTemplates* controller, the $LoC_{message}$ metric decreased by 0.02, and the SIIC metric decreased by 0.02, which can be considered stable results. However, SIDC1 improved to its maximum value of 1. This improvement could be explained by the possibility that we captured a transitional moment when a new operation was being created to replace another one in the future (see Figure 4.3). The other 26 controllers remained stable throughout the analyzed period.



(a) Concourse (b) GoogleCloudBuild (c) V2PipelineTemplates

Figure 4.2: Gate Controllers Evolution

- **Controllers of the Orca Microservice**. Figure 4.4 shows the evolution of this controller from version 7.43.1 to version 8.19.0. The figure indicates the interface cohesion for SIDC1 decreased while the cohesion in $LoC_{message}$ increased. Furthermore, there was a constant lack of cohesion in the implementation (SIIC) throughout the analyzed period.

  Figure 4.5 presents the observed changes in the source code of the *Capabilities* controller across releases. A new operation named *getExpressionCapabilities* was added, introducing a new attribute called *orcaCapabilities*. However, other operations did not share this attribute, resulting in an SIIC value equal to zero.

  The other eight controllers remained stable.

- **Controllers in the Clouddriver Microservice**

  Figure 4.6 illustrates the evolution of the *EcsCluster*, *Function*, and *Manifest* controllers from version 5.23 to version 5.75.

```
71  +  // TODO([redacted]): Deprecated. Will be replaced with /versions endpoint starting with 1.19.
72     @ApiOperation(value = "(ALPHA) List pipeline templates.", response = List.class)
73     @RequestMapping(method = RequestMethod.GET)
74     public Collection<Map> list(@RequestParam(required = false) List<String> scopes) {
75       return v2PipelineTemplateService.findByScope(scopes);
76     }
77
78  +  @ApiOperation(value = "List pipeline templates with versions", response = Map.class)
79  +  @RequestMapping(value = "/versions", method = RequestMethod.GET)
80  +  public Map<String, List<Map>> listVersions(@RequestParam(required = false) List<String> scopes) {
81  +    return v2PipelineTemplateService.findVersionsByScope(scopes);
82  +  }
83  +
84     @ApiOperation(value = "(ALPHA) Plan a pipeline template configuration.", response = HashMap.class)
85     @RequestMapping(value = "/plan", method = RequestMethod.POST)
86     public Map<String, Object> plan(@RequestBody Map<String, Object> pipeline) {
87       return v2PipelineTemplateService.plan(pipeline);
88     }
89
90     @ApiOperation(value = "(ALPHA) Create a pipeline template.", response = HashMap.class)
91     @RequestMapping(value = "/create", method = RequestMethod.POST)
92     @ResponseStatus(value = HttpStatus.ACCEPTED)
93     public Map create(
94         @RequestParam(value = "tag", required = false) String tag,
95         @RequestBody Map<String, Object> pipelineTemplate) {
96       validateSchema(pipelineTemplate);
97       Map<String, Object> operation = makeCreateOp(pipelineTemplate, tag);
98       return taskService.createAndWaitForCompletion(operation);
99     }
100
```

Figure 4.3: V2PipelineTemplatesController code changes.

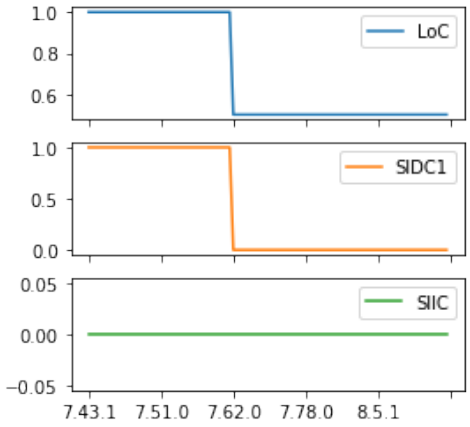
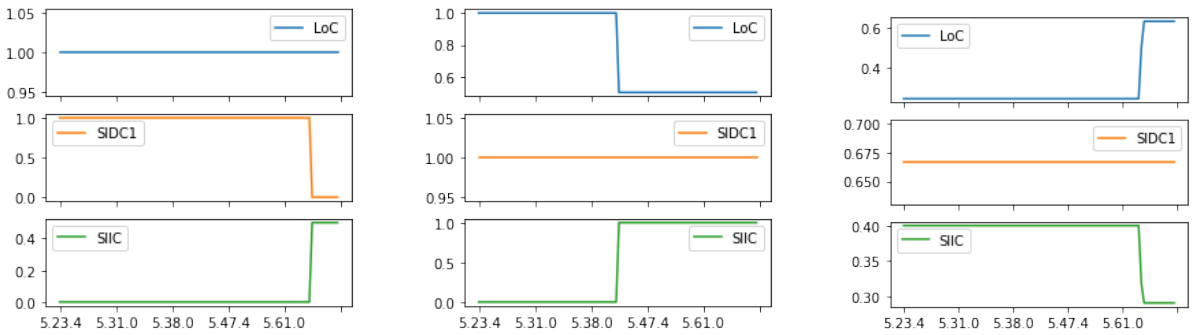


Figure 4.4: Capabilities controller from Orca

```
40      @GetMapping("/capabilities/deploymentMonitors")
41      public List<DeploymentMonitorDefinition> getDeploymentMonitors() {
-         if (monitoredDeployConfigurationProperties == null) {
42  +     if (deploymentMonitorCapabilities == null) {
43            return Collections.emptyList();
44        }
45
-       return monitoredDeployConfigurationProperties.getDeploymentMonitors().stream()
-           .map(DeploymentMonitorDefinition::new)
-           .collect(Collectors.toList());
46  +     return deploymentMonitorCapabilities.getDeploymentMonitors();
47  +   }
48  +
49  +   @GetMapping("/capabilities/expressions")
50  +   public ExpressionCapabilityResult getExpressionCapabilities() {
51  +     return orcaCapabilities.getExpressionCapabilities();
```

Figure 4.5: CapabilitiesController code changes.

