Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor Thesis

Practical Assessment of the Architecture Refactoring Helper: A Case Study on an Industrial Commercial Project

Noel Khan Uzunbeycan

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr.-Ing. Steffen Becker
Supervisor:	Sandro Speth, M.Sc, Jonas Frizsch, M.Sc.
Commenced:	November 28, 2024
Completed:	April 28, 2025

Abstract

Context. This thesis pertains to the field of software architecture evolution, focusing on the migration of legacy monolithic applications to microservice-based architectures using a tool-supported, quality-driven method, in particular the Architecture Refactoring Helper [Fri24a].
Problem. Monolithic applications that grow large often decrease developer productivity [VGC+15]. Organizations that want to migrate their monoliths towards microservices previously had no guidance to find a Service Identification Approach that aligns with their demands [Her24; Kno23]3.3.
Objective. This thesis, wants to validate the applicability of the Architecture Refactoring Helper [Fri24a] in constrained real-world environments, through a collaborative case study with the LF Consult GmbH [LF 24], by performing a proof of concept decomposition on a subcomponent of the 3LiterPPS-Server®. Additionally, this thesis suggests improvements for the shortcomings found.
Method. During this thesis, we conducted a scenario-based architecture review with developers and the software architect, as well as a preference review with LF Consult GmbH's [LF 24] Head of Development and Support, to configure the Architecture Refactoring Helper [Fri24a]. Afterward, we used different configurations to query for Service Identification Approaches. After attaining the highest-ranked applicable SIAs, we performed the proof-of-concept decomposition, compared, and evaluated the candidate services. Lastly, we evaluated the Architecture Refactoring Helper, suggested improvements, and discussed threats to validity.
Result. The ARH's [Fri24a] phased methodology systematically narrowed over one hundred SIAs to the Service Cutter [GKGZ16a] approach, yielding reproducible service candidates for the Dispo component in the 3LiterPPS-Server®, while revealing limitations in match aggregation, preference weighting, and UI clarity.
Conclusion. The ARH [Fri24a] proves valuable for structured migration planning under realistic constraints, though enhancements in metric expressiveness, preference exclusion, and weighting, and interface usability are recommended to strengthen decision support.

Kurzfassung

Kontext. Diese Arbeit betrifft das Gebiet der Evolution von Softwarearchitekturen und konzentriert sich auf die Migration monolithischer Altsysteme zu microservice-basierten Architekturen mittels eines toolgestützten, qualitätsgetriebenen Verfahrens, insbesondere des Architecture Refactoring Helper [Fri24a].
Problem. Monolithische Anwendungen, die stark anwachsen, verringern häufig die Produktivität der Entwickler [VGC+15]. Organisationen, die ihre Monolithen zu Microservices migrieren möchten, verfügten bislang über keine Anleitung zur Auswahl eines Service Identification Approach, der ihren Anforderungen entspricht [Her24; Kno23]3.3.
Ziel. Diese Arbeit soll die Anwendbarkeit des Architecture Refactoring Helper [Fri24a] unter realen, eingeschränkten Rahmenbedingungen validieren, indem in einer kollaborativen Fallstudie mit der LF Consult GmbH [LF 24] eine Proof-of-Concept-Dekomposition einer Subkomponente des 3LiterPPS-Server® durchgeführt wird. Zusätzlich werden Verbesserungen für die festgestellten Schwachstellen vorgeschlagen.
Methode. Im Rahmen dieser Arbeit wurden ein szenariobasiertes Architektur-Review mit Entwicklern und dem Softwarearchitekten sowie ein Präferenz-Review mit dem Leiter für Entwicklung und Support der LF Consult GmbH [LF 24] durchgeführt, um den Architecture Refactoring Helper [Fri24a] zu konfigurieren. Anschließend wurden mit unterschiedlichen Konfigurationen Service Identification Approaches abgefragt. Nach der Auswahl der am besten bewerteten anwendbaren SIAs erfolgte die Proof-of-Concept-Dekomposition, der Vergleich und die Evaluierung der Kandidatenservices. Abschließend wurde der Architecture Refactoring Helper evaluiert, Verbesserungsvorschläge formuliert und Bedrohungen der Validität diskutiert.
Ergebnis. Die phasenweise Methodik des ARH [Fri24a] reduzierte systematisch über hundert SIAs auf den Service-Cutter-Ansatz [GKGZ16a] und lieferte reproduzierbare Servicekandidaten für die Dispo-Komponente im 3LiterPPS-Server®, während Einschränkungen bei der Trefferaggregierung, Präferenzgewichtung und Benutzeroberflächenklarheit aufgezeigt wurden.
Schlussfolgerung. Der ARH [Fri24a] erweist sich als wertvolles Instrument für eine strukturierte Migrationsplanung unter realistischen Rahmenbedingungen. Zur Stärkung der Entscheidungsunterstützung werden jedoch Erweiterungen der Metrikausdrucksfähigkeit, Mechanismen zum Ausschluss und zur Gewichtung von Präferenzen sowie Benutzbarkeitsverbesserungen der Benutzeroberfläche empfohlen.

Contents

1 Introduction **1**

2 Foundation **3**

2.1 Architectural Styles . . . 3
2.1.1 Monolithic Architecture . . . 3
2.1.2 Microservice Architecture . . . 4
2.1.3 Modulithic Architecture . . . 4
2.2 Refactoring . . . 4
2.3 Architecture Tradeoff Analysis Method . . . 5
2.4 ASAAM . . . 5
2.5 Quality Attributes . . . 5
2.6 Architecture Refactoring Helper . . . 6
2.7 Application to Migrate . . . 7

3 Related Work **9**

3.1 Literature Research Methodology . . . 9
3.1.1 Previous Studies on the Architecture Refactoring Helper . . . 9
3.1.2 Other Migration Frameworks . . . 9
3.1.3 Service Identification Approaches . . . 10
3.2 Previous Studies on the Architecture Refactoring Helper . . . 10
3.2.1 Migrating Monolithic Architectures to Microservices: A Study on Software Quality Attributes . . . 10
3.2.2 Migration of monolithic applications in microservice-based architectures: A Case Study on a Service/Sales-Application . . . 11
3.2.3 Tool Supported Refactoring of Microservice Architectures: An Industrial Case Study . . . 11
3.3 Other Migration Frameworks . . . 11
3.4 Service Identification Approaches . . . 12
3.4.1 Automated Service Identification . . . 12
3.4.2 Manual Service Identification . . . 12
3.4.3 Analysis Type . . . 13
3.4.4 Input Artifacts . . . 13

4 Methodology **15**

4.1 Architecture Review . . . 15
4.2 Preference Interview . . . 16
4.3 Service Identification Approach Selection . . . 16
4.4 Service Identification . . . 16
4.5 Evaluation . . . 16

5 Architecture Review 19
5.1 Selection of Review Method . . . 19
5.2 Review Setting . . . 20
5.3 Review Steps . . . 21
5.3.1 Quality Attribute Selection . . . 21
5.3.2 Scenario Creation . . . 22
5.3.3 Importance and Difficulty Rating . . . 22
5.3.4 Associated Quality Attributes . . . 22
5.3.5 Scenario-Architecture Evaluation . . . 23
5.4 Discussion of Results . . . 28
5.4.1 Selection of Quality Attributes . . . 28
5.4.2 Scenario Creation . . . 28
5.4.3 Importance and Difficulty . . . 28
5.4.4 Associated Quality Attributes . . . 29
5.4.5 Scenario-Architecture Evaluation . . . 29
5.4.6 Conversations During Break . . . 30

6 Service Identification Approach 31
6.1 Preferences . . . 31
6.2 Different Filtering Queries . . . 32
6.2.1 Differences between Scenario-based and Manually Configured Recommendations . . . 34
6.2.2 Quality Attribute Reevaluation . . . 34
6.2.3 Full Preference Configuration . . . 37
6.2.4 Querying Without Quality Attributes . . . 39
6.3 SIA Selection . . . 40
6.3.1 Preliminary Filtering based on Associated Attributes . . . 40
6.3.2 Assessment of Potentially Viable Service Identification Approaches . . . 41
6.3.3 Choosing Service Identification Approaches . . . 45

7 Service Identification 47
7.1 Steinmetz . . . 47
7.1.1 Service Identification Methodology . . . 47
7.1.2 Execution . . . 48
7.2 Microservices Backlog . . . 48
7.2.1 Service Identification Methodology . . . 49
7.2.2 Exectution . . . 50
7.3 Service Cutter . . . 50
7.3.1 Service Identification Methodology . . . 50
7.3.2 Execution . . . 51
7.3.3 Results . . . 53
7.4 Result Discussion . . . 61

8 Evaluation 65
8.1 Architecture Refactoring Helper . . . 65
8.1.1 Strengths and General Usability . . . 65
8.1.2 Usability Issues and Filtering Mechanics . . . 65

8.1.3 Tool Potential and Recommendations 66
8.2 Organizational Context and Constraints 67
8.3 Threats to Validity 67
8.3.1 Construct Validity 67
8.3.2 Internal Validity 68
8.3.3 External Validity 68
8.3.4 Reliability 68

9 Conclusion **71**
9.1 Summary 71
9.2 Benefits 71
9.3 Limitations 72
9.4 Lessons Learned 73
9.5 Future Work 73

Bibliography **75**
.1 Architecture Refactoring Helper Repository Data 81
.2 Quality Attribute Lists 81

List of Figures

2.1 Homescreen and Overview of a Hosted Instance [Fri24b] of the Architecture Refactoring Helper [FBH+22]. 6
2.2 Top ten Service Identification Approaches with the most *Matches* on an exemplary configuration of the second phase of the Architecture Refactoring Helper [FBH+22]. 7

4.1 Overall methodology workflow 15

5.1 Frequency of attribute selections across all participants. 21
5.2 Utility Tree after focus QA selection 22
5.3 Utility Tree after initial scenario creation 23
5.4 Utility Tree after adding the Analysability attribute 24
5.5 Utility Tree after rating the importance and difficulty. 25
5.6 Utility Tree after associating additional quality attributes 26
5.7 Utility Tree with colored scenarios according to the favored architectural pattern. Scenarios are marked red when favoring the monolithic architectural style, blue for microservices, and grey if they were evaluated to benefit equally from both. . . . 27

6.1 Utility Tree after removing quality attributes with no matching Service Identification Approaches. Quality attributes that have been removed are shown in gray. 36

7.1 Visualization of the Dispo components structure. The used scalable vector graphic is available under [Zen23]. 52
7.2 Visualization of the decomposed Dispo component, generated by the Service Cutter [GKGZ16a], using the Leung clustering algorithm. A detailed JSON representation is available under [Zen23]. 55
7.3 Visualization of the decomposed Dispo component with all coupling criteria ignored, generated by the Service Cutter [GKGZ16a], using the Leung, Chinese-Whispers, or Markov MCL clustering algorithm. No clusters are formed, and every nanoentity is encapsulated separately in one of 122 services. A detailed JSON representation is available under [Zen23]. 56
7.4 Visualization of the decomposed Dispo component, generated by the Service Cutter [GKGZ16a], using the Chinese-Whispers clustering algorithm. A detailed JSON representation is available under [Zen23]. 57
7.5 Visualization of the decomposed Dispo component with all coupling criteria set to their maximum weight, generated by the Service Cutter [GKGZ16a], using Chinese-Whispers clustering algorithm. A detailed JSON representation is available under [Zen23]. 58
7.6 Visualization of the decomposed Dispo component with all coupling criteria weights on default, generated by the Service Cutter [GKGZ16a], using the Markov MCL clustering algorithm. A detailed JSON representation is available under [Zen23]. 59

7.7 Visualization of the decomposed Dispo component, generated by the Service Cutter [GKGZ16a], using the Girvan-Newman clustering algorithm with a requested service quantity of three. A detailed JSON representation is available under [Zen23]. 60
7.8 Visualization of the decomposed Dispo component, generated by the Service Cutter [GKGZ16a], using the Girvan-Newman clustering algorithm with a requested service quantity of six. A detailed JSON representation is available under [Zen23]. 61

List of Tables

5.1 Architecture review agenda 19
5.2 Demographic data of the participants of this architecture review. 20
5.3 Participants and Their Quality Attributes 21

6.1 Included and excluded preferences, selected by the Head of Development and Support. 32
6.2 Complete list of included and excluded preferences. 33
6.3 The 15 highest ranked Service Identification Approaches, ordered by the Tendency metric, for the manually configured quality attributes and scenario-based recommendations. The difference in ranking of the SIAs for the manual query is shown in parentheses. 35
6.4 Matching Service Identification Approaches for single attribute queries 37
6.5 The 15 highest ranked Service Identification Approaches, ordered by the Tendency metric, after removing quality attributes, that have no matching Service Identification Approaches. The difference in the ranking of the unadjusted quality attribute ranking (**Q1**) is shown in parentheses. 38
6.6 The 15 highest ranked Service Identification Approaches, ordered by the overall preference matches (**Q3**). The ranking in the adjusted quality attribute only query (**Q2**) is shown in parentheses. 38
6.7 The 15 highest ranked Service Identification Approaches, ordered by the Tendency metric with all preferences. The difference in ranking, when ordering by overall matches (**Q3**), is shown in parentheses. 39
6.8 The 15 highest ranked Service Identification Approaches, ordered by the overall preference matches (**Q4**). The Quality attributes have been removed. The difference in ranking of these SIAs compared to when querying with quality attributes (**Q3**) is shown in parentheses. The visual and percentage tendency ratings have been omitted due to the lack of quality attributes to measure. 40
6.9 Unfiltered pool of Service Identification Approach, ordered by their Id. 40
6.10 Pool of Service Identification Approach after categorization by needed inputs, grouped by their inclusion state, ordered by their Id. 41
6.11 Pool of Service Identification Approach after categorization by process strategy, grouped by their inclusion state, ordered by their Id. 41
6.12 Pool of Service Identification Approach after classifying Potentially-marked SIAs, grouped by their inclusion state, ordered by their Id. 45
6.13 Listing of selected filters that are explicitly assigned to the Service Identification Approaches. 46

1 Attribute Description 84

1 Introduction

Monolithic applications often become rigid and tightly coupled as they grow, reducing developer productivity [VGC+15], leading to slower development feedback loops and increased maintenance effort, prolonging release cycles. LF Consult GmbH [LF 24], a Stuttgart-based consultancy company specializing in production-planning solutions, maintains the 3LiterPPS-Server®, a Java-based monolithic backend application that has evolved for over 25 years to support lean manufacturing processes. The Architecture Refactoring Helper is a web-based, quality-driven tool that emerged from academic research, which helps architects plan the migration of such legacy monoliths to microservices by guiding users towards the most suitable Service Identification Approaches, based on stakeholder-defined quality attributes and non-functional preferences [Fri24a].

Despite the abundance of SIAs, practitioners lack systematic guidance to select an approach that aligns with their specific quality goals and operational constraints, having to resort to intuition or ad hoc evaluations, risking suboptimal service boundaries and migration strategies [Her24; Kno23]. While previous case studies evaluated the Architecture Refactoring Helper [Her24; Kno23], it was not applied within restricted environments, often found in larger industrial applications.

This thesis investigates the ARH's applicability under real-world constraints by addressing the research questions:
RQ1. "How do real-world constraints limit the applicability of the ARH?"
RQ2. "What practical limitations emerge during the usage of the ARH, and which targeted improvements can strengthen its decision support?"

To answer these questions, we first conducted a scenario-based architecture review, based on Svahnberg and Martensson's architecture evaluation process [SM07], with LF Consult's software architect and developers, and a preference interview with their Head of Development and Support, to configure the ARH's preference settings. We then issued multiple queries to the ARH, refining the quality attribute selection and gathering the highest ranked Service Identification Approaches. Subsequently, we selected the three SIAs, which had the highest preference overlap according to the Architecture Refactoring Helper. Using the Service Cutter SIA [GKGZ16a], we performed a proof-of-concept decomposition, evaluated the resulting candidate services, as well as the ARH.

This study focuses on the ARH's "System Comprehension" and "Strategy Definition" phases. We explicitly exclude the final "Architecture Definition" phase, as well as the implementation of the identified microservices.

The contribution of this study is threefold. First, a validation of the Architecture Refactoring Helper [Fri24a] in a constrained industrial setting, demonstrating its potential in guiding the SIA selection without low-level artifacts. Additionally, A proof-of-concept decomposition for a subcomponent of the 3LiterPPS-Server® was performed. Lastly, concrete recommendations for the improvement of ARH's metric expressiveness, preference handling, and interface usability were proposed.

Thesis Structure

The remainder of this thesis is structured as follows:

Chapter 2 – Foundation: Here, we provide an overview of foundational topics, the framework used, and the application to migrate.

Chapter 3 – Related Work: Within this chapter, we cover previous studies working with the Architecture Refactoring Helper, other migration frameworks, and different Service Identification Approaches.

Chapter 4 – Methodology: This chapter outlines the methodology of this thesis.