The metrics reveal a decrease in cohesion in the *EcsCluster* controller. In contrast, the SIIC metric for the *Function* controller increased, indicating an improvement in implementation cohesion. For the *Manifest* controller, we observed an increase in interface cohesion and a slight decrease in implementation cohesion. This can be attributed to creating operations that share output types in the *Function* controller while using non-shared types in the *Manifest* controller. The other 35 controllers remained stable.



(a) EcsCluster (b) Function (c) Manifest

Figure 4.6: Clouddriver Controllers Evolution

- **Controllers of the Igor Microservice**

  Figure 4.7 displays the evolution of several controllers in the Igor microservice from version 2.1 to Version 4.5. It should be noted that some controllers were introduced

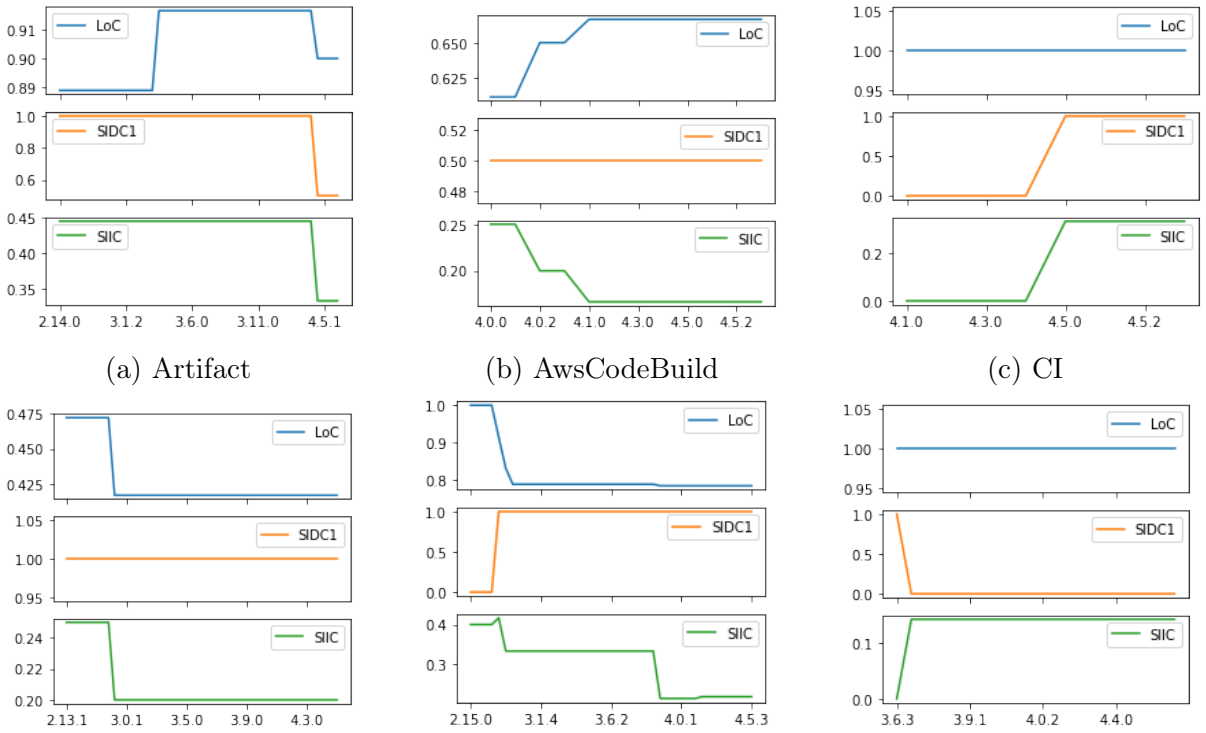
in earlier versions, such as the *AwsCodeBuild* controller in version 4.0, *CI* in version 4.1, *GoogleCloudBuild* in version 2.15, and *Nexus* in version 3.6. Nevertheless, all controllers were analyzed until version 4.5.

Concerning the *Artifact* controller, a new parameter type (*List*) was added to the input, as shown in Figure 4.8. Unfortunately, this change led to a decrease in cohesion.

In the *AwsCodeBuild* controller, cohesion decreased due to adding new operations with new elements and output types. For the *CI* controller, introducing new operations that share elements and input types increased both SIIC and SIDC metrics. Finally, the *Concourse* controller improved interface cohesion by introducing new operations that share output types. However, implementation cohesion decreased because these new operations do not share element types.

The *GoogleCloudBuild* controller's cohesion increased due to the sharing of input and output types. However, implementation cohesion decreased because the elements were not shared with other operations. On the other hand, implementation cohesion was slightly improved in the *Nexus* controller because new operations share elements such as classes, functions, and scripts.

The other four controllers remained stable during the analyzed period.



(a) Artifact (b) AwsCodeBuild (c) CI

(d) Concourse (e) GoogleCloudBuild (f) Nexus

Figure 4.7: Igor Microservice Controllers' Evolution

```
45      @GetMapping("/{provider}/{name}")
46      public List<String> getVersions(
-         @PathVariable("provider") String provider,
-         @PathVariable("name") String name
-       ) {
47  +       @PathVariable("provider") String provider,
48  +       @PathVariable("name") String name,
49  +       @RequestParam(value = "releaseStatus", required = false) List<String> releaseStatuses,
50  +       @RequestParam(value = "type", required = false, defaultValue = "deb") String type) {
51        ArtifactService artifactService = getService(provider);
-       return artifactService.getArtifactVersions(name);
52  +     return artifactService.getArtifactVersions(type, name, releaseStatuses);
53      }
54
-     @GetMapping("/{provider}/{name}/{version}")
-     public Artifact getVersions(
-       @PathVariable("provider") String provider,
-       @PathVariable("name") String name,
-       @PathVariable("version") String version) {
55  +   @GetMapping("/{provider}/{name}/{version:.+}")
56  +   public Artifact getArtifact(
57  +       @PathVariable("provider") String provider,
58  +       @PathVariable("name") String name,
59  +       @PathVariable("version") String version,
60  +       @RequestParam(value = "type", required = false, defaultValue = "deb") String type) {
61        ArtifactService artifactService = getService(provider);
-       return artifactService.getArtifact(name, version);
62  +     return artifactService.getArtifact(type, name, version);
63      }
64
```

Figure 4.8: ArtifactController code changes.

Table 4.1: Clouddriver Microservice Controllers

	LoC			SIDC1			SIIC		
Controller	**1st meas.**	**lst. meas.**	**state**	**1st meas.**	**lst meas.**	**state**	**1st meas.**	**lst meas.**	**state**
GoogleNamedImageLookup	0.940	0.940	stable	0.429	0.429	stable	0.077	0.077	stable
CloudFoundryImage	1.000	1.000	stable	1.000	1.000	stable	0.000	0.000	stable
EcsImages	1.000	1.000	stable	1.000	1.000	stable	0.000	0.000	stable
EcsCloudMetric	1.000	1.000	stable	1.000	1.000	stable	0.000	0.000	stable
EcsCluster	1.000	1.000	stable	1.000	0.000	decrease	0.000	0.500	increase
EcsServiceDiscovery	1.000	1.000	stable	1.000	1.000	stable	0.000	0.000	stable
EcsSecret	1.000	1.000	stable	1.000	1.000	stable	0.000	0.000	stable
EcsServerGroup	1.000	1.000	stable	1.000	1.000	stable	0.000	0.000	stable
Role	1.000	1.000	stable	1.000	1.000	stable	0.000	0.000	stable
CloudFormation	1.000	1.000	stable	0.000	0.000	stable	0.667	0.667	stable
AppengineStorage	1.000	1.000	stable	1.000	1.000	stable	0.000	0.000	stable
Image	1.000	1.000	stable	1.000	1.000	stable	0.000	0.000	stable
ServiceBroker	0.625	0.625	stable	1.000	1.000	stable	1.000	1.000	stable
Manifest	0.250	0.630	decrease	0.667	0.667	stable	0.400	0.291	decrease
EntityTags	1.000	1.000	stable	0.000	0.000	stable	0.333	0.333	stable
Artifact	0.861	0.867	decrease	0.500	0.500	decrease	0.222	0.160	decrease
ServerGroupManager	1.000	1.000	stable	1.000	1.000	stable	0.000	0.000	stable
Function	1.000	0.500	increase	1.000	1.000	stable	0.000	1.000	increase
EntityTagsAdmin	0.583	0.583	stable	1.000	1.000	stable	0.400	0.400	stable

Table 4.2: Fiat Microservice Controllers

	LoC			SIDC1			SIIC		
Controller	**1st meas.**	**lst. meas.**	**state**	**1st meas.**	**lst meas.**	**state**	**1st meas.**	**lst meas.**	**state**
Authorize	0.730	0.730	stable	1.000	1.000	stable	0.074	0.074	stable
Roles	0.583	0.583	stable	0.667	0.667	stable	0.312	0.333	increase

Table 4.3: Front50 Microservice Controllers

	LoC			SIDC1			SIIC		
Controller	**1st meas.**	**lst. meas.**	**state**	**1st meas.**	**lst meas.**	**state**	**1st meas.**	**lst meas.**	**state**
PipelineTemplate	0.859	0.867	decrease	0.800	0.800	stable	0.076	0.071	decrease

Table 4.4: Gate Microservice Controllers

	LoC			SIDC1			SIIC		
Controller	**1st meas.**	**lst. meas.**	**state**	**1st meas.**	**lst meas.**	**state**	**1st meas.**	**lst meas.**	**state**
PubsubSubscription	1.000	1.000	stable	1.000	1.000	stable	0.000	0.000	stable
V2PipelineTemplates	0.792	0.821	decrease	0.750	1.000	increase	0.200	0.182	decrease
EntityTags	0.729	0.729	stable	0.200	0.200	stable	0.364	0.364	stable
PipelineTemplates	0.817	0.817	stable	0.571	0.571	stable	0.150	0.150	stable
StorageAccount	1.000	1.000	stable	1.000	1.000	stable	0.000	0.000	stable
OidcConfig	1.000	1.000	stable	1.000	1.000	stable	1.000	1.000	stable
GoogleCloudBuild	1.000	1.000	stable	1.000	0.000	decrease	0.000	0.000	stable
Artifact	0.736	0.807	decrease	0.500	0.500	stable	0.400	0.333	decrease
Concourse	0.250	0.525	decrease	1.000	0.500	decrease	0.667	0.200	decrease
Executions	0.433	0.433	stable	0.400	0.400	stable	0.556	0.500	decrease
Artifactory	1.000	1.000	stable	1.000	1.000	stable	0.000	0.000	stable
ServerGroupManager	1.000	1.000	stable	1.000	1.000	stable	0.000	0.000	stable
BatchEntityTags	1.000	1.000	stable	1.000	1.000	stable	0.000	0.000	stable
Version	1.000	1.000	stable	1.000	1.000	stable	0.000	0.000	stable
Cleanup	0.733	0.733	stable	1.000	1.000	stable	0.400	0.400	stable
Gremlin	0.000	0.000	stable	1.000	1.000	stable	1.000	1.000	stable

Table 4.5: Halyard Microservice Controllers

	LoC			SIDC1			SIIC		
Controller	**1st meas.**	**lst. meas.**	**state**	**1st meas.**	**lst meas.**	**state**	**1st meas.**	**lst meas.**	**state**
Config	0.500	0.833	decrease	0.000	0.000	stable	1.000	0.333	decrease
Deployment	0.722	0.644	increase	1.000	0.444	decrease	0.400	0.077	decrease
Versions	1.000	0.750	increase	1.000	1.000	stable	0.000	0.500	increase
Account	0.455	0.570	decrease	1.000	1.000	stable	0.400	0.257	decrease
Provider	0.642	0.565	increase	1.000	0.750	decrease	0.485	0.324	decrease
Features	0.625	0.625	stable	0.750	0.750	stable	0.417	0.333	decrease