Chapter 5 – Architecture Review: In this chapter, we conduct an architecture review and share its outcome and the inputs for the Architecture Refactoring Helper.

Chapter 6 – Service Identification Approach: Here, we go over the filtering preferences and selection process to gather the Service Identification Approaches.

Chapter 7 – Service Identification: This section covers the execution and the outputs of the Service Identification Approaches.

Chapter 8 – Evaluation: Within this chapter, the resulting candidate architectures and the ARH's impact will be evaluated.

Chapter 9 – Conclusion We conclude our thesis summarizing our key finding and their beneficiaries. Discuss limitations and the lessons we learned during this thesis, as well as giving an outlook for future work.

2 Foundation

This chapter provides the groundwork for our thesis. We begin by surveying three architectural styles, monolithic, microservices, and modulithic, and highlight how they compare to each other. Next, we introduce the concept of refactoring, with particular emphasis on architectural refactoring. We then present two complementary analysis methods, ATAM [KKB+98] and ASAAM [Tek04], which employ scenario-based evaluation. Afterward, we introduce the quality attribute taxonomy used in this thesis. Then, we introduce the Architecture Refactoring Helper [Fri24a], the web-based migration guidance tool, which we evaluate in this thesis. Finally, we acquaint the application to migrate, the 3LiterPPS-Server®.

2.1 Architectural Styles

In this section, we examine the architecture styles: monolithic, microservice, and modulithic, and how they organize system components, influence quality attributes, listed in Table 1, and suit different application scenarios.

2.1.1 Monolithic Architecture

The monolithic architecture style describes the simplest software architecture [LF14]. All application components are combined into a single program, stored and deployed in one unit, resulting in a simple deployment process. Furthermore, the low latency in-process calls and the minimal overhead from running in a single runtime yield good Time Behaviour and Resource Utilization. Due to the coupling of components, technical details, like the programming language, must be aligned across the entire system [Men20]. As Fowler noted, it is the "natural way" of designing software and requires little effort to set up and test by developers [LF14]. Regarding scalability, monoliths are usually scaled vertically by provisioning increased resources to a single deployment [LF14]. However, as features are added, the codebase becomes more tightly coupled through internal dependencies, which decreases developer productivity and increases overall complexity [VGC+15].

Monolithic architectures are suitable for small to medium-sized teams building applications with relatively simple domain complexity, where operational simplicity, low latency, and minimal infrastructure overhead are priorities. It is a good option for rapid prototyping or when a straightforward deployment process is needed.

2.1.2 Microservice Architecture

In contrast, the microservice architecture style decomposes an application into a collection of small, independently deployable services [New21]. Each service encapsulates a specific business capability and exposes functionality via lightweight APIs. Because services communicate over standardized protocols, they are technology-agnostic. Teams can choose different languages, frameworks, and data stores per service [LF14; New21]. Microservices are independently deployable, as a change isolated to one service, so long as its public interface remains unchanged, can be built, tested, and released without redeploying the entire application [LF14; New21]. Moreover, each service can be independently scaled to match demands.

The microservice architecture style is particularly suited to large-scale applications characterized by significant variability in load across functional areas and supported by multiple autonomous teams. It is most appropriate in organizations that require independent scaling of discrete capabilities and are prepared to invest in the operational infrastructure, such as container orchestration, service discovery, centralized monitoring, and logging, necessary to manage a distributed services landscape.

2.1.3 Modulithic Architecture

The modulithic, or modular monolith, architecture style combines the simplicity of a monolith with the modular organization of microservices. An application is structured as a set of loosely coupled modules, each encapsulating a distinct business capability and interacting through well-defined, in-process interfaces, eliminating the network overhead typical of microservices, yet deployed as a single unit [Mal24; SL24]. This enables moduliths to retain the Performance of monolithic applications while achieving Modularity. Modules map naturally to Domain-Driven Design bounded contexts, improving separation of concerns and making the codebase easier to understand and evolve [Sil22]. The single artifact is deployed like a traditional monolith, while still supporting independent development, testing, and versioning of each module. Scalability is achieved through vertical scaling of the single runtime, with the option to extract modules into standalone services if demand grows [SL24]. Operational complexity stays low, due to no distributed orchestration, but Maintainability, Analyzability, and Agility are significantly improved compared to a classic monolith [Mal24].

The modulithic architecture style is particularly applicable when one seeks the maintainability and clarity benefits of a service-oriented design, without incurring the operational overhead of distributed systems. It is especially suitable for mid-sized applications that anticipate future growth and potential decomposition into microservices, and for teams that require greater structural discipline while retaining a single deployment artifact.

2.2 Refactoring

Refactoring refers to the process of restructuring a software application to improve its non-functional attributes, while maintaining its functionality [KN10]. There are a variety of refactoring types, including the well-known Code Refactoring, Architectural Refactoring, User Interface Refactoring, Database Refactoring, and many more, but in this thesis, we will mainly focus on architectural refactoring.

2.3 Architecture Tradeoff Analysis Method

The Architecture Tradeoff Analysis Method (ATAM) [KKB+98] analyzes software architectures in a structured way. Its primary focus is assessing how well an architecture candidate meets critical quality attributes and elucidating the trade-offs and risks inherent in architectural decisions. The methodology involves collecting scenarios, requirements, and constraints, as well as describing architectural views. Afterward, the scenarios are simulated on said software architectures to show whether or not the previously defined requirements are met. After ensuring that the proposed architectures satisfy the requirements, an analysis for specific quality attributes, for example, performance or security, is performed. In the last step of ATAM, sensitivities and trade-off points are identified. A measured value for a quality attribute that changes significantly due to a specific alteration within the architecture is sensitive to that change [KKB+98]. If multiple attributes are respectively inversely sensitive to an element of the architecture, this element is considered a trade-off point [KKB+98], as it impedes the maximization of both quality attributes. After completing the steps mentioned above, we either reiterate the ATAM process with an adapted architecture candidate or continue with a more detailed design, if all defined requirements are fulfilled.

2.4 ASAAM

Like ATAM [KKB+98], the Aspectual Software Architecture Analysis Method (ASAAM) [Tek04] utilizes scenarios. Said scenarios are categorized as direct scenarios, which do not require changes to the current architecture. If the architecture needs to be modified, the scenario is indirect. Furthermore, an aspectual scenario is realized within multiple architectural components and cannot be localized in one component [Tek04]. Components are also categorized according to a set of scenarios. Cohesive components perform semantically close scenarios. If a component performs scenarios that are not semantically close but consists of multiple cohesive components that enable the decomposition of said component, it is called a composite. Tangled components perform an aspectual scenario [Tek04]. Ill-defined components are the miscellaneous components. They do not perform semantically close scenarios, cannot be decomposed, or perform aspectual scenarios. After the components have been categorized for the subsets of scenarios they perform, the architecture can be modified if the components have scenarios for which they are aspectual components. Aspectual components can be “detangled” by introducing a new component, an architectural aspect, which performs the aspectual scenario. This method inherently reduces the average amount of functionalities per component, increasing the components’ cohesiveness [Tek04].

2.5 Quality Attributes

Whenever we refer to *Quality Attributes*, or *QAs*, we mean non-functional characteristics beyond the application’s core functionalities. Formulated by Koch in his master’s thesis [Koc22], they are categorized into upper- and lower-level attributes. Lower-level QAs like *Analyzability*, or *Testability*, are clustered by their upper-level QA *Maintainability*. Table 1 summarizes the full set of attributes referenced in this thesis.

2.6 Architecture Refactoring Helper

The Architecture Refactoring Helper [FBH+22] is a web-based tool, that guides users through a three-phase process of migrating monolithic applications towards a microservice architecture.

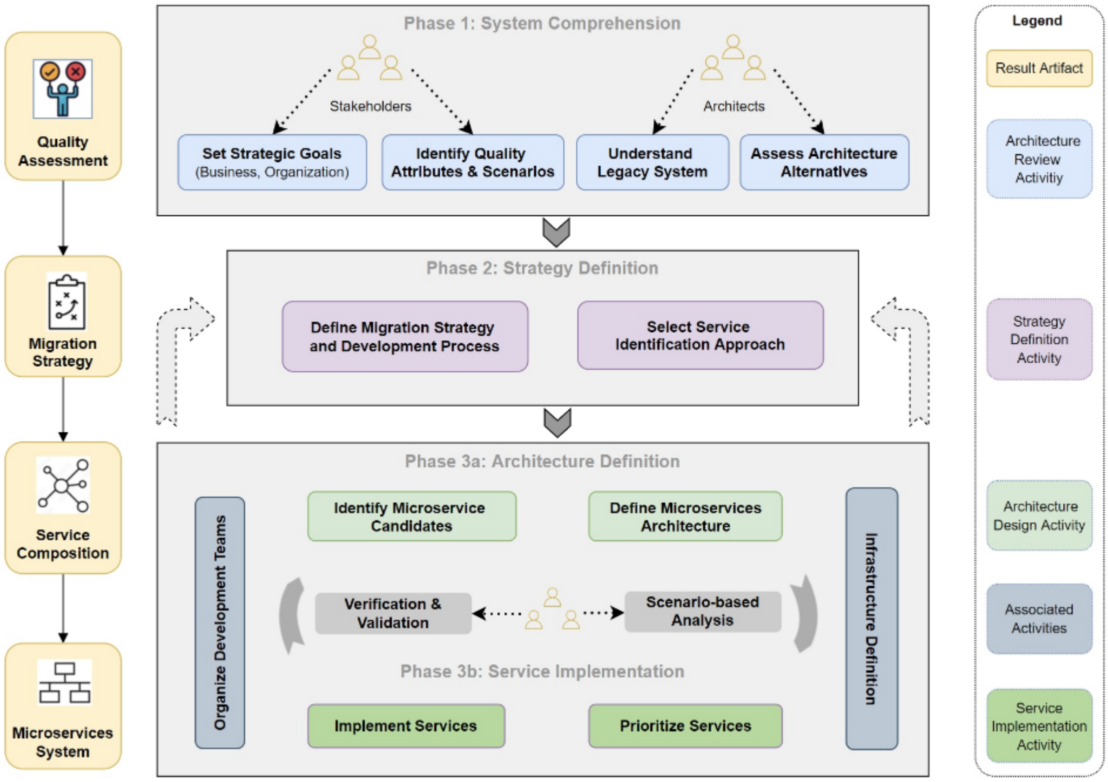

Figure 2.1: Homescreen and Overview of a Hosted Instance [Fri24b] of the Architecture Refactoring Helper [FBH+22].

In the first phase, "System Comprehension", a comprehensive view of the system to be migrated is created. Collaborating with key stakeholders, business and organizational goals for the application are defined, and significant *Quality Attributes* are extracted using a scenario-based architecture analysis method like ATAM [KKB+98] or ASAAM [Tek04]. The gathered scenarios and their corresponding quality attributes are then looked at in more detail in consultation with the system architect, and it is evaluated whether or not the microservice architecture is a fitting architectural style.

During the second phase "Strategy Definition", the previously defined scenarios with their affiliated quality attributes are combined with additional settings regarding *Inputs* used in the SIA, which will not include *Source Code* in this thesis for reasons that will be explained in Section 6.1. Further preferences include the *Process Strategy*, *Levels of automation*, *Output Representation*, and different *Usability preferences*. After the user has expressed his desired preferences for the migration process, a list containing all Service Identification Approaches is displayed. This list is sorted by matching preferences, showcasing the overall best-fitting SIAs with links to their publication at the top.

Another user flow in the second phase is the option not to use pre-entered scenarios and to configure all quality attributes by hand, or display a list of all Service Identification Approaches, or all tools from its underlying repository.

Information				Qualities			Actions
Matches	ID	Title	Authors	QA Matches	SP Matches	Tendency	
16 / 39	2018	Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis	Matias, T. et al.	6 / 10	1 / 3		
14 / 39	2101	A multi-model based microservices identification approach	Daoud, M. et al.	2 / 10	0 / 3		
13 / 39	1805	Discovering Microservices in Enterprise Systems Using a Business Object Containment Heuristic	De Alwis, A.A.C. et al.	2 / 10	1 / 3		
13 / 39	1905	From Monolithic Architecture to Microservices Architecture	De Lauretis, L.	5 / 10	0 / 3		
13 / 39	2117	Microservices backlog – a genetic programming technique for identification and evaluation of microservices from user stories	Vera-Rivera, F.H. et al.	7 / 10	1 / 3		
12 / 39	2012	Steinmetz: Toward automatic decomposition of monolithic software into microservices	Jakob, L. and Oprescu, A.M.	0 / 10	0 / 3		
12 / 39	2013	Towards Identifying Microservice Candidates from Business Rules Implemented in Stored Procedures	Maia, P. et al.	6 / 10	0 / 3		
11 / 39	1804	Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems	De Alwis, A.A.C. et al.	0 / 10	1 / 3		
11 / 39	2201	GreenMicro: Identifying Microservices from Use Cases in Greenfield Development	Bajaj, D. et al.	5 / 10	1 / 3		
11 / 39	2003	Extracting Microservices' Candidates from Monolithic Applications: Interface Analysis and Evaluation Metrics Approach	Al-Debagy, O. and Martinek, P.	3 / 10	1 / 3		

Figure 2.2: Top ten Service Identification Approaches with the most *Matches* on an exemplary configuration of the second phase of the Architecture Refactoring Helper [FBH+22].

After selecting and executing the Service Identification Approach, the user obtained a proposed decomposition for the monolithic application, which in the further course of this thesis we call "service candidates".

The final phase of the Architecture Refactoring Helper "Architecture Definition" revolves around the selection of design patterns and best practices to implement the service candidates, which have been output by the previously selected SIA. Like in the Strategy Definition phase, the scenarios from the first phase are used for filtering. As for this thesis, we will not be looking at this phase, as this will be part of the execution of the migration plan for each service, which is out of scope.

2.7 Application to Migrate

The 3LiterPPS® system is grounded in the principles of lean production management, specifically tailored to meet the demands of small and medium-sized enterprises. Developed to streamline production planning and control, the system emphasizes simplicity and efficiency. Unlike traditional production planning systems that are often rigid and complex, 3LiterPPS® is designed to adapt to the existing organizational structures of a company, enabling more flexible and responsive operations.

At the heart of this system is the 3LiterPPS-Server®, the central backend component responsible for its core functionalities. Developed as a Java-based monolithic application, the 3LiterPPS-Server® has evolved over more than 25 years, incorporating gradual and iterative enhancements to meet the complex demands of modern production environments. This component handles the business logic, manages asynchronous communication through an event queue, and oversees the data models essential to the application's operations. Given its critical role, reducing interdependencies within the 3LiterPPS-Server®, through independent modules or microservices, is vital for improving maintainability and flexibility. Simplifying this monolithic architecture will enhance the ease of adding new functionalities and streamline deployment processes, making the system more adaptable to future technological advancements and business needs. To maintain confidentiality while ensuring reproducibility, we will conduct this study without using the source code as an input parameter for our Service Identification Approaches. This approach allows us to achieve consistent and repeatable results without compromising the sensitive implementations contained within the application.

3 Related Work

Due to the large popularity of the microservices architecture style and the demand for strategies to migrate monolithic applications, a plethora of research has been conducted in this field. This chapter gives a comprehensive overview of the literature related to the Architecture Refactoring Helper [Fri24a], Microservice migration, and our thesis. We begin by outlining our literature research methodology, acquiring previous publications on the ARH, other meta-level migration frameworks, and Service Identification Approaches. The subsequent sections then elaborate on the found research.

3.1 Literature Research Methodology

To ensure a robust and comprehensive foundation for our related work analysis, our research methodology integrates systematic database searches and direct examination of the Architecture Refactoring Helper's internal repository [Zen23]. All literature searches have been conducted with the Google Scholar search engine [Goo25].

3.1.1 Previous Studies on the Architecture Refactoring Helper

To acquire literature on previous studies on the Architecture Refactoring Helper [Fri24a], we used the query string *"Architecture Refactoring Helper" OR "Architektur Refactoring Helper"*, as we knew that there was previous work that was conducted in German. This query returned six results. We excluded three of them, as they revolve around the development and implementation of the Architecture Refactoring Helper [Fri24a] itself, which we covered in the Foundations chapter 2.6. What we were left with were three theses, one being Koch's study on the quality attributes [Koc22], which we employ throughout this thesis, and two studies evaluating the ARH within a case study, similarly to our work.

3.1.2 Other Migration Frameworks

As previous studies evaluating the Architecture Refactoring Helper [Her24; Kno23] mentioned, they could not find comparable migration frameworks that operated on a meta level, guiding the user towards the selection of suitable SIAs, architectural patterns, and best practices. To verify these claims, we searched for publications that propose other migration frameworks with the following query: *(intitle:framework OR intitle:tool OR intitle:helper) AND ("architecture refactoring" OR "service identification framework" OR "migration tool" OR "decomposition strategy") AND ("microservice" OR "monolith" OR "software architecture migration").* The

19 publications, matching this query, either revolved around the ARH itself, proposed Service Identification Approaches, or compared them to each other. Like our predecessors, we could not find comparable frameworks or tools, emphasizing the uniqueness of Fritzsch's ARH [Fri24a].