Table 4.6: Igor Microservice Controllers

	LoC			SIDC1			SIIC		
Controller	**1st meas.**	**lst. meas.**	**state**	**1st meas.**	**lst meas.**	**state**	**1st meas.**	**lst meas.**	**state**
AbstractCommit	0.722	0.806	decrease	0.167	0.250	increase	0.133	0.143	increase
Commit	0.800	0.800	stable	0.500	0.500	stable	0.125	0.111	decrease
Admin	1.000	1.000	stable	1.000	1.000	stable	0.000	0.000	stable

Table 4.7: Orca Microservice Controllers

	LoC			SIDC1			SIIC		
Controller	**1st meas.**	**lst. meas.**	**state**	**1st meas.**	**lst meas.**	**state**	**1st meas.**	**lst meas.**	**state**
V2PipelineTemplate	1.000	1.000	stable	1.000	1.000	stable	0.000	0.000	stable
CorrelatedTasks	0.750	0.750	stable	0.500	0.500	stable	0.167	0.167	stable
Concourse	1.000	1.000	stable	1.000	1.000	stable	0.000	0.000	stable
Capabilities	1.000	0.500	increase	1.000	0.000	decrease	0.000	0.000	stable

Table 4.8: Rosco Microservice Controllers

	LoC			SIDC1			SIIC		
Controller	**1st meas.**	**lst. meas.**	**state**	**1st meas.**	**lst meas.**	**state**	**1st meas.**	**lst meas.**	**state**
V2Bakery	1.000	1.000	stable	1.000	1.000	stable	0.000	0.000	stable

4.3 Results per Metric

4.3.1 $LOC_{message}$ Results

This metric indicates that an interface is highly cohesive when its value is closer to 0 and low cohesion when it is closer to 1.

- *Gate Microservice.* In Figure 4.2, there was a decrease in the cohesion of Concourse and V2PipelineTemplates controllers. In the microservice perspective, Figure 4.9, we can observe an increase in the lack of cohesion.
- *Orca Microservice*, Figure 4.4, Capabilities Controller cohesion was increased. In the microservice perspective, Figure 4.9, we can observe a decrease in the lack of cohesion.
- *Clouddriver Microservice.* In Figure 4.6, there was a decrease in the cohesion of Manifest and an increase in the Function controller. In the microservice perspective, Figure 4.9, we can observe an increase in the lack of cohesion.
- *Igor Microservice.* In Figure 4.7, there was a decrease in the cohesion of AbstractCommit, Artifact, and AwsCodeBuild. Concourse and GoogleCloudBuild increase. In the microservice perspective, Figure 4.9, we can observe an increase in the lack of cohesion.

4.3.2 SIDC1 Results

This metric indicates high cohesion as close to 1 and low cohesion as close to 0.

- *Gate Microservice.* In Figure 4.2, we plot the cohesion assessment of the controllers: Concourse, GoogleCloudBuild, and V2PipelineTemplates. Thus, we can affirm that V2PipelineTemplates cohesion has improved, and Concourse and GoogleCloudBuild decreased.
- *Orca Microservice*, Figure 4.4, have decreased to zero, meaning a huge loss of cohesion in CapabilitiesController.
- *Clouddriver Microservice.* In Figure 4.6, we can affirm that the EcsCluster has decreased to zero, which means a huge loss of cohesion.
- *Igor Microservice.* In Figure 4.7, we can affirm that Artifact and Nexus decreased and AbstractCommit, CI, and GoogleCloudBuild have improved.

4.3.3 SIIC Results

This metric indicates high cohesion when its value is closer to 1 and low cohesion when it is closer to 0.

- *Gate Microservice.* In Figure 4.2, there was a decrease in the cohesion of Concourse and V2PipelineTemplates controllers.

- *Orca Microservice*, Figure 4.4, Capabilities Controller cohesion was still lacking of cohesion.
- *Clouddriver Microservice.* In Figure 4.6, there was a decrease in the cohesion of Manifest and an increase in the Function and EcsCluster controller.
- *Igor Microservice.* In Figure 4.7, there was a decrease in the cohesion of Artifact, AwsCodeBuild, Concourse, and GoogleCloudBuild. AbstractCommit, CI, and Nexus increase.

In general, controllers' cohesion is stable under the perspective of the selected metrics.

4.4 The Microservice Perspective

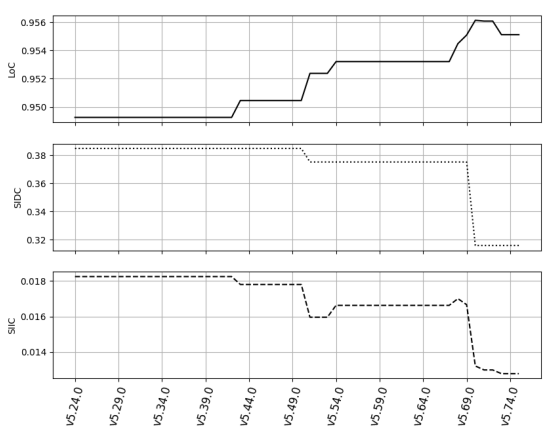
This section provides an overview of the results obtained from analyzing Spinnaker's microservices, focused on examining the cohesion metrics for each microservice.

The idea of emergent properties in systems engineering [11], where the whole entity is more than the sum of its parts, is relevant in cohesion analysis in software applications. While studying the individual parts of a system, such as controllers in a microservice-based application, it is important not to overlook the context and environment in which these parts operate. The concept of emergent properties highlights the need to consider the cohesion of the application as a whole rather than focusing solely on the cohesion of its constituent microservices.

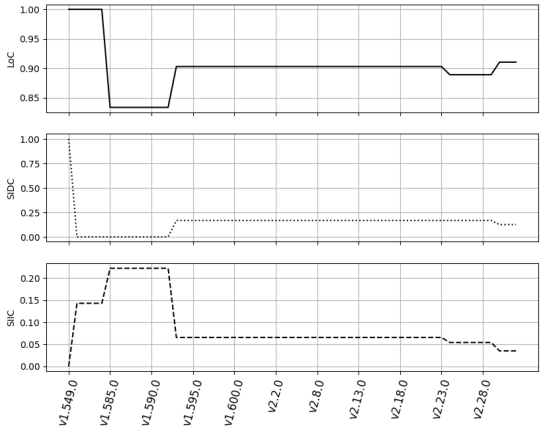
In microservices, the idea of layered structures, as described by Peter Checkland [11], can be applied. Microservices can be seen as autonomous parts within a larger whole, similar to departments within a university or subsystems within a larger system. These microservices exhibit emergent properties, and studying their cohesion can provide valuable insights into the overall system behavior.

The metrics defined for studying cohesion in Service-Oriented Architecture (SOA) applications, such as controllers in the case of microservices, can still be applicable in the context of Microservice Architecture (MSA). However, it is important to consider the granularity of the cohesion in this study. The results indicate that studying the cohesion of an MSA application at the microservice level, rather than individual controllers, using the metrics defined for SOA applications can provide meaningful insights into the system's behavior and maintainability. For example, in Cloudriver Microservice (Table 4.1), from 57 assessments of cohesion (19 controllers and 3 metrics), only 6 assessments in 3 controllers were decreasing. In the first analysis, due to a large number of controllers stables or increased cohesion, 16 of 19 in total, the software engineer may be guided to a *misinterpretation of the stability* in the cohesion while the measurement of the microservice shows a *consensus for the decrease* in cohesion (Table 4.9). In summary, the result indicates that controllers remain stable but do not contribute to the single responsibility of a microservice.

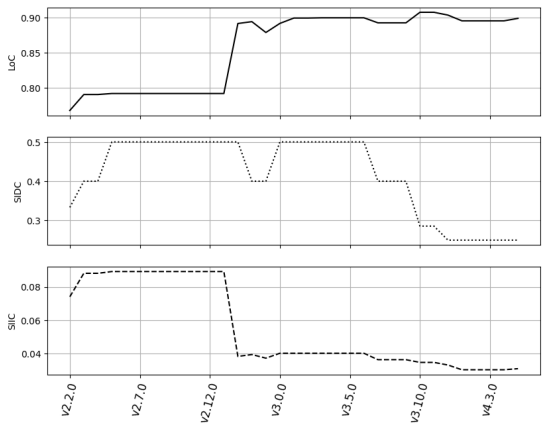
Notably, one meaningful observation is the general decay in cohesion across the microservices evolution (Table 4.9), although the cohesion assessment in the controller perspective remains stable (Table 4.1, 4.2, 4.4, 4.7, 4.8). This decay is only evident