3.1.3 Service Identification Approaches

The Service Identification Approaches listed in Section 3.4 are drawn from the extensive repository of the Architecture Refactoring Helper and have not been researched through traditional literature search engines. We provide the used repository version under [Zen23]. For the distinct sections, we filtered the available SIAs and selected the first matching SIAs. The different filtering strings were: *"Automatic"*, for Section 3.4.1, *"Manual"*, for Section 3.4.2, *"Static"*, *"Lexical"*, and *"Historic"* for Section 3.4.3, and lastly *"executableInputs": [{"name": "Source Code"*, *"Version Control System"*, *"Human expertise"*, *"API / Interface"*, and *diagram"*.

3.2 Previous Studies on the Architecture Refactoring Helper

In the past few years, the Architecture Refactoring Helper [Fri24a] has been the subject of several empirical investigations demonstrating its applicability and revealing enhancement opportunities. Koch extended the ARH with a quality-driven migration model, mapping microservice-relevant quality attributes to design patterns and validating the approach through a practitioner survey, which confirmed improved usability and decision support [Koc22]. Knodel conducted a hands-on case study with a web-based CRM application, using the ARH's filtering mechanisms to select a strangler-pattern approach and implementing a proof-of-concept PDF service extraction, thereby illustrating its end-to-end utility in a SaaS context [Kno23]. Herrmann focused on the ARH's role in optimizing existing microservice architectures, assessing service granularity and communication model efficiency in an industrial setting, and identifying both strengths and implementation challenges [Her24]. Together, these studies confirm the ARH's versatility across migration and refactoring scenarios and inform our understanding of how quality modeling, tool support, and real-world constraints shape its evolution.

3.2.1 Migrating Monolithic Architectures to Microservices: A Study on Software Quality Attributes

Koch's study from 2022 [Koc22] extended the Architecture Refactoring Helper [Fri24a] by incorporating a quality-driven approach into the migration process from monolithic architectures to microservices. To achieve this, he first conducted a literature review to identify relevant quality attributes for microservices architectures, analyzing their role in migration and their relationships with architectural patterns and best practices. He then developed a quality model that systematically maps these QAs to migration approaches, design patterns, and best practices. After creating this theoretical foundation, Koch integrated the quality model into the Architecture Refactoring Helper [Fri24a], extending its functionality to support practitioners in defining quality goals for a target architecture, selecting migration strategies that optimize these goals, and identifying architectural patterns that align with them. Finally, he conducted a usability evaluation through a survey with four industry practitioners, assessing the tool's effectiveness in guiding migration decisions. The results demonstrated that the integration of the quality model improved the tool's usability and its ability to support migration planning in practice.

3.2.2 Migration of monolithic applications in microservice-based architectures: A Case Study on a Service/Sales-Application

In his master's thesis [Kno23], Knodel performed a case study using the Architecture Refactoring Helper [Fri24a] to migrate a web-based monolithic client-server service/sales application to a microservices architecture. As the author described in his thesis, the application, which was developed by the collaborating organization, L-mobile [L-m25], is oriented towards a Customer-Relationship-Managment-System, and is offered as a Software-as-a-Service Solution, which can be used on its own, or in combination with Enterprise-Resource-Planning Software [Kno23]. To generate scenarios for the ARH, he conducted an architecture evaluation based on the framework proposed by Svahnberg and Martensson [SM07], identifying security, reliability, and functional suitability as the most critical quality attributes, as defined in the ISO/IEC 25010 standard [Int11]. After applying multiple filters to the extensive repository of the Architecture Refactoring Helper [Fri24a], the approach proposed by Li et al. [LML20], which is based on the strangler pattern, was selected for the migration. A total of 40 services were identified and organized into five bounded contexts. Among these, the PDF-generating service was chosen for a proof-of-concept implementation due to its recurring issues and relatively loose coupling with the monolithic system.

3.2.3 Tool Supported Refactoring of Microservice Architectures: An Industrial Case Study

Herrmann conducted a case study [Her24] to assess the effectiveness of the Architecture Refactoring Helper [Fri24a] in optimizing microservice architectures. The study focused on evaluating service granularity and communication model efficiency. Using an adapted version of Svahnberg and Martensson's architecture evaluation framework [SM07], six key quality attributes were prioritized: scalability, modularity, time behavior, deployability, fault tolerance, and recoverability. These attributes were defined based on Koch's work [Koc22]. After filtering various Service Identification Approaches, two leading candidates emerged: Vera-Rivera et al.'s microservices backlog [VPAG21] and Daoud et al.'s multi-model-based microservice identification approach [DEF+21]. While both approaches outlined the use of tools, neither provided an implementation. Due to time constraints and implementation challenges, their executability was evaluated hypothetically. The findings suggest that the Architecture Refactoring Helper has significant potential in guiding the refactoring process, indicating that it could become a valuable tool for both transitioning to and optimizing existing microservice architectures.

3.3 Other Migration Frameworks

While numerous decomposition approaches have been proposed in the literature, like Service Cutter [GKGZ16a], Mono2Micro [KXK+21], or CARGO [NARK22], to name a few, the Architecture Refactoring Helper [Fri24a] stands out due to its unique meta-level operation, which enables the evaluation and comparison of multiple decomposition strategies rather than advocating for a single approach. Preceding studies covering the ARH [Kno23][Her24] further underscore this distinction, as they also found no comparable frameworks that operate at a similar level of abstraction or provide equivalent flexibility in evaluating diverse migration strategies. This gap in the literature highlights the ARH's novelty and its potential to address a critical need for more generalized and adaptable tools in software architecture migration.

3.4 Service Identification Approaches

In recent years, many decomposition procedures have been published that focus on identifying service candidates within monolithic software. Because these approaches vary widely in their defining characteristics, we structured this section to introduce a subset of them along three complementary dimensions, degree of automation, analysis type, and input artifacts, to clarify how each technique discovers candidate microservices within monoliths.

The automation spectrum spans fully manual workflows, where practitioners apply design patterns and expert judgment, to fully automated tools that infer service boundaries from runtime behavior. Service Identification is done through the analysis of the monolith, but the type of analysis can vary. Some approaches statically analyze the invocations and instantiations to infer coupling, while others obtain their data from dynamic runtime data, evaluate historic data, or compare the lexical similarity of entities. Finally, SIAs differ in the representations they consume, from source code and execution logs to architecture diagrams and human expertise; each input artifact shapes the method's applicability, precision, output format, and scalability.

3.4.1 Automated Service Identification

The Mono2Micro approach [KXK+21], developed by IBM [IBM25], automatically decomposes Java applications into microservices. The tool collects runtime execution traces while the application is being used under specific business scenarios. It then analyzes the sequence and frequency of method calls to automatically group classes that consistently interact, resulting in candidate microservices that exhibit high functional cohesion. This process is designed for legacy systems undergoing gradual transformation and minimizes the need for manual intervention, whilst reflecting the operational structure of the application within the identified services. Other automated approaches were published by Abdullah et al. [AIE19] and Zaragoza et al. [ZSS+21].

3.4.2 Manual Service Identification

Ivanov and Tasheva introduced “A Hot Decomposition Procedure” [IT21] for extracting microservices from operational monolithic systems. Their method is structured into seven distinct steps. Initially, the migration benefits are assessed, and bounded contexts are defined in alignment with the underlying business model. The first concrete decomposition step involves creating a separate service for the user interface and subsequently removing it from the monolith, while the remaining services are addressed in the final stages. By employing an extension of the strangler fig pattern [Fow04] and the branch by abstraction pattern [Fow25], functionalities are grouped by business areas and iteratively detached from the monolith. Finally, the procedure incorporates data synchronization mechanisms between the monolith's shared database and the newly formed microservices, along with traffic redirection through an API Gateway to ensure minimal downtime. Notably, this approach is entirely manual and does not utilize any tools. Decomposition strategies that are comparable in their manual nature include Gao et al.'s artificial immune algorithm [GCL+20] and the independent micro-workflow approach proposed by Alaasam et al. [ART21].

3.4.3 Analysis Type

Artifacts used as inputs can be analyzed in various ways to identify service boundaries. Static analysis captures inherent structural dependencies, for instance, the approach presented by Saidani et al. [SOMS19] formulates the extraction of microservices as a multiobjective optimization problem, relying on said structural dependencies such as method invocations and class interactions to assess cohesion and coupling. In a similar vein, CARGO [NARK22] employs a context-sensitive static analysis to construct a comprehensive system dependency graph that captures relationships including call return, data flow, heap dependencies, and transactional interactions, to delineate service boundaries. Lexical analysis provides an additional perspective by examining the semantic aspects of the code. Sun et al. [SBH+24] create API similarity graphs based on textual features, such as identifier naming conventions and documentation, to derive similarity measures between components. This analysis type might detect functional associations that are not evident when statically analyzing applications. Historical analysis further complements these techniques by introducing a temporal dimension to the evaluation of component interactions, analyzing the coupling of components through their joint changes. Löhnerts and Oprescu [LO20] combined all the aforementioned analysis types into one Service Identification Approach. Their tool, Steinmetz [Löh25] visualizes the coupling of classes based on three graphes: a static coupling graph, that group classes based on their instantiations and method calls, a semantic or lexical graph that is built through similarity in natural language, and lastly an evolutionary graph which analyzes the commit history of an application, grouping classes based on shared commits, since according to the SRP [Rob03].

3.4.4 Input Artifacts

To identify microservice candidates, all SIAs need some form of representation of the application that is undergoing a decomposition, but the required artifacts differ between approaches. Many approaches rely on source code as their primary input, often using static analysis to generate candidate services. While Steinmetz [LO20] required source code and version control information, other SIAs utilize different representations. Dinh-Tuan and Beierle [DB22], for example, only require human expertise, while Sun et al. [SBH+24] automatically create a similarity graph based on API descriptions. Another artifact that is used by multiple Service Identification Approaches is diagrams. Such as Alaasam et al. [ART21] using data flow diagrams, class diagrams, used by Park et al. [PKY20], or Li et al. [LXXW22], which additionally use activity diagrams.

4 Methodology

This chapter describes the step-by-step methodology of performing a proof of concept decomposition to evaluate the migration process supported by the Architecture Refactoring Helper [Fri24a] and the resulting service candidates generated. Figure 4.1 summarizes the overall process.

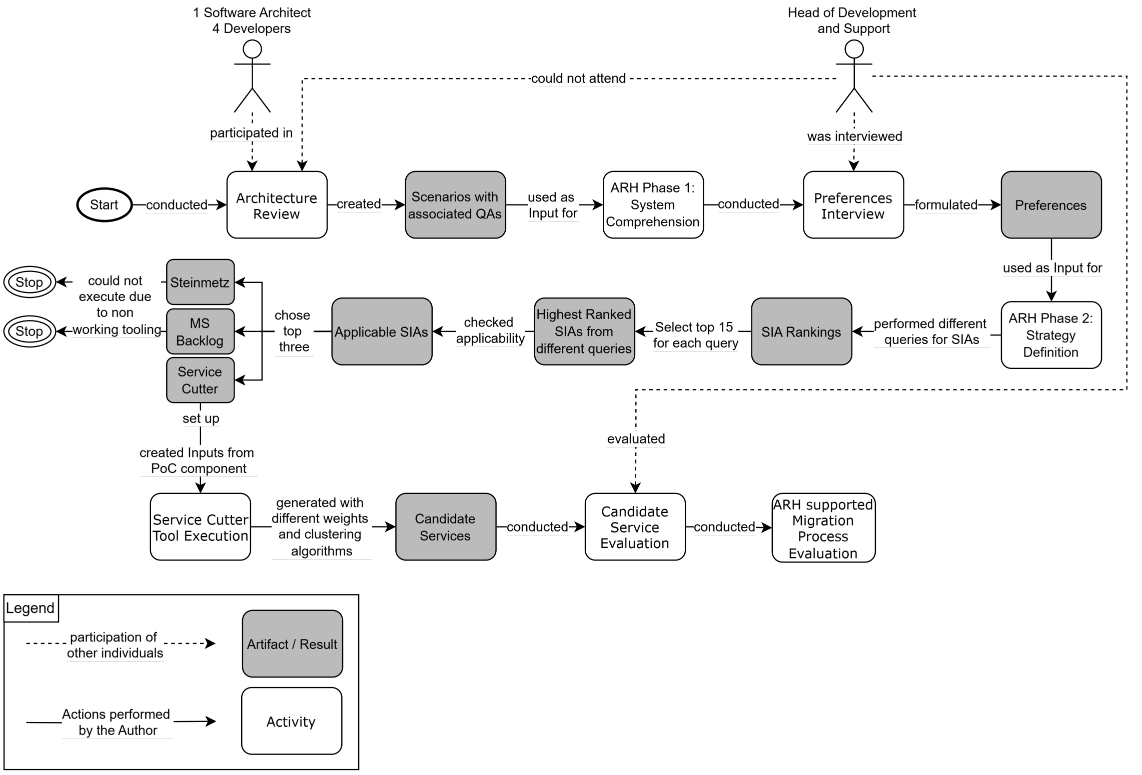

Figure 4.1: Overall workflow of the ARH-supported migration methodology, boxes denote activities, shaded boxes denote artifacts or results, dashed arrows indicate participation of other stakeholders, and continuous arrows mark actions that have been performed by the author.

4.1 Architecture Review

The initial step of our case study was the conduct of an architecture review. Four developers and the software architect participated in the review, which was performed with a modified version of Svahnberg and Martensson's architecture evaluation process [SM07], which itself is a scenario-based process. The output of this review was the creation of scenarios with associated quality attributes, which were then put into the first phase of the Architecture Refactoring Helper [Fri24a], “System Comprehension”.

4.2 Preference Interview

The preference selection in the second phase of the ARH [Fri24a] was based on the preference interview conducted with the Head of Development and Support of the collaborating organization. Fritzsch, the creator of the ARH [Fri24a], advised us to exclude options that are incompatible with the organization’s preferences and include desired options. All non-quality preference options were successively assessed.

4.3 Service Identification Approach Selection

A series of different queries was performed to retrieve and rank Service Identification Approaches. Our first query was configured solely on the quality attributes, which were acquired in the first step. We then reevaluated the selected quality attributes and performed another query, as we found certain QAs that increased the “Matches” metric, without the SIAs being categorized with said attributes. The third query was configured with all configuration options retrieved in the architecture and preference reviews. Lastly, we also queried the ARH’s repository only using non QA preferences.

Afterward, we selected the 15 highest-ranked SIAs for each query, of which we checked the applicability for our use case. From the six remaining Service Identification Approaches, we selected the top three highest ranked ones to perform a proof of concept decomposition on a component within the 3Liter-PPS-Server® application.

4.4 Service Identification

The execution of the Steinmetz [LO20] and Microservices Backlog [VPAG21] SIAs were suspended, as we were unable to set up the required tools, developed by the authors, due to different complications, which we could not overcome during the scope of this thesis.

Based on the Dispo component, we created inputs [GKGZ15b; GKGZ15d; GKGZ15e] for the Service Cutter Tool [GKGZ15a], to then generate service candidates with different implemented clustering algorithms [Bie06; GN02; NG04; RAK07; Van08].

4.5 Evaluation

Subsequently, in collaboration with the Head of Development and Support, we evaluated the different candidate services generated by the Service Cutter [GKGZ16a], by comparing the results to each other, noting suggestions for improvement.

At last, we evaluate the Architecture Refactoring Helper, outlining its strengths in guiding the migration process, addressing areas for improvement, and proposing concrete suggestions. Moreover, we put the impact of the organizational context, influencing this thesis, into perspective before covering the threats to validity.

5 Architecture Review

Different Service Identification Approaches (SIAs) are designed to enhance specific Quality Attributes (QAs). Therefore, identifying and focusing on the QAs that are most important to the organization is a key step in selecting SIAs that align with stakeholder priorities.

To achieve this, we conducted a scenario-based architecture review to gather the relevant QAs and meet the input requirements of the Architecture Refactoring Helper. The review results are expressed as natural language scenarios describing the application. Each scenario includes associated Quality Attributes, along with ratings for difficulty and importance on a scale from A to C, where A represents the most challenging or the most critical.

5.1 Selection of Review Method

To prepare for the Architecture Review, we examined the analysis methods SAAM [KBAW94], ATAM [KKB+98], and ASAAM [Tek04]. The latter two were reviewed because they are referenced by the Architecture Refactoring Helper (ARH), while SAAM was included as it served as the foundation for the development of the other two methods. Additionally, we studied the theses by Knodel [Kno23] and Herrmann [Her24], as these prior case studies utilized the ARH and conducted architecture reviews within the scope of a thesis. Knodel notes that performing a full-scale SAAM-based architecture review is impractical in the context of a thesis [Kno23], as it requires several weeks, which is incompatible with both the limited timeframe and the company's need to avoid engaging key stakeholders for such an extended period. As a result, we adopted a modified version of Svahnberg and Martensson's architecture evaluation process [SM07], as shown in Table 5.1.