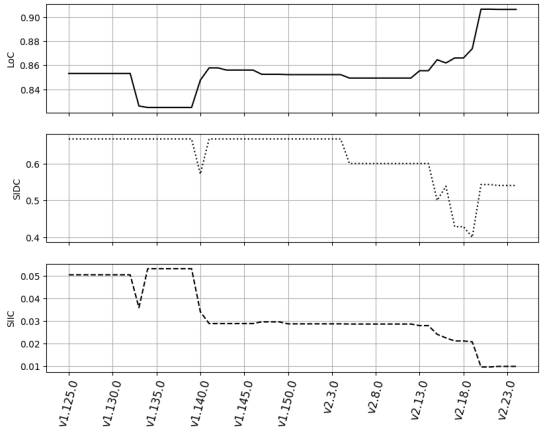
(a) Clouddriver



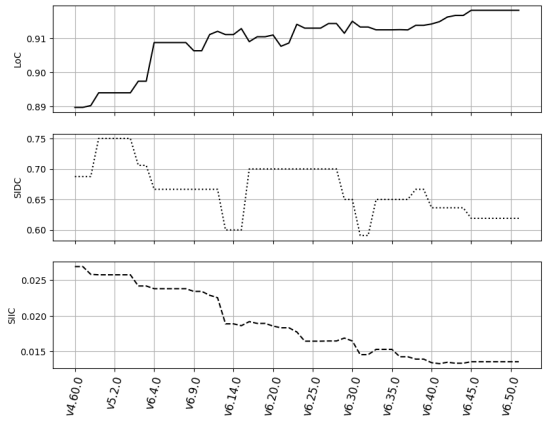
(b) Echo



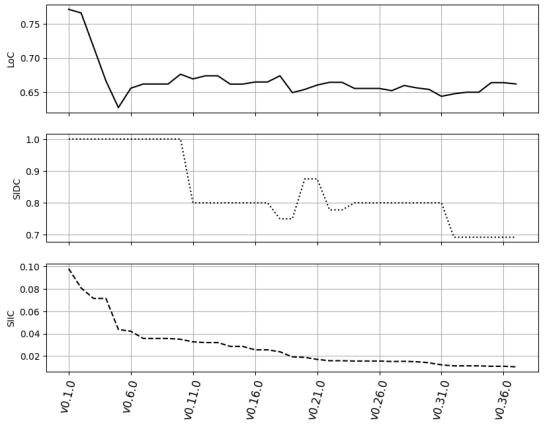
(c) Igor



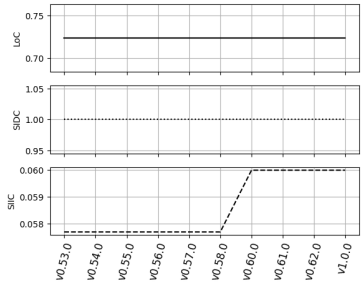
(d) Front50



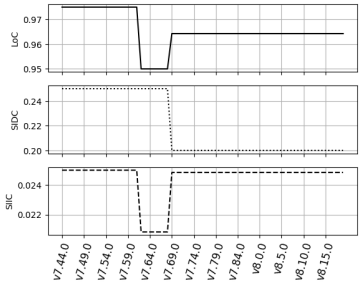
(e) Gate



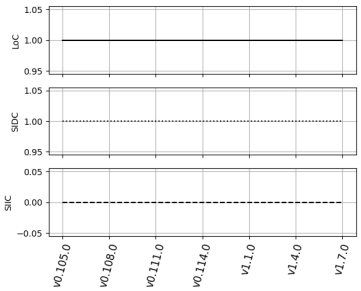
(f) Halyard



(g) Fiat



(h) Orca



(i) Rosco

Figure 4.9: Microservices' Cohesion Evolution

Table 4.9: Cohesion Assessment per Microservice

Microservices	**Metric Result**			**Is consensus?**
	LoC	SIDC1	SIIC	
Clouddriver	decrease	decrease	decrease	yes
Echo	increase	increase	decrease	no
Fiat	stable	stable	increase	no
Front50	decrease	decrease	decrease	yes
Gate	decrease	decrease	decrease	yes
Halyard	increase	decrease	decrease	no
Igor	decrease	decrease	decrease	yes
Orca	increase	decrease	stable	no
Rosco	stable	stable	stable	yes

when considering the cohesion of the entire microservice, indicating a decline in its overall quality. The charts corresponding to each microservice demonstrate this trend.

Table 4.9 and Figure 4.9 aggregates the results, and we can observe: a general and consensual decrease of cohesion in Clouddriver, Front50, Gate, and Igor considering all metrics; For Echo, the interface increased, but the implementation decreased; For Fiat, the interface stability and implementation increased; Halyard and Orca had mixed results; and, for Rosco, everything remains stable. The reason why each microservice decreased is described in Section 4.2.

4.5 Controller vs. Microservice Perspective

This section discusses the results found in Section 4.2 and Section 4.4, respectively, for Controller and Microservice perspectives. We evaluated the metrics from each controller's perspective to begin our analysis. We examined each controller separately, as this is the prevalent method for measuring cohesion in service-oriented applications. Essentially, we determined how well each controller fulfilled its responsibilities. Following this, we assessed the cohesion of each microservice by considering it as a "single controller". It allowed us to measure how effectively the entire microservice fulfilled its designated responsibility.

In our initial analysis, we observed that controllers exhibit a stable level of cohesion due to their singular responsibility. However, in our second analysis, where we evaluated the entire microservice as a single controller, we noted a decrease in cohesion (as seen in Table 4.9). This occurred because additional features were added to the application without introducing new microservices, resulting in larger services, taking on more responsibilities than before, and consequent decay in cohesion.

One important insight is that creating a new microservice for a simple feature may be more costly than adding a method to the service layer. This high cost of creating small features could hinder the creation of new microservices. Initially, the negative impact may not be noticeable since a single method does not appear to be problematic. However, if this occurs repeatedly, it can gradually degrade software maintainability.

A second insight is that software practitioners should avoid unnecessary coupling. Considering why a new microservice should be introduced if it will only serve one single client is crucial. This kind of decision can significantly impact the future of the application. Introducing a new microservice that serves only one client can lead to unnecessary complexity and dependencies within the application, making it more difficult to operate, maintain, and scale. Considering both benefits and drawbacks is essential before introducing a new microservice.

In their research, Curtis and Walz [14] propose a layered behavioral model for software development. This model takes into account individual, team, and organizational factors. At the individual level, software engineering psychology focuses on the problem to be solved. At the team and project levels, team dynamics, structures, and social factors are crucial to success. Both individual skills and effective group communication, collaboration, and coordination are essential. At the highest level, organizational behaviors determine a company's actions and response to the business environment.

At the individual level in Spinnaker, the task may be developed using top-notch patterns and tools to support the delivery of new versions. At the team and project level, there may be cultural differences among the software engineers from different companies involved in the Spinnaker project, leading to potential conflicts that may need to be resolved as the project progresses. These conflicts could potentially be reflected in the project's source code, resulting in maintainability issues, particularly in terms of cohesion.

4.6 Threats to validity

This assessment is limited to the metrics set and the quantitative and qualitative analysis. In this way, this Section discusses threats to our observations, possibly impacting the conclusions, under a pragmatist worldview [29].

4.6.1 Internal Validity

Internal Validity refers to factors affecting the cause-and-effect relationships for metrics assessment and changes in source code. The authors performed a systematic and openly available evaluation of the metrics and the quantitative analysis using scripts developed by the authors. We analyzed the source code qualitatively to explain the quantitative results successfully, mitigating this threat. External researchers may review to corroborate the results.

4.6.2 External Validity

External Validity refers to which the results can be generalized to other applications. The results of this research cannot be generalized. Each microservice application can be developed using different design patterns, data structures, and other technologies. Therefore, we intend to allow practitioners to compare our findings and discuss the applicability of our methodology to their applications.

4.6.3 Construct Validity

Construct Validity refers to the investigated concepts and their operational measures according to the research goals. The set of cohesion metrics was collected as defined in the literature. According to the results, those metrics provide a good perspective of cohesion in the microservice context.

4.6.4 Reliability

Reliability concerns the extent to which the data and the analysis depend on the specific researchers [29]. The metrics were assessed using a systematic tool available on Github [2], the case analysis is entirely independent of the authors and publicly available on Github. Furthermore, the code was analyzed quantitatively using the *compare* feature available for free on GitHub. Thus, researchers can perform the same quantitative and qualitative observations using procedures, tools, and source code.

[2]Available online: `https://doi.org/10.5281/zenodo.6897830`

Chapter 5

Conclusion

This study conducted a comprehensive analysis to assess the feasibility of utilizing the cohesion metrics proposed in the literature for Service Oriented Architecture (SOA) applications in real-world Microservice Architecture (MSA) applications.

To do so, we designed a method to evaluate SOA metrics in a real-world MSA application. This method consists of five steps. The first step was conducting a literature review to find metrics used to measure cohesion in Microservice Architecture (MSA) applications. The selected metrics were: Lack of Message-level Cohesion ($LoC_{message}$), Service Interface Data Cohesion (SIDC1), and Service Interface Implementation Cohesion (SIIC). $LoC_{message}$ measures the cohesion of an interface given its input and output. $SIDC1$ quantifies the cohesion of a service based on the cohesiveness of the operations exposed in its interface, which means operations share the same type of input parameter. Finally, $SIIC$ quantifies the cohesion of a service based on the cohesiveness in the implementation of the operations.