Step	**Planned Duration**
Introduction of the methodology	5 min
Introduction of the application to migrate	5 min
Identification of focus QAs	10 min
Creation of scenarios	45 min
Break	15 min
Rating importance and difficulty	15 min
Identification of secondary QAs for every scenario	15 min
Assessment of architecture type	20 min
Debriefing	20 min

Table 5.1: Architecture review agenda

5.2 Review Setting

The Architecture Review was organized with six participants: our head of development and support(HoDaS), the system architect, two senior developers, two junior developers, and our evaluation team. The demographic data for the participants, including their role, the years of experience with software engineering and microservices, is provided in Table 5.2. Unfortunately, the HoDaS could not attend due to unforeseen circumstances, and one senior developer participated remotely. A 150-minute timeslot was reserved in advance, as per the agenda.

	Experience in years	
Role description	**Software Engineering**	**Microservices**
Head of Development and Support	19	14
Software Architect	16	14
Senior Developer	33	21
Senior Developer	16	8
Junior Developer	7	2
Working Student	2,5	1

Table 5.2: Demographic data of the participants of this architecture review.

Each developer received a physical copy of the quality attributes list. The distributed version was a German translation of what can be found in Table 1, as the review was conducted in German. The handed-out version was made available as “qualityattributes.pdf” under [Zen23] These quality attributes, based on the work of Koch [Koc22], extend the ISO25010 standard [Int11].

To ensure transparency, the evaluation team shared their screen throughout the meeting so all participants could follow the notes in real-time and intervene if needed to prevent miscommunication or misrepresentation. The session was audio-recorded, and the shared screen was captured to mitigate the risk of missing critical details, as the evaluation team comprised only one person instead of the recommended three to four members [SM07]. All participants were informed about the recording at the beginning of the meeting, and no objections were raised regarding its use for this thesis.

The session began with an introduction to the methodology of the architecture review, a clear explanation of its objectives, and a reassurance that the purpose was to gather insights into the quality requirements of the application, not to evaluate the project’s overall quality or the performance of the developers. This clarification aimed to prevent defensive behavior that could inhibit participant contributions and compromise the review’s outcomes. Following this, one of the developers provided a brief overview of the application’s architecture to establish a shared understanding for subsequent discussions.

5.3 Review Steps

During the review, the participants produce intermediate results through the selection of primary quality attributes, formulating natural language scenarios, rating the importance and difficulty, associating additional quality attributes implied by the scenarios, and lastly evaluating what architectural style benefits the scenarios the most. This section describes how the participants came to their results while incrementally building the utility tree, showcasing the results.

5.3.1 Quality Attribute Selection

The first task required participants to select three lower-level quality attributes (Sub-QAs) they considered most critical for the application. Each participant made their selections individually from the QA Listing, shown in Table 1 [Zen23]. To minimize mutual influence, participants were instructed not to communicate with each other, except for clarifications about the meaning of specific quality attributes. The individual selections are shown in Table 5.3, while the accumulated selections are summarized in Figure 5.1.

Participant	**1**	**2**	**3**	**4**	**5**
Quality Attribute	Availability Time Behavior Modifiability	Fault Tolerance Recoverability Analysability	Availability Fault Tolerance Recoverability	Availability Time Behavior Modularity	Availability Fault Tolerance Modifiability

Table 5.3: Participants and Their Quality Attributes

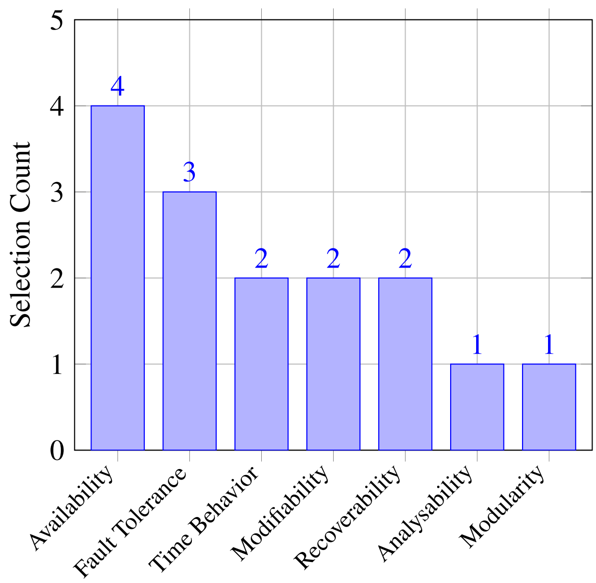

Figure 5.1: Frequency of attribute selections across all participants.

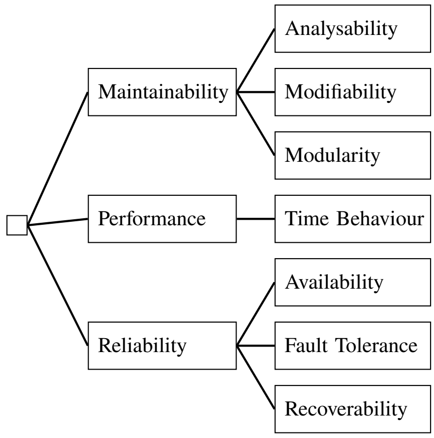

Figure 5.2: Utility Tree after focus QA selection

5.3.2 Scenario Creation

Following a brief discussion, participants agreed to create two scenarios for each quality attribute that received more than one selection. As a result, the Analyzability and Modularity attributes were excluded from the initial scenario creation. Proposals for scenarios were suggested by one participant and refined collaboratively until no objections remained. The resulting intermediate utility tree is shown in Figure 5.3.

After crafting two scenarios for each of the five prioritized quality attributes, the group decided to include the Maintainability sub-attribute, Analyzability. They added two scenarios to the existing ones, resulting in the updated utility tree shown in Figure 5.4.

5.3.3 Importance and Difficulty Rating

Each scenario was rated on a scale from “A” to “C”, where “A” represented the highest importance and difficulty, and “C” represented the lowest importance and least difficulty. This uniform rating scale aligns with the first phase of the Architecture Refactoring Helper. Although participants initially had diverse opinions, brief discussions consistently led to consensus, producing a universally accepted importance and difficulty rating. The ratings, presented in the format “Importance/Difficulty”, are visualized in Figure 5.5.

5.3.4 Associated Quality Attributes

The final step in scenario creation involved identifying additional associated quality attributes for each scenario. Participants discussed which other attributes were implied by the scenarios. These secondary attributes are included in the utility tree at a lower level, as seen in Figure 5.6. In the utility tree, secondary attributes are visually distinguished from the primary focus attributes to reflect the order of artifact creation. However, this distinction holds no significance for the Architecture Refactoring Helper, as all attributes associated with a scenario are treated equally.

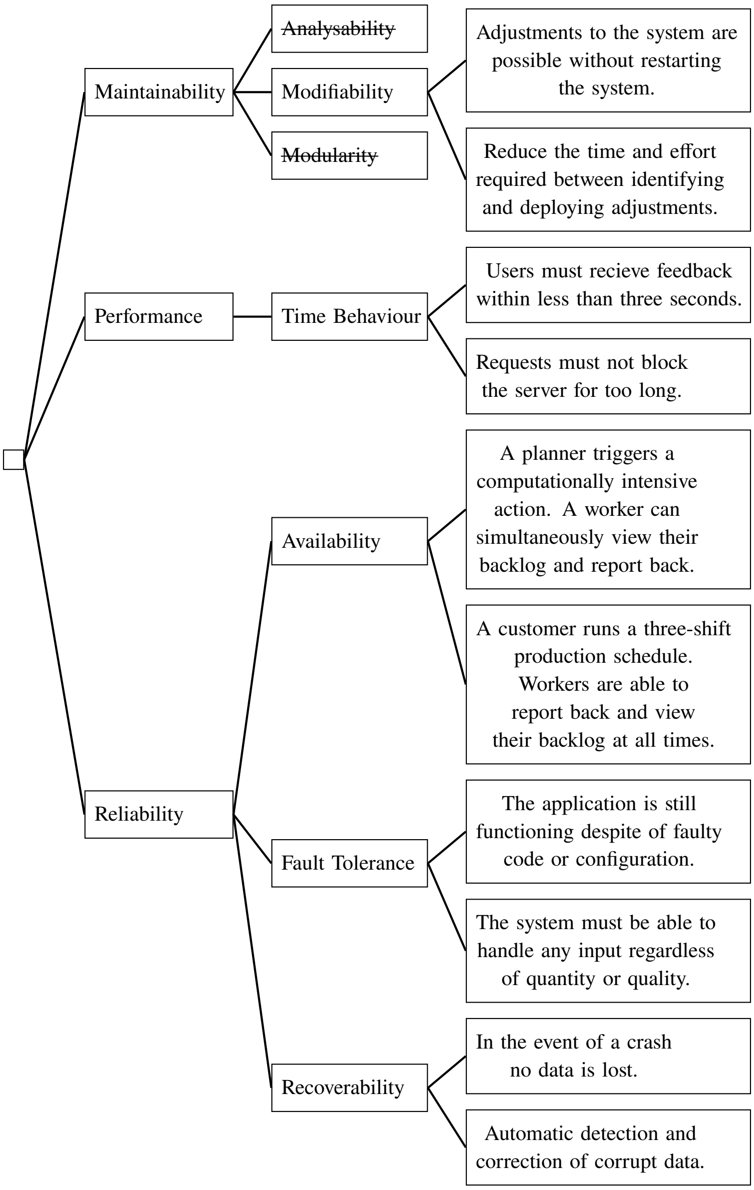

Figure 5.3: Utility Tree after initial scenario creation

5.3.5 Scenario-Architecture Evaluation

The final step of the architecture review involved evaluating each scenario against architectural styles. Participants discussed whether a monolithic or microservice-based architecture would better support each scenario. If a scenario was deemed architecture-independent or if the advantages of the two styles were considered equal, it was categorized as neutral. In Figure 5.7, blue scenarios favor a microservice-based architecture, red scenarios favor a monolithic architecture, and neutral scenarios are shown in gray.

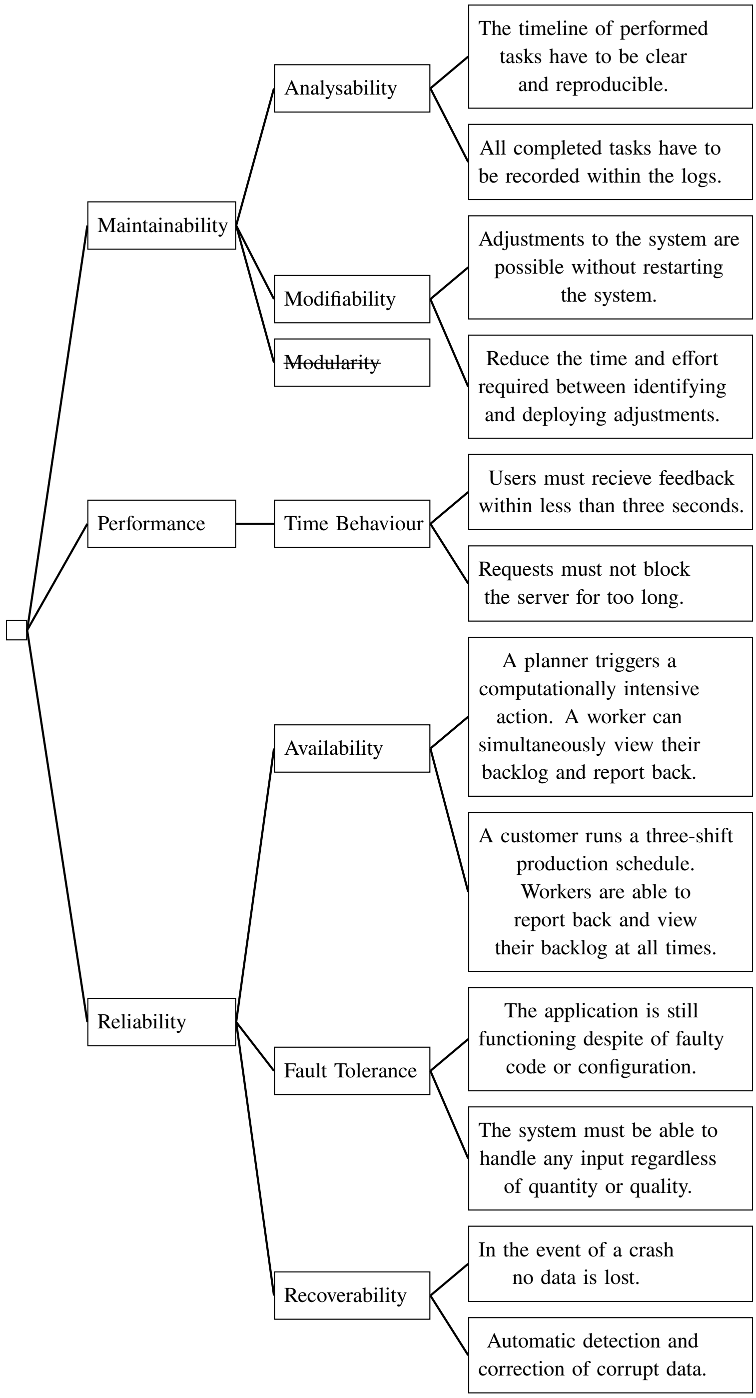

Figure 5.4: Utility Tree after adding the Analysability attribute

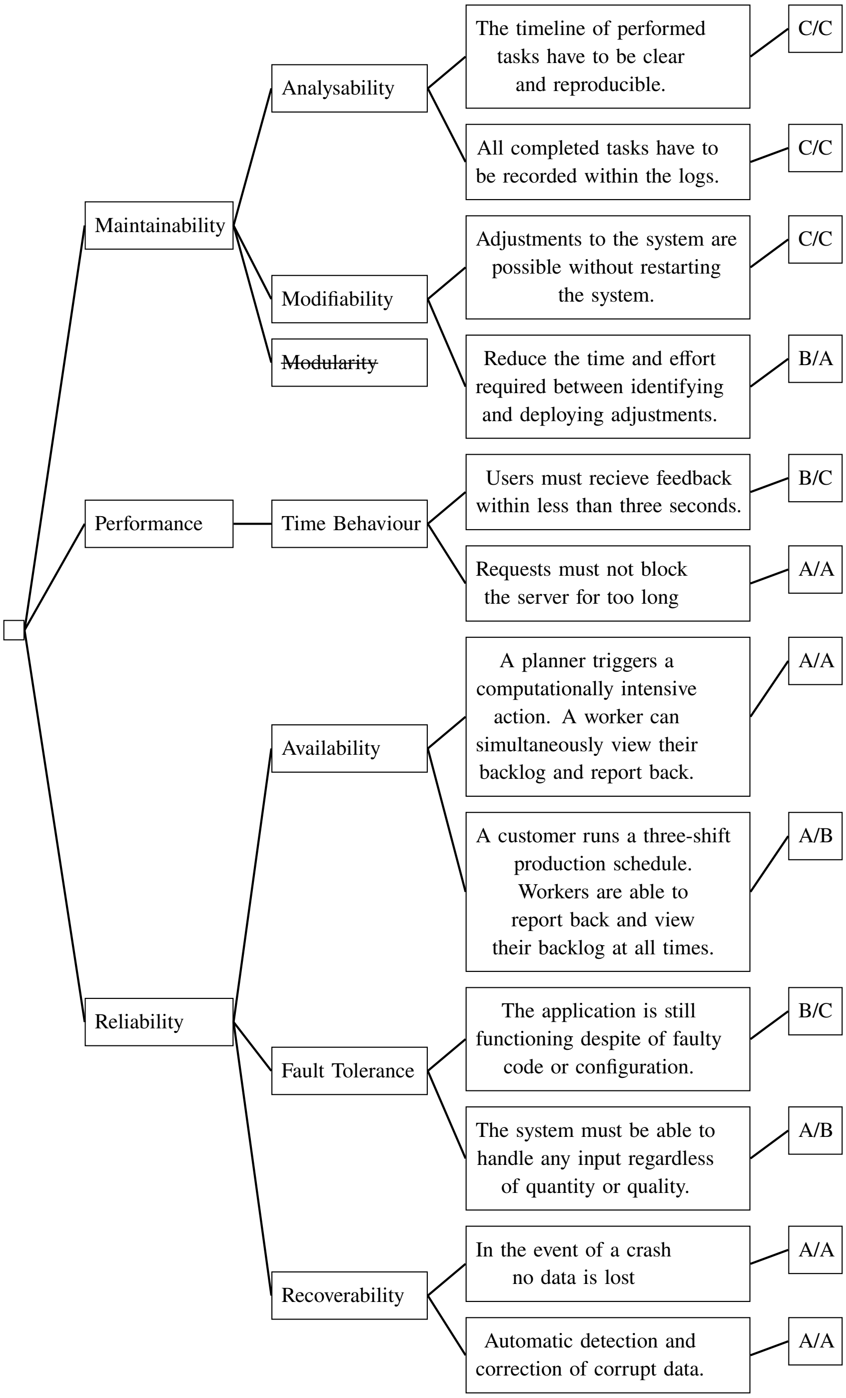

Figure 5.5: Utility Tree after rating the importance and difficulty.