Second, we developed a tool called Aegeus to help developers analyze microservices and provide them with an indicator of the degree of cohesion in their microservices. Third, we selected the repository to be mined. We chose Spinnaker, an open-source Microservice Architecture. Spinnaker is a multi-cloud continuous delivery platform with a large code base of thousands of historical changes and releases. Fourth, we mined Spinnaker repositories. We mined Spinnaker's microservices: Clouddriver, Echo, Fiat, Front50, Gate, Halyard, Igor, Kayenta, Orca, and Rosco. These microservices have more than four thousand releases together. Finally, we evaluated the latest stable versions of every controller during a frame interval using the metrics.

Finally, the quantitative and qualitative analysis showed the Spinnaker project is stable primarily in terms of cohesion because from 138 controllers, 124 controllers remain stable, and only 14 significantly change the values of the metrics.

The metrics indicate a decrease in cohesion when some design and implementation issues happen: (i) new operations or parameter types are added into the interface; (ii) their operation implementations change without sharing internal elements (attributes, methods, functions, etc.), (iii) use of meaningless types in the interface (`Map` and `Object` from Java API). On the other hand, if a new operation shares input and output types, the cohesion increases, which means the controller elements belong together. However, some metrics can be reformulated or added to improve some inherent characteristics of

MSA applications.

These results have shown that it is possible to visualize the evolution of cohesion in microservices architectures using the selected SOA metrics. Our methodology supports the monitoring and analysis of cohesion during microservices' evolution, helping software teams track maintainability issues during the whole software development life cycle.

Also, the Ageus tool can work as a valuable resource for evaluating the level of cohesion in microservices. The tool automatically collects the metrics for each REST controller, making it easier for developers to find improvements to increase cohesion. Furthermore, the tool supports multiple application frameworks and can be used in various contexts.

We understand that methods and tools to support the monitoring and analysis of cohesion during the evolution of microservices are required so that software teams can easily track maintainability issues as soon as they appear in the software life cycle.

The answer for RQ1 is positive, i.e., the cohesion metrics SIDC1, $LOC_{message}$, and SIIC for SOA applications can be applied in MSA applications, as we could verify the evolution of cohesion inside the Spinnaker. Furthermore, although there are some differences between SOA and MSA, the analyzed metrics use common elements in both (i.e., interfaces, operations, input and output parameters, classes, and functions), making it possible to apply them in MSA applications. Also, it is possible to reformulate or add other metrics to better attend to some inherent characteristics of MSA applications.

Our observations show the lack of sharing new elements in past implementations of operations, inputs, or outputs is the most common problem. This leads to a rapid decrease in cohesion, as internal elements within a module are not shared to achieve a single goal. In conclusion, we can affirm that SOA metrics, specifically the SIDC1, $LOC_{message}$, and SIIC metrics, can be applied to MSA applications.

Regarding applying these metrics in the controller or the microservice, we found that controllers generally exhibit a stable level of cohesion, owing to their specific responsibilities. However, when evaluating the entire microservice as a single controller, cohesion may decrease due to adding new features without introducing new microservices. This highlights the importance of carefully managing microservice boundaries and responsibilities to maintain cohesion for microservice.

Considering the layered behavioral model proposed by Curtis and Walz [14], we recognize the significance of individual, team, and organizational factors in effective software development. Addressing these factors can influence cohesion and overall software quality. The Spinnaker project's involvement with diverse software engineers from various cultures underscores the significance of managing cultural conflicts and their potential impact on the project's evolution and is reflected in the source code.

The results and method presented in this study have several implications for academia and the industry. From an academic perspective, this study contributes to the research on cohesion metrics in the context of Microservice Architecture (MSA) applications. The study expands our understanding of cohesion measurement in modern software architectures by evaluating the applicability of Service Oriented Architecture (SOA) metrics in real-world MSA applications. The method developed for assessing these metrics provides valuable guidance for future research in the field. The study also highlights the need for further research and refinement of cohesion metrics tailored to MSA applications. While

the selected SOA metrics (SIDC1, $LoC_{message}$, and SIIC) showed potential and provided insights into the cohesion dynamics of microservices, there is room for the development of additional metrics that capture the unique characteristics of MSA architectures. This opens up future research to refine existing metrics and propose new ones that better address MSA applications' specific challenges and characteristics.

From an industry perspective, the study offers practical implications for software development teams working with microservices. The Aegeus tool developed in this study allows developers to analyze their microservices' cohesion and identify potential improvement areas. By automatically collecting and calculating the metrics for each REST controller, the tool simplifies the process of evaluating cohesion and facilitates the identification of maintainability issues. The study results indicate that maintaining and improving cohesion in microservices is crucial for ensuring the long-term maintainability and quality of the software. The observed decrease in cohesion when design and implementation issues occur highlights the importance of considering cohesion during the software development life cycle. Software teams can proactively track and manage cohesion by using the metrics in this work and addressing the identified problems, leading to more maintainable microservices.

Overall, the study's findings contribute to academia and the industry by providing insights into cohesion measurement in MSA applications, offering a practical tool for cohesion evaluation, and emphasizing the importance of maintaining and improving cohesion for microservices. In addition, the study sets the stage for further research in this area and guides software development teams seeking to enhance the quality and maintainability of their microservices. We want to emphasize the cohesion measurement in terms of the microservice instead of each controller; observing the cohesion at a microservice level helps the software architect determine whether a microservice has more responsibility than it should, as observed in Spinnaker's microservices.

In future works, we aim to expand our understanding of the scope and boundaries of our methodology by implementing it in various application patterns that impact the decision-making process for architecture. These patterns may include client-side discovery, back-ends for front-ends, service registry, and other microservices decomposition strategies based on business capabilities, subdomains, or teams. Furthermore, it is essential to include the metrics not selected in this work, Lack of Domain-level Cohesion LoC_{domain} and Total Interface Cohesion of a Service (TICS) or others not found in the literature proposed in the future. In this way, to gain a comprehensive perspective on the versatility and efficacy of our approach in diverse scenarios.

Furthermore, it is necessary to explore the connection between cohesion and coupling. Engineers may have been influenced by a fear of increased coupling, which could have impacted their decision-making process. For instance, they may have attempted to reduce coupling by grouping various objects into a single microservice to avoid extra HTTP calls or discovery overhead. Then it is crucial to investigate why coupling is still prioritized despite a decrease in cohesion, as it is possible that the engineers have not yet acknowledged this decline.

In the evolution of microservices, numerous controllers have been developed and subsequently terminated. Examining these temporary controllers may uncover signs of both

advantageous and disadvantageous architectural decisions. Analyzing these "fossils" would enable us to establish metrics that measure instances of premature termination and anticipate or recommend strategies to address them.

Bibliography

[1] Nuha Alshuqayran, Nour Ali, and Roger Evans. A systematic mapping study in microservice architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 44–51. IEEE, 2016.

[2] Daniel RF Apolinário and Breno BN de França. A method for monitoring the coupling evolution of microservice-based architectures. *Journal of the Brazilian Computer Society*, 27(1):1–35, 2021.

[3] Dionysis Athanasopoulos, Apostolos V. Zarras, George Miskos, Valerie Issarny, and Panos Vassiliadis. Cohesion-driven decomposition of service interfaces without access to source code. *IEEE Transactions on Services Computing*, 8(4):550–562, July 2015.

[4] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.

[5] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, page 73–87, New York, NY, USA, 2000. Association for Computing Machinery.

[6] Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. Assuring the evolvability of microservices: Insights into industry practices and challenges. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 546–556, 2019.

[7] Justus Bogner, Stefan Wagner, and Alfred Zimmermann. Automatically measuring the maintainability of service- and microservice-based systems: A literature review. In *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, IWSM Mensura '17, page 107–115, New York, NY, USA, 2017. Association for Computing Machinery.

[8] Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51(3):245–273, 2000.

[9] Frederick P Brooks Jr. *The mythical man-month: essays on software engineering*. Pearson Education, 1995.

[10] Ned Chapin, Joanne E. Hale, Khaled Md. Khan, Juan F. Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30, 2001.

[11] Peter Checkland. *Systems thinking*, pages 45–56. Oxford University Press, 1999.

[12] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. *SIGPLAN Not.*, 26(11):197–211, nov 1991.

[13] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[14] Bill Curtis and Diane Walz. Chapter 4.1 - the psychology of programming in the large: Team and organizational behaviour. In J.-M. Hoc, T.R.G. Green, R. Samurçay, and D.J. Gilmore, editors, *Psychology of Programming*, pages 253–270. Academic Press, London, 1990.

[15] M. Dagpinar and J. H. Jahnke. Predicting maintainability with object-oriented metrics -an empirical comparison. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.*, pages 155–164, Nov 2003.

[16] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham, 2017.

[17] Michael Felderer and Guilherme Horta Travassos. The evolution of empirical methods in software engineering. In *Contemporary Empirical Methods in Software Engineering*, pages 1–24. Springer, 2020.

[18] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123:176–189, 2017.

[19] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 21–30, 2017.

[20] Martin Hitz and Behzad Montazeri. Measuring coupling and cohesion in object-oriented systems. 10 1995.

[21] ISO/IEC. *SO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. ISO/IEC, 2011.

[22] M.M. Lehman, J.F. Ramil, P.D. Wernick, D.E. Perry, and W.M. Turski. Metrics and laws of software evolution-the nineties view. In *Proceedings Fourth International Software Metrics Symposium*, pages 20–32, 1997.

[23] James Lewis and Martin Fowler. Microservices. `https://martinfowler.com/articles/microservices.html`, 2019. Accessed: 2019-11-15.

[24] Mateus Gabi Moreira and Breno Bernard Nicolau De França. Analysis of microservice evolution using cohesion metrics. In *Proceedings of the 16th Brazilian Symposium on Software Components, Architectures, and Reuse*, SBCARS '22, page 40–49, New York, NY, USA, 2022. Association for Computing Machinery.

[25] S Newman and Building Microservices. O'reilly media inc. *Building Microservices*, 2015.

[26] Michael P Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: a research roadmap. *International Journal of Cooperative Information Systems*, 17(02):223–255, 2008.

[27] Mikhail Perepletchikov, Caspar Ryan, and Keith Frampton. Cohesion metrics for predicting maintainability of service-oriented software. In *Seventh International Conference on Quality Software (QSIC 2007)*, pages 328–335, 2007.

[28] Mikhail Perepletchikov, Caspar Ryan, and Zahir Tari. The impact of service cohesion on the analyzability of service-oriented software. *IEEE Transactions on Services Computing*, 3(2):89–103, 2010.

[29] Kai Petersen and Cigdem Gencel. Worldviews, research methods, and their relationship to validity in empirical software engineering research. 2013.

[30] Roger S Pressman and Bruce R Maxim. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Professional, New York, NY, 8 edition, January 2014.

[31] Bingu Shim, Siho Choue, Suntae Kim, and Sooyong Park. A Design Quality Model for Service-Oriented Architecture. *2008 15th Asia-Pacific Software Engineering Conference*, pages 403–410, 2008.

[32] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Syst. J.*, 13(2):115–139, June 1974.

[33] E Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering*, pages 492–497, 1976.

[34] Shmuel Tyszberowicz, Robert Heinrich, Bo Liu, and Zhiming Liu. Identifying Microservices Using Functional Decomposition. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10998 LNCS:50–65, 2018.

[35] M. Vidoni. A systematic process for mining software repositories: Results from a systematic literature review. *Information and Software Technology*, 144:106791, 2022.

[36] Eberhard Wolff. *Microservices: flexible software architecture*. Addison-Wesley Professional, 2016.