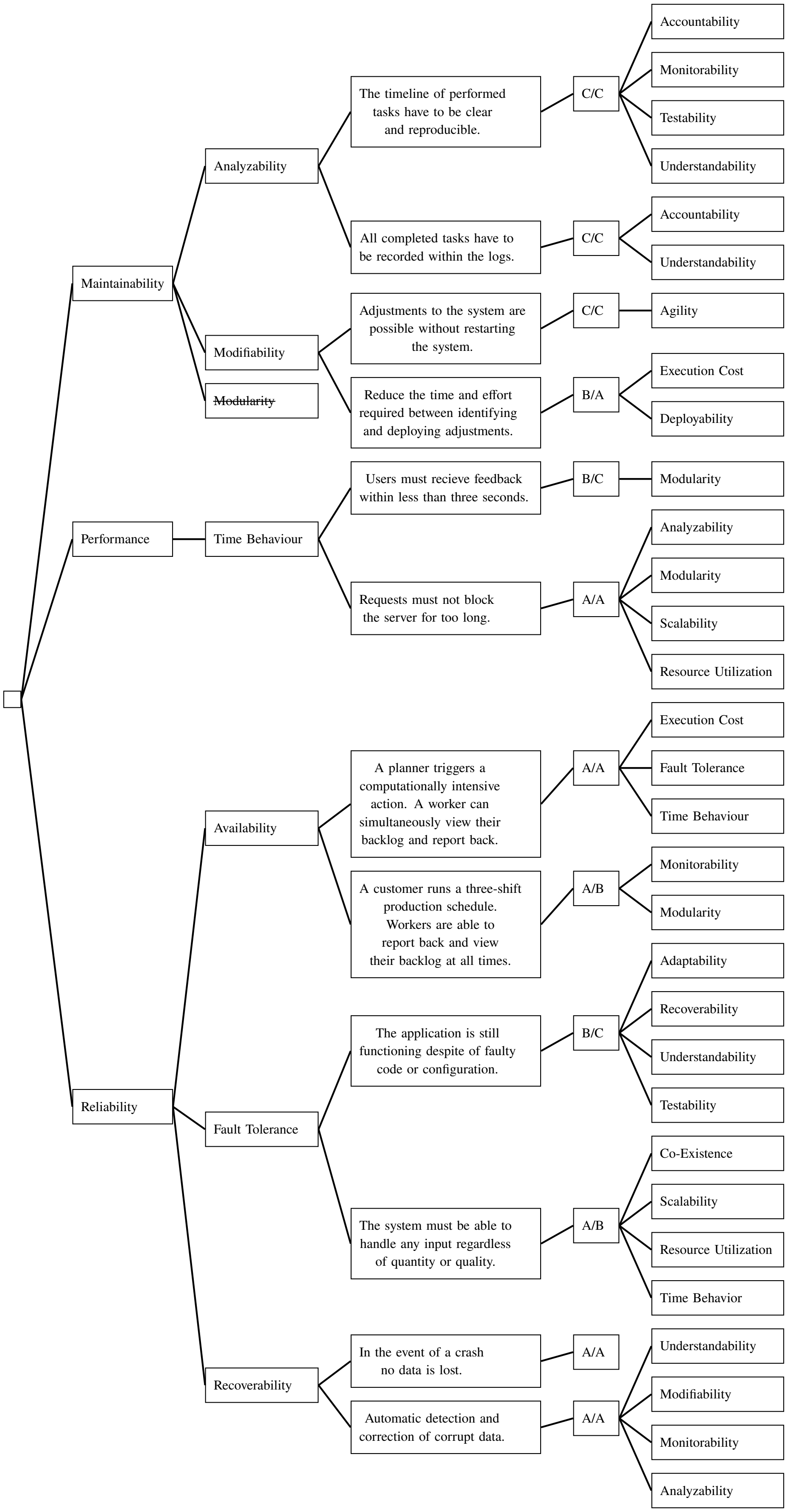

Figure 5.6: Utility Tree after associating additional quality attributes

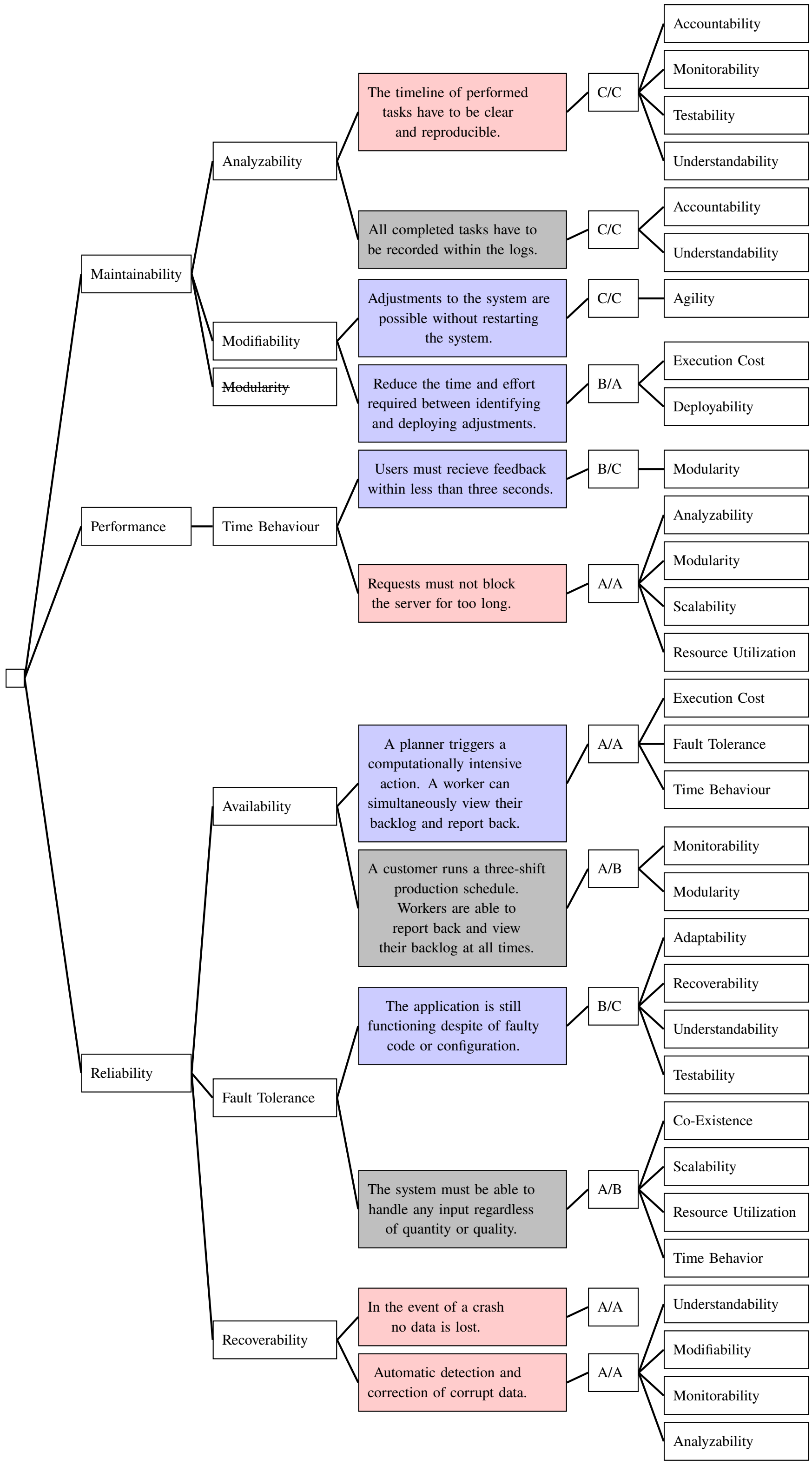

Figure 5.7: Utility Tree with colored scenarios according to the favored architectural pattern. Scenarios are marked red when favoring the monolithic architectural style, blue for microservices, and grey if they were evaluated to benefit equally from both.

5.4 Discussion of Results

In the preceding section, we presented the results and the setting in which they were attained. Here we discuss the individual results, in chronological order, matching the previous section. Lastly, we share an anecdotal story that happened during the break, showing the difficulty of preventing biases from influencing results.

5.4.1 Selection of Quality Attributes

Quality attributes from the Reliability category were chosen nine times, accounting for 60% of all selected sub-QAs. Each participant anonymously identified their top three priority quality attributes, after which we discussed the results and the rationale behind their selections. The primary reason for prioritizing the Reliability group, Availability, Fault Tolerance, and Recoverability, was the belief that if a system does not function reliably, all other aspects become irrelevant.

A few hours following the interview, I spoke with some of the participants, and one suggested that the emphasis on reliability might have been influenced by the order in which the quality attributes were presented. Even though all participants reviewed the entire document, they might have focused on the first page the most, as it was the first information they were presented with, where the three reliability attributes were prominently featured, resulting in a heightened emphasis on them.

5.4.2 Scenario Creation

When tasked with creating scenarios for their selected quality attributes, participants initially struggled to define what a scenario should entail. To address this, they agreed upon a definition: a scenario represents a situation within the application's context, described concisely in natural language in no more than three sentences. The process began with selecting one of the prioritized quality attributes, followed by a brainstorming session in which participants proposed scenarios. Initially, engagement was equal among all participants, with scenarios being discussed, refined, and revised. However, as the discussion progressed, an increasing number of participants became disengaged, leading to the scenario development process being dominated by the component's main developer and a working student with only superficial knowledge of the component. The other participants sporadically contributed to the discussion. This problem did not carry over to later review steps, as the participants regained focus during the break.

5.4.3 Importance and Difficulty

The rating for the individual scenarios reflects how the quality attributes have been valued. Earlier, we stated the importance of the Reliability attributes for the participants. Five out of the six reliability scenarios were given the highest importance. The one scenario that is an exception to this is the Fault Tolerance one, demanding the functionality of the application despite faulty code or configuration. To explain this irregularity, it is important to note that the ratings were given based on the degree of them being already fulfilled. Since multiple safeguards were already implemented to ensure correct inputs, or the internal correction of faulty inputs, as well as the validation of functionality, this scenario was categorized as low difficulty.

5.4.4 Associated Quality Attributes

The identification of additional quality attributes related to the scenarios took place through an open discussion, where participants suggested a QA, which was then either accepted or dismissed. During this discussion, some participants periodically went through their copies of the quality attributes, while others solely relied on their memory. While the participants did associate many different attributes to the scenarios, doing so from memory might have resulted in a less diverse discussion about quality attributes, which were not as internalized as others.

5.4.5 Scenario-Architecture Evaluation

The architectural style that best implements the scenario was again openly discussed. The results are based on the personal opinions of the participants. We present the summarized and paraphrased argumentation presented by the participants during the evaluation, which led to the results for each scenario.

The timeline of performed tasks has to be clear and reproducible: The distribution of tasks across services could introduce uncertainties when executing tasks concurrently.

All completed tasks have to be recorded within the logs: The level of detail in logs is independent of the number of services.

Adjustments to the system are possible without restarting the system: When having multiple services, the affected ones can be turned off while the remaining stay running, while with a monolith, the application is deployed as a whole and can therefore not be replaced without restarting the system.

Reduce the time and effort required between identifying and deploying adjustments: Because of the separation into services, the location to adjust can be identified and deployed faster.

Users must receive feedback within less than three seconds: Horizontal scaling, which is easier to achieve with microservices, allows the application to perform tasks more efficiently without the need to scale the entirety of the system.

Requests must not block the server for too long: The protocols used to communicate between services introduce more latency than standard program calls.

A planner triggers a computationally intensive action. A worker can simultaneously view their backlog and report back: The action triggered by the planner could be executed by a different service, which executes the task concurrently to the worker's requests, executed in another service.

A customer runs a three-shift production schedule. Workers can report back and view their backlog at all times: The uptime of an application is predominantly determined by the robustness of the system, and periodically executed backups are independent of the service count.

The application is still functioning despite faulty code or configuration: If a certain part of the system is not working properly due to faulty code or configuration, having independent services would decrease the number of affected components.

The system must be able to handle any input regardless of quantity or quality: A monolithic implementation can handle the same amount of input as a horizontally scaled microservice application if it allocates the same amount of resources for concurrent execution.

In the event of a crash, no data is lost: Having the data managed by one entity reduces the risk of data loss during communication between services.

Automatic detection and correction of corrupt data: When the data is passed between services, validation would have to occur at each service, hence requiring more effort.

Contradicting Arguments

As we previously mentioned, during the architecture review, we did not interfere with the discussions held by proposing our own opinion. Yet we wanted to elaborate on some arguments made, as we see arguments contradicting each other:

Having the response time reduced by a microservice-based implementation would also reduce the time the application is blocked by a request. Additionally, the component that accepts requests could forward the execution of the task to a microservice, further reducing the time it is blocked.

Another scenario we want to discuss further is the availability within a three-shift production schedule. Having microservices, the periodic restarting and backup procedures could be improved by starting a second instance of a certain service in the same state, leading to less or no downtime.

Having a system process input of any quantity or quality could benefit from microservices, when accounting for horizontal scaling, which was acknowledged by the participants in the response time scenario.

5.4.6 Conversations During Break

While we did not actively work on the architecture review, the 15-minute break played a crucial role in maintaining the review's quality, allowing participants to regain focus and energy. During this time, the main developer of the 3LiterPPS-Server® asked what additional results they need to produce, so that this review is in alignment with our goals. When we asked why he suspected an attempt to steer the results, he shared his concern that since the decision to migrate to a microservice-based architecture had already been made, our study might merely justify it retrospectively. We clarified that the idea of microservices had been thoroughly discussed before the start of this thesis with the system architect, lead developers, and company management. The primary objective of this thesis is to assess the feasibility of a microservice architecture for our specific use case and to evaluate the applicability of the Architecture Refactoring Helper in an industrial setting. Additionally, we emphasized that this study does not seek to predetermine any outcomes and that if a monolithic architecture proves to be the superior choice for our needs, we would substantiate that conclusion, instead of trying to force certain results. Furthermore, we reassured the developer that even if a transition to microservices were to occur, it would still be based on the legacy code, with necessary adjustments, rather than discarding prior work. The existing codebase would continue to serve as the foundation for core application functionalities. This explanation visibly reassured the developer, who had initially perceived the transition as an implicit dismissal of past development efforts.

6 Service Identification Approach

Through a structured, multi-step process, we narrow down over a hundred academic Service Identification Approaches to a handful that align with our organization's needs and constraints. Beginning with stakeholder-driven preferences covering quality, inputs, process, output, and usability dimensions, we apply successive filters to surface the most promising candidates. Finally, we check input and process requirements, assess edge cases in detail, and select the three approaches that best fit our context.

6.1 Preferences

In addition to quality attributes, which we already defined through the Architecture Review, the filter preferences include system or design properties, different types of inputs, process preferences regarding the techniques used and process strategy, the format of the output, and usability preferences like the degree of automation and different types of tools to assist in identifying services. The initial preference selection was performed in cooperation with the Head of Development and Support. All preferences that have been either included or excluded are listed in Table 6.1. The design properties, as a whole, are desirable properties for the candidate architectures, but to avoid including all options, we limited them to the three design properties, which are the subjectively most desirable by the HoDaS. The most sensitive part of the preference selection was the segment of the inputs, as the collaborating company is cautious not to expose critical information. Therefore, all forms of source code had been excluded. Additionally, the database file was excluded, as the application does not use a typical database for data persistence, making SIAs relying on database schemas inapplicable. The remaining input options, which can be provided without extensive effort, were included. Within the process preferences, the process strategies were selected based on whether or not it was seen as feasible to perform a migration with those strategies. Greenfield development and rewrite/rebuild require the application to be completely rewritten, increasing the potential for unwanted behavior from newly written source code. Contrary strategies like the strangler pattern and continuous evolution focus on the decomposition of existing components, which is easier to perform in parallel to regular production.

Parts of the potential process preferences and all of the Output and Usability preferences have been omitted at this stage, as the HoDas wanted us to be able to make further adjustments on our own, as the remaining options are more relevant to the executors of the service identification approach. Therefore, we decided on the following list of preferences. For the level of automation, we included automatic and semi-automatic, as being supported by external tools would simplify the decomposition of an application of this size. We did not want to exclude the manual approaches, as those SIAs remain a valid option. We included all analysis types except dynamic analysis, as the realization requires publishing the application's executables to have a reproducible case study. Like with the design properties, we decided to limit the number of included techniques to a maximum of

Preference Name	Preference Group	State of inclusion
Autonomy	Design Property (Quality)	✓
Granularity		✓
Isolation		✓
Human Expertise	Domain artifacts (Input)	✓
Ontology		✓
Version Control System		✓
Log traces	Runtime artifacts (Input)	✓
Business process model	Model artifacts (Input)	✓
Class diagram		✓
Use case model		✓
API / Interface (No specification)	Executables (Input)	✓
Database file (No specification)		×
Source Code (Java)		×
Source Code (No specification)		×
Source Code (Python)		×
Continuous Evolution	Process Strategy (Process)	✓
Greenfield		×
Refactor		✓
Rewrite / Rebuild		×
Strangler		✓
Class	Atomar Unit (Process)	✓
Function		✓
Interface		✓

Table 6.1: Included and excluded preferences, selected by the Head of Development and Support.

five. We did not favor any of the output formats but excluded Source Code for the same reason as for the inputs. Moreover, we are skeptical of an SIA, if it exists, that generates fully functional Source Code without the need for the same as input. Besides the tool types database and dynamic analysis, which we did not include for the same reasons as above, reverse engineering was not included, because we did not see the need to reverse engineer the company's project, for which we have human expertise, and "Other" for being too unspecific. This leads to the complete preference selection shown in Table 6.2

6.2 Different Filtering Queries

To better reflect the organization's needs, we performed four queries. We started by comparing the effects of the weights attributed to scenarios, compared to unweighted, manually configured quality attributes (**Q1**). Afterward, we reevaluated the quality attributes to reduce the skewing of results, introduced by the aggregation of matches through QAs without associated SIAs (**Q2**). Subsequently, we performed a query with all preferences set, retrieving the SIAs with the highest overall preference overlap (**Q3**). Lastly, we excluded all quality attributes originating from the architecture review, as the first two queries ranked the SIAs purely by their matching QAs (**Q4**).

Preference Name	Preference Group	State of inclusion
Autonomy	Design Property (Quality)	✓
Granularity		✓
Isolation		✓
Human Expertise	Domain artifacts (Input)	✓
Ontology		✓
Version Control System		✓
Log traces	Runtime artifacts (Input)	✓
Business process model	Model artifacts (Input)	✓
Class diagram		✓
Use case model		✓
API / Interface (No specification)	Executables (Input)	✓
Database file (No specification)		×
Source Code (Java)		×
Source Code (No specification)		×
Source Code (Python)		×
Continuous Evolution	Process Strategy (Process)	✓
Greenfield		×
Refactor		✓
Rewrite / Rebuild		×
Strangler		✓
Class	Atomar Unit (Process)	✓
Function		✓
Interface		✓
Automatic	Levels of automation (Process)	✓
Semi-automatic		✓
Historic	Analysis types (Process)	✓
Lexical		✓
Static		✓
Clustering	Techniques (Process)	✓
Custom heuristics		✓
Domain-Driven Design		✓
General Guidelines		✓
Graph-based		✓
Source code	Representation (Output)	×
Decomposition	Tool types (Usability)	✓
Java		✓
Open Source		✓
Static Analysis		✓
Visualization		✓

Table 6.2: Complete list of included and excluded preferences.

6.2.1 Differences between Scenario-based and Manually Configured Recommendations

Besides the scenario-based recommendations for Service Identification Approaches, the user can manually configure the quality attributes. Using the Architecture Refactoring Helper, scenario-based recommendations seem more intuitive, as the scenarios document the reason for the selected quality attributes. Additionally, they require less effort to use than manual configuration. Despite that, there are considerable reasons not to use scenario-based recommendations. When a quality sub-attribute is selected, its parent attribute is automatically selected and considered for the SIA ranking. This inclusion can only be circumvented, if desired, by manually configuring the quality attributes. Another reason is the duplication of shown preferences. In our case, including the automatically included parent attributes, a total of 26 quality attributes have been selected. Utilizing the scenarios, 72 are listed, as each quality attribute and its parent attribute are listed in every scenario they occur. These duplications clutter the screen, making it harder to determine missing attributes. Both the "Matches" and "QA Matches" metrics are unaffected by this weighting; what does change with importance is the tendency metric, which incorporates the importance of scenarios and their associated scenarios. The difference between the two recommendation types lies in the importance rating for scenarios. If a scenario was rated with an importance of A, B, or C, it was weighted with the factors 3, 2, and 1, respectively [Fri24a]. As seen in Table 6.3, which presents the results of **Q1**, there is little difference between the rankings of manual configuration and scenario-based. In two instances, the order of two SIAs has swapped due to the weighting. Furthermore, the seventh-highest-ranked SIA by scenario-based recommendation would have had a lower visual tendency rating, because it passed the threshold of 30% As seen in Table 6.3, there are differences in ranking order when ordering by tendency. This originates in the higher weighting of more important quality attributes. Due to the little differences and the scenarios acting as the reasoning for quality attributes, we will continue using the scenario-based recommendations for further queries.

6.2.2 Quality Attribute Reevaluation

The ranking of the Service Identification Approaches is based on the number of matching preferences. There is a "Matches" metric, ordering the SIAs by highest overall preference matches, a "QA Matches" metric, ordering by quality attributes only, and a "SP Matches" metric, only taking system properties into account. Furthermore, a suitability score or tendency calculates the overall percentage of matches of quality and system properties for each SIA. The metric is visualized by an arrow, which is facing up if the percentage value is over 75%, inclined up between 50% and 75%, horizontal between 30% and 50%, inclined down between 15% and 30%, and down below 15%. If the percentage value is a threshold value, it is always assigned towards the higher rating, for example, 30% will result in a horizontal arrow[Fri24a]. The Architecture Refactoring Helper utilizes a repository of academic papers comprising 109 Service Identification Approaches, at the time of this thesis. We provide the repository in its applied state under [Zen23]. Including the sub-attributes, there are 36 different quality attributes to select. Therefore, we decided to check whether or not there are matching SIAs for each quality attribute associated with our scenarios, created in the architecture review. Because if there are attributes that no SIA represents, the metrics will show worse results, even though there are no entries in the repository that could fulfill the preferences to the fullest. This adaptation will reduce the skewing of the resulting suitability score. The results of the single attribute queries are shown in Table 6.4. As a consequence,

			Tendency rating			
			Visual		Percentage	
Ranking (Difference)	**Matches/Total Preferences**	**SIA ID**	**Scenario**	**Manual**	**Scenario**	**Manual**
1 (→)	15 / 26	2117	↗	↗	63%	58%
2 (→)	12 / 26	2018	→	→	49%	46%
3 (→)	12 / 26	2013	→	→	46%	46%
4 (→)	11 / 26	2201	→	→	45%	42%
5 (→)	10 / 26	1808	→	→	38%	38%
6 (→)	9 / 26	1905	→	→	36%	35%
7 (↓1)	7 / 26	2011	→	↘	31%	27%
8 (↑1)	9 / 26	2101	→	→	30%	35%
9 (→)	7 / 26	2109	↘	↘	28%	27%
10 (→)	6 / 26	2202	↘	↘	25%	23%
11 (→)	6 / 26	2102	↘	↘	25%	23%
12 (↓1)	5 / 26	1910	↘	↘	23%	19%
13 (↑1)	6 / 26	2205	↘	↘	21%	23%
14 (→)	5 / 26	1805	↘	↘	20%	19%
15 (→)	5 / 26	1904	↘	↘	20%	19%

Table 6.3: The 15 highest ranked Service Identification Approaches, ordered by the Tendency metric, for the manually configured quality attributes and scenario-based recommendations. The difference in ranking of the SIAs for the manual query is shown in parentheses.

the quality sub-attributes Adaptability, Agility, Recoverability, Analyzability(/Analysability), and Accountability have been removed from the scenarios. This resulted in the reduction of quality preferences from 25(/26) to 20. As seen in Table 6.4, no Service Identification Approach was categorized with the attributes Analyzability/Analysability, Adaptability, Agility, Recoverability, and Accountability. The SIA 2117, which is the highest rated on the first query, had Analyzability, Analysability, and Recoverability listed as matches, even though in the detail view, none of those are present. This is because if an upper-level quality attribute had been associated with an SIA, all its sub-attributes would also match. This skewed the metric in favor of the SIA.

The adjusted scenarios are shown in Figure 6.1. The scenarios for the Analyzability and Recoverability have been created, as those have been seen as important quality attributes, but only their additionally associated quality attributes have an impact on the Service Identification Approach ranking. Furthermore, one of the Recoverability scenarios didn't have other quality attributes, making it impactless and only serving a documentary purpose, despite being ranked as of highest importance. The scenarios now have, including duplications, 51 quality attributes, or 19 distinct quality attributes. The results after removing the zero-match quality attributes (**Q2**) are shown in Table 6.5. Through this change, SIAs that profited from the quality attributes without any direct matches have seen a downward correction, while those that were unaffected have seen an upward shift in their ranking, as the number of possible matches has decreased. Since the SIAs are not gaining matches from quality attributes that cannot match any SIAs directly, we will continue our queries with these adjusted scenarios and quality preferences.

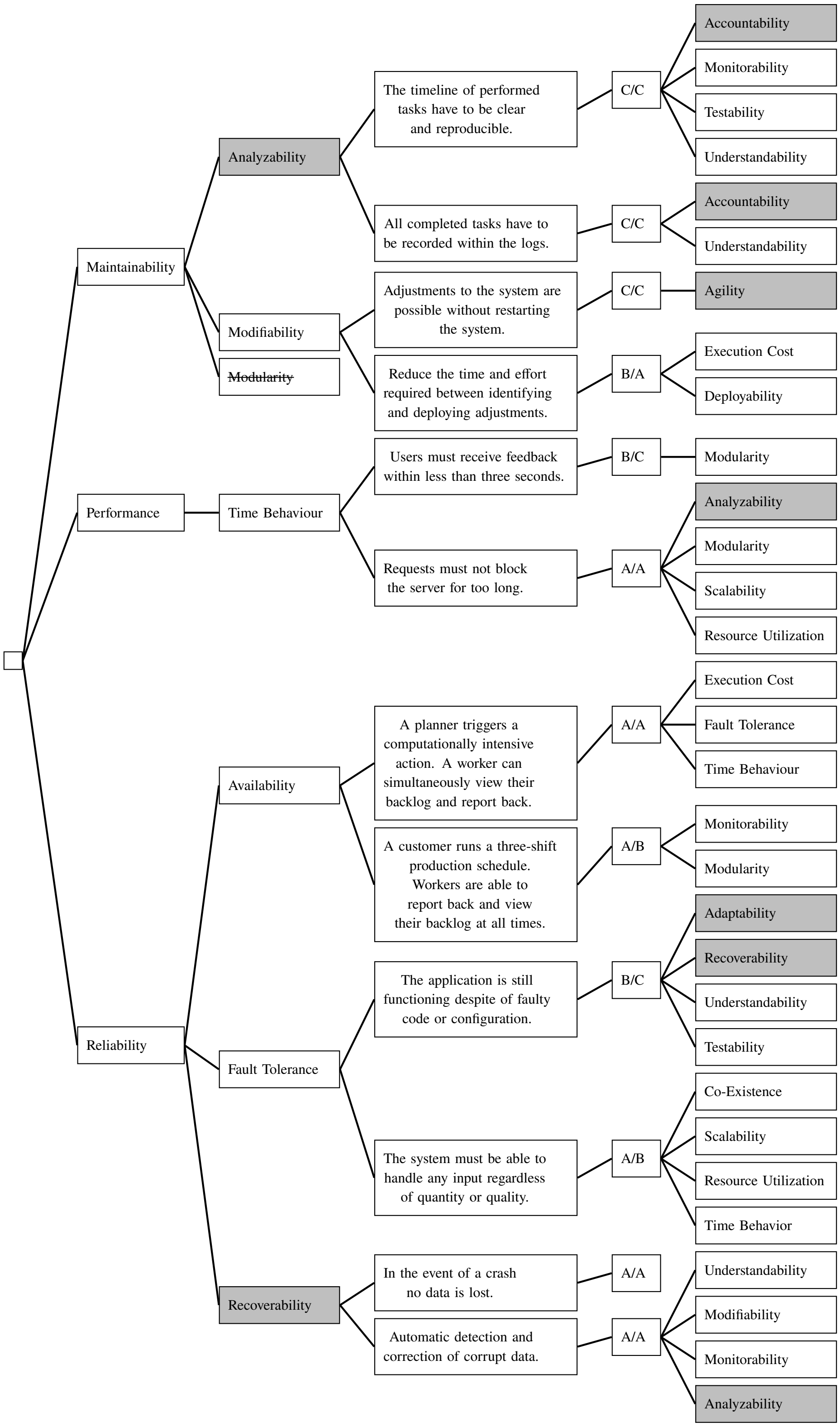

Figure 6.1: Utility Tree after removing quality attributes with no matching Service Identification Approaches. Quality attributes that have been removed are shown in gray.

Upper Quality Attribute	Lower Quality Attribute	Number of matching SIAs
Business		6
	Execution Cost	3
Compatibility		1
	Co-Existence	1
Maintainability		5
	Analysability/Analyzability	0
	Modifiability	2
	Modularity	16
	Monitorability	3
	Testabiltiy	1
	Understandabiltiy	3
Performance		25
	Resource Utilization	3
	Time Behaviour	7
Portability		6
	Adaptability	0
	Agility	0
	Deployability	11
Reliability		5
	Availability	7
	Fault Tolerance	2
	Recoverability	0
Scalability		20
Security		3
	Accountability	0

Table 6.4: Matching Service Identification Approaches for single attribute queries

6.2.3 Full Preference Configuration

In this section, we will perform a query that considers all preferences (**Q3**) since we have only looked at SIAs that best match our quality preferences so far (**Q1, Q2**). Theoretically, the results of this query should yield the best results, regarding the organization's preferences. The ranking in Table 6.6 is ordered by overall matches, not only quality attributes. We can see a large shift in Service Identification Approaches, for example, SIA 2012 ranked as second best while being in 48th place when filtering by quality attributes only, and SIA 2115 was ranked as 12th best, in comparison to 3rd last on 107th place. This discrepancy can be explained when looking at the quality attributes associated with the SIAs. In the case of SIA 2012, only three quality attributes have been identified: Coupling, Granularity, and Modularity(Maintainability). While others like the SIA 2018 hold multiple upper-level quality attributes: Maintainability, Performance, and Scalability. As mentioned in Section 6.2.2, those upper-level QAs will match all of their sub-attributes, even if not explicitly stated. As seen in Table 6.7, when ordering the recommendations by tendency, we keep multiple SIAs in the upper rankings, as they gain matches through upper attributes, while others that have been performing well according to the overall matches metric disappear.

Ranking (Difference to (Q1))	Matching/Total Preferences	SIA ID	Tendency rating Visual	Tendency rating Percentage(Difference)
1 (→)	13 / 20	2117	↗	68%(+6)
2 (→)	11 / 20	2018	↗	56%(+9)
3 (↑1)	10 / 20	2201	↗	51%(+8)
4 (↓1)	9 / 20	2013	→	46%(+1)
5 (↑1)	8 / 20	1905	→	41%(+5)
6 (↓1)	8 / 20	1808	→	39%(+1)
7 (↑2)	7 / 20	2109	→	35%(+8)
8 (↓1)	6 / 20	2011	→	32%(+1)
9 (↑1)	6 / 20	2202	→	31%(+7)
10 (↑1)	5 / 20	2102	↘	26%(+6)
11 (↑4)	5 / 20	1805	↘	25%(+5)
12 (↑4)	5 / 20	1904	↘	25%(+5)
13 (↑4)	5 / 20	2308	↘	25%(+5)
14 (→)	5 / 20	2004	↘	24%(+4)
15 (↓3)	5 / 20	2302	↘	24%(-7)

Table 6.5: The 15 highest ranked Service Identification Approaches, ordered by the Tendency metric, after removing quality attributes, that have no matching Service Identification Approaches. The difference in the ranking of the unadjusted quality attribute ranking (**Q1**) is shown in parentheses.

Ranking (Difference to (Q2))	Matching/Total Preferences	SIA ID	Tendency rating Visual	Tendency rating Percentage
1 (↑1)	21 / 51	2018	↗	54%
2 (↑46)	17 / 51	2012	↓	6%
3 (↓2)	16 / 51	2117	↗	66%
4 (↑3)	16 / 51	2109	→	37%
5 (↑11)	15 / 51	2101	↘	24%
6 (↓1)	14 / 51	1905	→	38%
7 (↑25)	14 / 51	1811	↘	17%
8 (↓5)	14 / 51	2201	→	49%
9 (↑2)	14 / 51	1805	↘	27%
10 (↓6)	13 / 51	2013	→	42%
11 (↑39)	13 / 51	2108	↓	6%
12 (↑95)	13 / 51	2115	↓	0%
13 (↓4)	12 / 51	2202	→	33%
14 (↑16)	12 / 51	2305	↘	17%
15 (↓9)	11 / 51	1808	→	42%

Table 6.6: The 15 highest ranked Service Identification Approaches, ordered by the overall preference matches (**Q3**). The ranking in the adjusted quality attribute only query (**Q2**) is shown in parentheses.

			Tendency rating	
Ranking (Difference to (Q3))	Matching/Total Preferences	SIA ID	Visual	Percentage
1 (↑2)	16 / 51	2117	↗	66%
2 (↓1)	21 / 51	2018	↗	54%
3 (↑2)	14 / 51	2201	→	49%
4 (↑11)	11 / 51	1808	→	42%
5 (↑5)	13 / 51	2013	→	42%
6 (→)	14 / 51	1905	→	38%
7 (↓3)	16 / 51	2109	→	37%
8 (↑27)	9 / 51	2011	→	34%
9 (↑4)	12 / 51	2202	→	33%
10 (↑17)	10 / 51	2102	↘	27%
11 (↑22)	9 / 51	1904	↘	27%
12 (↓3)	14 / 51	1805	↘	27%
13 (↑19)	9 / 51	2004	↘	26%
14 (↑11)	10 / 51	2308	↘	26%
15 (↑26)	9 / 51	2302	↘	26%

Table 6.7: The 15 highest ranked Service Identification Approaches, ordered by the Tendency metric with all preferences. The difference in ranking, when ordering by overall matches (**Q3**), is shown in parentheses.

6.2.4 Querying Without Quality Attributes

During the analysis of the query with all preferences, we saw multiple scenarios when looking at Service Identification Approaches. Some matched many quality attributes while having almost no matches in any other categories, like inputs or processes. The other extreme was the SIAs that matched little to no quality attributes, but many of the remaining preferences. Furthermore, there are examples like the SIA 2303, having only four attributes associated with it: Coupling, Guideline/Workflow, No validation, and No tool support. There are no quality attributes, no inputs, no process, and almost no usability associated. With this set of categorized attributes, this SIA cannot rank high when out of all preferences, only at most four can match, and other Service Identification Approaches, in our query, accumulate triple the amount of matches by having three upper QAs associated with them. Therefore, we performed another query (**Q4**) without having any quality attributes defined. This query ranks the SIAs with the highest overlap of preferences affecting the migration process, not the quality attribute-related outcome. We chose to include the system properties, as those have been consistently assigned to Service Identification Approaches, unlike the quality attributes. In Table 6.8 we see that we again gained new SIAs that have not ranked high in previous queries. Out of the listed Service Identification Approaches, eight have not been represented in the queries where quality attributes were factored in. Across these 15 SIAs, most of them have been categorized without any quality attributes.

Ranking (Difference to (Q3))	Matching/Total Preferences	SIA ID
1 (↑1)	16 / 32	2012
2 (↑10)	13 / 32	2115
3 (↑8)	12 / 32	2108
4 (↓3)	11 / 32	2018
5 (↑12)	11 / 32	1804
6 (↑1)	11 / 32	1811
7 (↑23)	10 / 32	1701
8 (↓3)	10 / 32	2101
9 (↑18)	9 / 32	1907
10 (↑14)	9 / 32	2208
11 (↑26)	9 / 32	2301
12 (↑26)	9 / 32	1902
13 (↑5)	9 / 32	1602
14 (↓5)	9 / 32	1805
15 (↑4)	9 / 32	2113

Table 6.8: The 15 highest ranked Service Identification Approaches, ordered by the overall preference matches (**Q4**). The Quality attributes have been removed. The difference in ranking of these SIAs compared to when querying with quality attributes (**Q3**) is shown in parentheses. The visual and percentage tendency ratings have been omitted due to the lack of quality attributes to measure.

6.3 SIA Selection

As a collection to choose from, we took the top 15 SIAs from every query, for a total of 60 SIAs. After removing duplicates, it comes down to 31 distinct Service Identification Approaches, listed in Table 6.9.

1602	1701	1804	1805
1808	1811	1902	1904
1905	1907	1910	2004
2011	2012	2013	2018
2101	2102	2108	2109
2113	2115	2117	2201
2202	2205	2208	2301
2302	2305	2308	

Table 6.9: Unfiltered pool of Service Identification Approach, ordered by their Id.

6.3.1 Preliminary Filtering based on Associated Attributes

As mentioned earlier, some of the filters we apply are strict requirements rather than mere preferences. Regarding the inputs, these are Source Code and Database files, primarily due to confidentiality and availability concerns. Since some SIAs still rank highly despite possessing these excluded attributes, we will remove them from the selection pool if they have no other inputs beyond the restricted ones. However, if at least one acceptable input type is present, certain SIAs may still be feasible. The updated selection pool is shown in Table 6.10, with the Approaches grouped accordingly. Eight Approaches have been excluded due to their lack of input options, and 13 have to be looked at in detail to check if they are executable without the use of the unavailable inputs. The process

Included	Potentially	Excluded
1602, 1904	1701, 1804	1808, 1902
1905, 1907	1805, 1811	2108, 2109
1910, 2004	2012, 2013	2202, 2205
2011, 2117	2018, 2101	2208, 2301
2302, 2308	2102, 2113	
	2115, 2201	
	2305	

Table 6.10: Pool of Service Identification Approach after categorization by needed inputs, grouped by their inclusion state, ordered by their Id.

strategies “Greenfield” and “Rewrite / Rebuild” are handled the same way. If there are additional process strategies listed, we will review them to assess their applicability and exclude them if they are the only ones listed. The updated SIA inclusion states are shown in Table 6.11.

Included	Potentially	Excluded
2004	1602, 1701, 1804	1808, 1902
2308	1805, 1811, 1904	2108, 2109
	1905, 1907, 1910	2201, 2202
	2011, 2012, 2013	2205, 2208
	2018, 2101, 2102	2301
	2113, 2115, 2117	
	2302, 2305	

Table 6.11: Pool of Service Identification Approach after categorization by process strategy, grouped by their inclusion state, ordered by their Id.

6.3.2 Assessment of Potentially Viable Service Identification Approaches

Here, we will assess the applicability of the SIAs categorized as “Potentially” in Table 6.11. We will review the papers individually until we can either exclude them or reason why they are still applicable, despite having undesirable properties.

1602 - Service Cutter

While reviewing Service Cutter [GKGZ16a], we explicitly found domain models and use cases mentioned as usable inputs. Furthermore, Gysel et al. explicitly mention green field development and monolith to microservice as a usage scenario for their SIA [GKGZ16b]. The wider variety of inputs and the usage for a decomposition without green field development resulted in bringing the Service Cutter Approach back into the pool of included SIAs.

1701 - Microservices Identification Through Interface Analysis

As the title of the SIA suggests, the approach by Baresi et al. [BGD17] requires the execution of an analysis of interfaces. More precisely, they require the OpenAPI specification for the interfaces, which is designed for HTTP APIs [Ope24]. Our application is not using HTTP APIs and therefore this approach does not fit our specific use case and is excluded.

1804 - Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems

The “Discovery Process” of the SIA needs the application’s source code and SQL queries for its Business Object Analyser [DBPF18]. As we mentioned earlier, the application does not utilize a SQL database for data persistence, and due to confidentiality reasons, the source code cannot be used. Hence, De Alwis et al.’s Service Identification Approach is excluded.

1805 - Discovering Microservices in Enterprise Systems Using a Business Object Containment Heuristic

This SIA, also published by De Alwis et al. [DBFP18] shares the same problems as their other SIA and is therefore also excluded.

1811 - Visualization Tool for Designing Microservices with the Monolith-First Approach

The Visualization Tool for Designing Microservices by Nakazawa et al. [NUEH18] cannot be used without the need for the application’s source code, which directly excludes it from our selection pool.

1904 - Unsupervised learning approach for web application auto-decomposition into microservices

Abdullah et al. [AIE19] proposed a decomposition procedure for web applications using the application’s source code as input. Unfortunately, the application is not web-based. Therefore, our application is not part of this SIA’s target group, leading to its exclusion.

1905 - From Monolithic Architecture to Microservices Architecture

The Approach to identify microservices by De Lauretis was published in a state, where the author himself considers it not functionally complete [De 19]. We therefore will not consider this SIA further.

1907 - A dataflow-driven approach to identifying microservices from monolithic applications

The reason we had to further review Lia et al.'s dataflow-driven Service Identification Approach [LZJ+19] was that the Approach was categorized as needing to rewrite the source code of the microservice candidates and not reuse the legacy code. Reviewing the literature, we did not find arguments for that to be necessary; hence, no arguments to exclude the SIA.

1910 - Efficient hybrid research for QoS-aware microservice composition

The paper by Singhal et al. [SSR19] does not focus on decomposing monolithic applications into microservices. Instead, it addresses how to efficiently compose and coordinate microservices using a hybrid approach that combines orchestration and choreography. For this reason, we exclude this paper and propose its exclusion from the Architecture Refactoring Helpers repository.

2011 - Optimization of Microservice Composition Based on Artificial Immune Algorithm Considering Fuzziness and User Preference

Gao et al. focused on the optimization of service composition for already existing microservice-based applications rather than decomposing monolithic applications[GCL+20]. Despite being focused on cloud-based applications, this approach might be utilized once the decomposition of the 3Liter-PPS-Server® has been performed, albeit in an adjusted form to cater to the non-cloud nature of our use case. Consequently, we will exclude said SIA for the time being.

2012 - Steinmetz: Toward automatic decomposition of monolithic software into microservices

Due to the intermediate artifacts created during the Steinmetz [LO20] service identification, we do not need to share the actual source code to make this reproducible. This led to the inclusion of this paper.

2013 - Towards Identifying Microservice Candidates from Business Rules Implemented in Stored Procedures

In their paper, Barbosa and Maia use stored procedures, which are database artifacts related to SQL [GM20]. As we previously mentioned, this does not overlap with our application. For that reason, we omit their SIA.

2018 - Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis

Matias et al. proposed a [MCF+20] their SIA with source code as a required input. Like Steinmetz [LO20], we might have been able to use their approach as there are intermediate artifacts produced, which are sufficient to reproduce this work, without having to share source code, if it weren't for the fact that this approach requires Python code.

2101 - A multi-model based microservices identification approach

In contrast to the information given by the ARH, we did not find reasons for the need for inputs like source code. Instead, Daoud et al. [DEF+21] use higher-level artifacts like BPMN representations for their approach. Therefore, we will include this approach back into our selection

2102 - A Hot Decomposition Procedure: Operational Monolith System to Microservices

In the Service Identification Approach by Ivanov and Tasheva, database decomposition is one of the core concepts [IT21]. Due to availability reasons, we have to exclude this approach.

2113 - Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices

We cannot execute the Service Identification with Mono2Micro [KXK+21] without having to share using source code. Therefore, this SIA will be excluded

2115 - Microservice decomposition and evaluation using dependency graph and silhouette coefficient

We have to exclude dos Santos and de Paula's decomposition approach [SP21], due to its usage of source code, jars, and the commit history of the application to compute a similarity for services.

2117 - Microservices backlog – a genetic programming technique for identification and evaluation of microservices from user stories

The classification of Vera-Rivera et al.'s Microservice backlog [VPAG21] indicates that this SIA is designed for refactorings and rewrites/rebuilds. Even though it is possible to rewrite an application with this approach, we do not see the necessity to disregard the legacy code and therefore add it to our selection pool.

2302 - Actor-driven Decomposition of Microservices through Multi-level Scalability Assessment

In their paper, Camilli et al. [CCRZ23] give an overview of their methodology, during which they added a footnote stating that it is possible but undesirable to start with a monolith, especially if other quality attributes besides scalability are important [CCRZ23]. Our initial situation leaves us with a monolith and a collection of other QAs we desire. We will not pursue this Identification Approach further.

2305 - A methodology for refactoring ORM-based monolithic web applications into microservices

We exclude Freitas et al.'s SIA[FFC23], based on their direct need for source code without intermediate artifacts. Additionally, their proposed methodology was designed for ORM-based monolithic web applications. Our application is neither ORM-based nor a web application.

Included	Excluded
1602, 1907	1701, 1808, 1804, 1805, 1811, 1902, 1904
2004, 2012	1905, 1910, 2108, 2109, 2011, 2013, 2018
2117, 2308	2101, 2113, 2115, 2102, 2201, 2202, 2205,
	2208, 2301, 2302, 2305

Table 6.12: Pool of Service Identification Approach after classifying Potentially-marked SIAs, grouped by their inclusion state, ordered by their Id.

6.3.3 Choosing Service Identification Approaches

After selecting and filtering the Service Identification Approaches from multiple queries, we end up with six SIAs, shown in Table 6.12, that we consider applying to the 3LiterPPS-Server®. To determine the three SIAs we move forward with, we compared them based on their classification within the Architecture Refactoring Helper [Fri24a]. We chose this method of evaluation, as it reduces skewing introduced through the Matches metric, which we described earlier in Section 6.2.2. The overview of the rated SIAs is shown in Table 6.13. Based on this metric, we select the three Service Identification Approaches with the highest scores:

1. 2012 - Steinmetz [LO20]: 13 Matches
2. 2117 - Microservices backlog [VPAG21]: 10 Matches
3. 1602 - Service Cutter [GKGZ16a]: 8 Matches

	1602	1907	2004	2012	2117	2308
Availability						
Business						✓
Co-Existence						
Compatibility	✓					
Deployability			✓			
Execution Cost						
Fault Tolerance					✓	
Maintainability					✓	
Modifiability						
Modularity				✓	✓	
Monitorability					✓	
Performance			✓		✓	✓
Portability						
Reliability					✓	
Resource Utilization						
Scalability			✓			
Testability						
Time Behaviour						
Understandability						
Autonomy						
Granularity	✓		✓	✓	✓	
Isolation						
Human Expertise						
Ontology						
Version Control System				✓		
Log Traces			✓	✓		
Business Process Model						✓
Class Diagram						
Use Case Model	✓	✓			✓	
API/Interface						
Continuous Evolution						
Refactor	✓	✓	✓	✓	✓	✓
Strangler						
Class				✓		
Function				✓		✓
Interface						
Automatic	✓			✓		
Semi-Automatic		✓	✓			
Historic				✓		
Lexical				✓		
Static	✓	✓		✓	✓	✓
Clustering	✓	✓		✓		✓
Custom Heuristics	✓					
Domain-Driven Design						
General Guidelines						
Graph-based				✓		
Decomposition						
Java						
Open Source						
Static Analysis						
Visualization						
Sum	8	5	7	13	10	7

Table 6.13: Listing of selected filters that are explicitly assigned to the Service Identification Approaches.

7 Service Identification

This chapter reports on the proof-of-concept execution of the, in Section 6.3 selected, Service Identification Approaches and compares the candidate microservices they produced. In Section 7.1, we introduce the Steinmetz SIA [LO20] and describe how we attempted to locally set up an instance of the corresponding tool [Löh25]. Section 7.2 covers the Microservices Backlog SIA [VPAG21], and our efforts to locate and execute the tool mentioned by it. The explanation and execution of the Service Cutter approach [GKGZ16a] in Section 7.3. We conclude this chapter in Section 7.4 with a discussion that contrasts the different service cuts, highlights key observations, and draws lessons for the future microservices architecture of the 3LITERPPS-SERVER® Dispo subcomponent.

7.1 Steinmetz

Löhnerts and Oprescu [LO20] identify service candidates by modeling a monolith's internal dependencies along three complementary dimensions, which are unified into a single coupling graph for clustering. The beginning of this section outlines their methodology, while the second part recounts our hands-on attempt to install and run the Steinmetz tool [Löh25], the outdated dependencies we encountered, our partial workarounds, and why we ultimately had to abandon its execution.

7.1.1 Service Identification Methodology

In [LO20], the authors introduce a methodology that decomposes a monolithic application into candidate microservices by constructing and integrating three distinct coupling graphs. The central idea is to capture various dimensions of interdependency among software components to identify cohesive clusters.

The process begins with the derivation of the static coupling graph. This graph is obtained by analyzing the source code to extract a call graph where each node represents a class and an edge indicates a method invocation between two classes. The interaction strength is quantified using a modified Response for a Class (RFC) metric [BDW99]:

$$\mathrm{RFC}_0(c_1, c_2) = \frac{|\mathrm{IM}(c_1, c_2) \cup \mathrm{IM}(c_2, c_1)|}{|\mathrm{AM}(c_1, c_2) \cup \mathrm{AM}(c_2, c_1)|},$$

with $\mathrm{IM}(c_i, c_j)$ denoting the set of methods invoked from c_i to c_j and $\mathrm{AM}(c_i, c_j)$ representing the set of accessible methods. This graph provides a concrete representation of the system's structure.

In parallel, the methodology constructs a semantic coupling graph that captures conceptual relationships among the source files. This is achieved by processing the source code through natural language processing techniques: the code is tokenized, filtered to remove language-specific keywords and non-informative symbols, and then transformed into vectors. Similarity measures are computed between these vectors, yielding a graph where the edge weights reflect the semantic closeness of the components.

The evolutionary coupling graph, generated by mining version control system logs, provides a third perspective. This graph captures the historical co-evolution of classes that are frequently modified together, receiving higher coupling weights. The underlying assumption is that such classes will likely belong to the same functional module.

These three graphs are then integrated into a single, comprehensive coupling graph G_{combined}. This is done by intersecting the static graph with the semantic and evolutionary graphs:

$$G_{\text{combined}} = (G_{\text{static}} \cap G_{\text{semantic}}) \cap G_{\text{evolutionary}},$$

Ensuring that only those relationships confirmed by the static analysis are considered, while the additional dimensions refine the coupling strengths. The combined coupling graph encapsulates a holistic view of the interdependencies within the monolith by unifying the structural, conceptual, and historical perspectives. Graph clustering algorithms are then applied to G_{combined} to partition the graph into clusters. Each cluster is interpreted as a potential microservice, grouping classes with high internal cohesion and minimal external coupling.

7.1.2 Execution

To identify microservice candidates with the Steinmetz approach [LO20], we cloned the Steinmetz git repository [Löh25] and tried to set up the tool according to the documentation. Unfortunately, at the time of this thesis, the dependencies have not been updated since August 2020, and some accessed repositories and certain dependencies and dependency versions were unavailable. We were able to find the missing dependency, the graph clustering algorithm Chinese Whispers [Bie06] in another repository, and updated the dependencies reference. The shutdown repository was replaced with the Maven Central repository [Son25]. For the unavailable dependency versions, we tried the latest and the oldest available versions, but ultimately had to update the tool's Kotlin version. As a consequence, the tool did not compile anymore as the used language version had different standard library functions. After unsuccessfully trying to adopt the tools' code towards the newer language version, we decided to abort the execution of this approach, as we neither had enough knowledge about Kotlin, and the tools' implementation details, nor did we have enough time to invest in gaining it. This tool automatically processes inputs, which are easily attainable, unlike other automated approaches that require inputs in formats specific to their tools. For this reason, we want to eliminate the described hurdles in the future, as this approach could enable comfortable identification of microservice candidates for large projects without much overhead.

7.2 Microservices Backlog

Inspired by the product backlog of user stories, the Microservices Backlog model, by Vera-Rivera et al. [VPAG21] treats service decomposition as a genetic-programming problem. In theory, an accompanying tool automates this solution. However, in practice, the available implementation is not directly provided, and once found, it proved to be unmaintained, undocumented, and written in Portuguese, preventing us from executing it within the scope of this thesis.

7.2.1 Service Identification Methodology

In [VPAG21], the authors introduce the Microservices Backlog (MB) model, a semi-automatic technique that utilizes genetic programming to identify and evaluate microservices from user stories contained in a product backlog. The primary objective is to determine the optimal granularity of microservices by minimizing a composite metric that integrates several key quality attributes.

The quality of a candidate microservices decomposition is quantified by a set of metrics that capture desired properties:

- Coupling (CpT): Measures the degree of interdependency among microservices. Lower coupling implies greater service independence.
- Cohesion ($CohT$): Assesses the internal relatedness of user stories within a microservice, where higher cohesion is preferred.
- Granularity ($WsicT$): Represents the size of a microservice, defined by the maximum number of user stories assigned to it.
- Cognitive Complexity (CxT): Estimates the difficulty of understanding and maintaining a microservice.
- Semantic Similarity (SsT): Quantifies the textual or conceptual similarity among user stories. To align its contribution with the other metrics, it is transformed as $(100 - SsT)$ so that higher similarity results in a lower overall impact.

These metrics are combined into a single objective function, the granularity metric G_m, which is defined as:

$$G_m = \sqrt{(10 \cdot CpT)^2 + CohT^2 + CxT^2 + WsicT^2 + (100 - SsT)^2}.$$

Minimizing G_m implies a decomposition characterized by low coupling, low complexity, appropriate granularity, and high semantic cohesion [VPAG21].

The MB model explores possible decompositions using a genetic programming approach. Each candidate solution is encoded as an assignment matrix (chromosome), where the rows correspond to microservices and the columns to user stories. A binary entry (1 or 0) indicates whether a user story is assigned to a particular microservice. Formally, a chromosome is represented as:

$$\text{chromosome} \in \{0, 1\}^{n \times m},$$

with n being the number of microservices and m the number of user stories.

The genetic algorithm iteratively refines the population through:

1. Reproduction: Combining parent solutions to generate offspring.
2. Mutation: Randomly flipping bits in the chromosome to explore new assignments.
3. Selection: Retaining individuals with lower G_m values (i.e., better fitness).

This iterative process continues until convergence criteria are met, such as a significant portion of the population achieving similar G_m values.

Once the genetic algorithm converges, the best candidate decomposition is selected based on its minimized G_m value. The resulting assignment is visualized in a Microservices Backlog diagram, which graphically presents the distribution of user stories among the candidate microservices, as well as key metrics such as coupling, cognitive complexity, and estimated development time.

7.2.2 Exectution

Like the authors behind Steinmetz [LO20], Vera-Rivera et al. [VPAG21] implemented a tool to automate their approach to a certain extent. While they mentioned the usage of a tool in their paper, no link or reference to an implementation was given. Herrmann faced a similar problem during his thesis [Her24]. He also favored the MB model [VPAG21] and found a repository from two of the authors named like the sought-after tool microservicesbacklog [Viv24]. As expected, we faced the same obstacles as described in the preceding work. The repository has, at the time of this thesis, not been maintained for two years, no documentation was found, and all information was in Portuguese. We were therefore unable to resolve the occurring errors in a reasonable amount of time, hence not pursuing this approach further, as the manual execution would not be feasible.

7.3 Service Cutter

In this section, we cover Service Cutter [GKGZ16a], a tool that leverages a catalog of architectural coupling criteria and graph-clustering algorithms to propose service cuts. We began by modeling the Dispo component's representations in the required JSON format, then ran the too's Docker image [GKGZ15a] to execute four clustering algorithms under different coupling-priority settings. The following subsections expand on the SIA's methodology, the explored configurations, and the range of produced outcomes.

7.3.1 Service Identification Methodology

In [GKGZ16a], the authors propose Service Cutter, a systematic, knowledge-driven approach to decomposing a software system into candidate services. This method supports service identification by processing design artifacts through a set of architecturally significant coupling criteria, constructing a weighted graph of nanoentities, and applying graph clustering techniques to derive candidate service cuts.

Service Cutter is grounded in a catalog of 16 coupling criteria derived from literature, industry experience, and collaborative workshops, which capture the key factors driving service decomposition. These criteria are organized into four categories:

1. Cohesiveness: Criteria that identify common characteristics among nanoentities (for example, semantic proximity, identity, and lifecycle commonality), suggesting they should be grouped.
2. Compatibility: Criteria that flag divergent properties, indicating that certain nanoentities should not reside within the same service.

3. Constraints: Criteria that impose high-impact requirements, such as consistency or security constraints, necessitating either joint inclusion or separation.
4. Communication: Criteria that address the technical cost of remoting, such as evaluating network traffic or mutability.

Nanoentities are the basic building blocks extracted from system specification artifacts (SSAs) such as domain models, use cases, or DDD entities. Each nanoentity may represent data, operations, or artifacts that are later assigned to services.

Service Cutter maps the relationships among nanoentities onto an undirected, weighted graph $G = (V, E)$:

- V is the set of nanoentities.
- E contains edges weighted by the aggregated scores derived from the coupling criteria.

For each pair of nanoentities, individual coupling criteria yield a score ranging from -10 to $+10$, which is then weighted by a priority value. The sum of these weighted scores constitutes the edge weight between two nodes. A high positive weight suggests that the two nanoentities should be grouped within the same service, whereas a negative weight indicates that they should reside in separate services.

Once the weighted graph is built, Service Cutter employs graph clustering algorithms to partition the nanoentities into clusters that represent candidate services. The tool framework is designed to be extensible, supporting multiple clustering techniques such as:

- Girvan–Newman: A deterministic algorithm that removes edges with high betweenness to reveal community structure. [GN02; NG04]
- Leung: A non-deterministic label propagation method that updates node labels based on the most frequent neighbor label. [RAK07]
- Chinese Whispers: A fast, non-deterministic algorithm that iteratively assigns nodes to the most common label among neighbors. [Bie06]
- Markov MCL: An algorithm that simulates random walks using expansion and inflation steps to identify densely connected clusters. [Van08]

These clustering methods identify densely connected subgraphs, each corresponding to a candidate service cut that exhibits high internal cohesion and low external coupling.

7.3.2 Execution

Service Cutter [GKGZ16a] is one of the oldest Service Identification Approaches within the repository of the Architecture Refactoring Helper, published in 2016. The authors explicitly state that the project is no longer maintained and the build is currently broken [GKGZ15a]. However, we had no problems setting up the tool locally, due to the prebuilt Docker images provided.

We decided that we will perform the service identification on a smaller component within the 3LiterPPS-Server®, as the ServiceCutter tool works with User Representations [GKGZ15f], which have specific formats. Providing the whole application as an ERM [GKGZ15d] and defining all use cases with every read and written nanoentity would have been infeasible within our timeframe. Consequently, we discussed what component would be most appropriate for a proof-of-concept execution, with our Head of Development and Support, and involved developers. We decided on the Dispo component because it is relatively loosely coupled within the application and has been causing issues within the monolith. The Dispo component should be extracted from the monolith, decomposed into microservices, and made into an independently acting module.

The component serves as a critical verification and coordination module within the 3LITERPPS® software. It ensures that every production order is supplied with the necessary parts or materials. Keeping track of internal stock, externally sourced components, or outputs produced by other production orders as inventory quantity, combined with a time component. A visual representation of the component structure can be found in Figure 7.1.

The Service Cutter tool requires an Entity Relationship Model as a basis for the service identification. As there has not been an ERM that we could have used, we created one manually and had the component's main developer verify its validity. The transformed ERM, recognized by the ServiceCutter, is available under [Zen23].

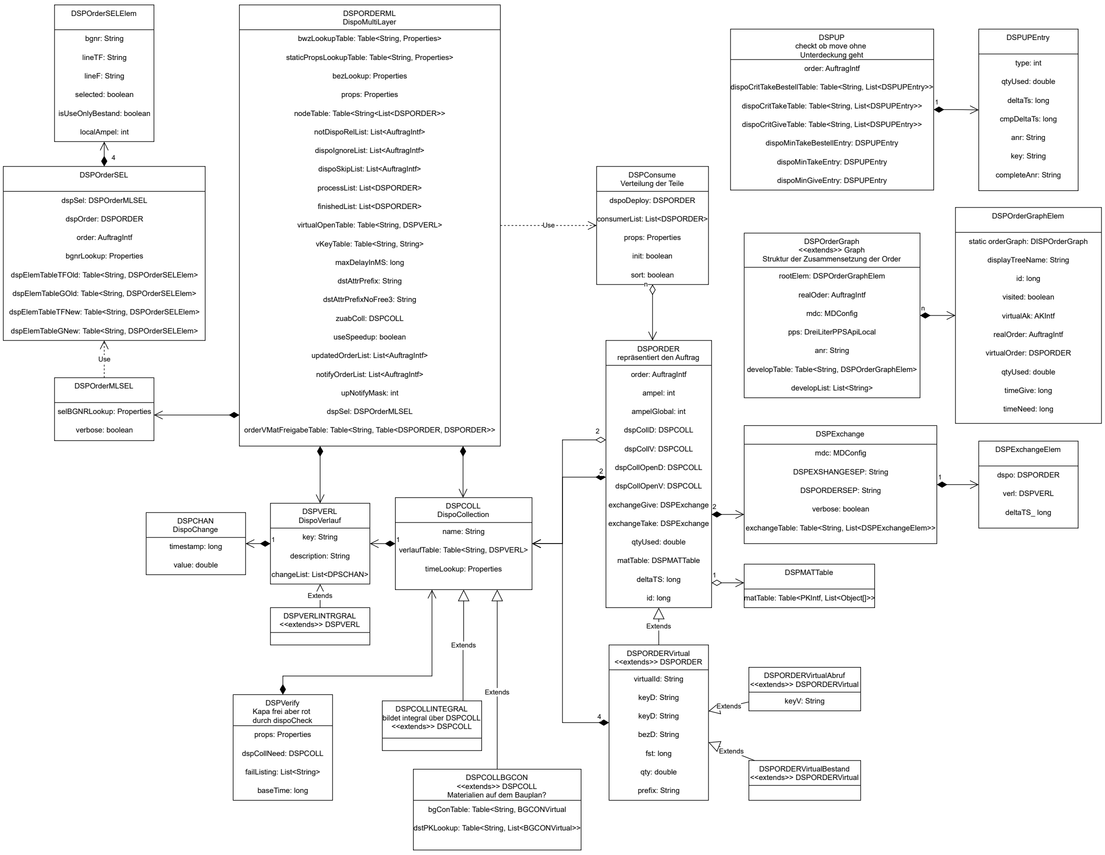

Figure 7.1: Visualization of the Dispo components structure. The used scalable vector graphic is available under [Zen23].

Aside from the ERM, we can upload further user representations to enhance the system and provide more semantics. Out of all documented user representations [GKGZ15f], we were able to provide use cases and aggregates.

We defined the three use cases realized by the Dispo component:

1. Covered by: generate a top-down Dispo-graph for a certain order, creating a time-traceable chart, visualizing material flow that leads to the coverage of this production order
2. Supplying to: generate a bottom-up Dispo-graph for a certain order, creating a time-traceable chart, visualizing material flow that originates from this production order, moving towards orders that this order supplies.
3. Recalculate Dispo: recalculate the dispositive materials in the system and validate all production orders and the timely availability of their required materials. Visualize the feasibility of all production orders based on the timeline of supplying orders and stock.

and transformed them into the Service Cutter format [GKGZ15e], by listing the read and written nanoentities for each use case, which were added to the “DispoUserRepresentations.json” file, available under [Zen23].

Afterward, we defined the aggregate of the component, a Dispochange or “DSPCHAN”, composed of a timestamp and a value, representing the change in absolute quantity of a certain material. As the wiki [GKGZ15b] defines an Aggregate to be the basic elements of transfer or data storage [Fow13], and we did not identify any further composite aggregates, besides the components’ atomic unit. To add this refinement, we appended the user representation JSON file from the previous paragraph.

With the User Representations we provided, we cater to the following Coupling Criteria [GKGZ15c]

- Identity & Lifecycle Commonality
- Semantic Proximity
- Consistency Constraint

Since this component is rather isolated and small, we could not add further User Representations like predefined services or security zones, which will be considered for a cross-component execution of this SIA. Therefore, we can explicitly set the unaffected coupling criteria to the “IGNORE”-state.

At this point, we are ready to generate candidate services with the Service Cutter by setting different priority levels for the three coupling criteria, Ignore and XS-XXL, and trying different clustering algorithms, Leung [RAK07], Girvan-Newman [GN02; NG04], Chinese-Whispers [Bie06], and Markov-MCL [Van08].

7.3.3 Results

To generate candidate service configurations, we explored numerous setups, including edge cases. However, we did not exhaustively evaluate all possible combinations, as this would result in $n \cdot m^k$ total configurations, where:

- n is the number of clustering algorithms,

- m is the number of available coupling criteria, and
- k is the number of weighting options per criterion.

This leads to $4 \cdot 3^7 = 8748$ configurations. Additionally, due to one algorithm being randomized and another non-deterministic, multiple runs would be required for each configuration to achieve representative results. This still omits accounting for the 31 different values for the desired number of services used by the Girvan-Newman algorithm [GN02; NG04].

Leung Service Cuts

The Leung clustering algorithm is inherently non-deterministic [RAK07], which means that even with fixed configuration parameters, repeated executions can yield varying candidate service decompositions. To address this variability, we executed the algorithm multiple times until a consistent pattern emerged.

Using the default configuration, with all coupling criteria set to a medium weight, denoted as "M", the algorithm most frequently partitions the component into a common structure. In this configuration, the `DSPUP` and `DSPUPEntry` nanoentities, as well as the `DSPCOLLBGCON` nanoentities, consistently form individual services, because they are not referenced in any of the use cases. In addition, two distinct services are regularly produced: one service consolidates all nanoentities used exclusively by the "Recalculate Dispo" use case, while another encapsulates all of the remaining nanoentities. A representative candidate service configuration is shown in Figure 7.2.

When we increase the weights for one or more coupling criteria, the resulting configuration exhibits two notable changes. First, a single randomly selected nanoentity is occasionally isolated into its own service. Second, nanoentities that belong to subcomponents with a collection-element relationship, for example, `DSPOrderGraph` with `DSPOrderGraphElem`, `DSPExchange` with `DSPExchangeElem`, or `DSPOrderSEL` with `DSPOrderSELElem`, tend to be clustered together into a single service.

Contrarily, decreasing the coupling criteria weights preserves the fundamental decomposition observed under the default setting while introducing additional variability. In these cases, we see an increased number of services where multiple nanoentities from the same class are grouped separately. This leads to a less cohesive and more interdependent service structure.

Finally, when all coupling criteria are ignored, the algorithm produces the most granular decomposition possible. Each nanoentity is assigned to its own service. This outcome is illustrated in Figure 7.3.

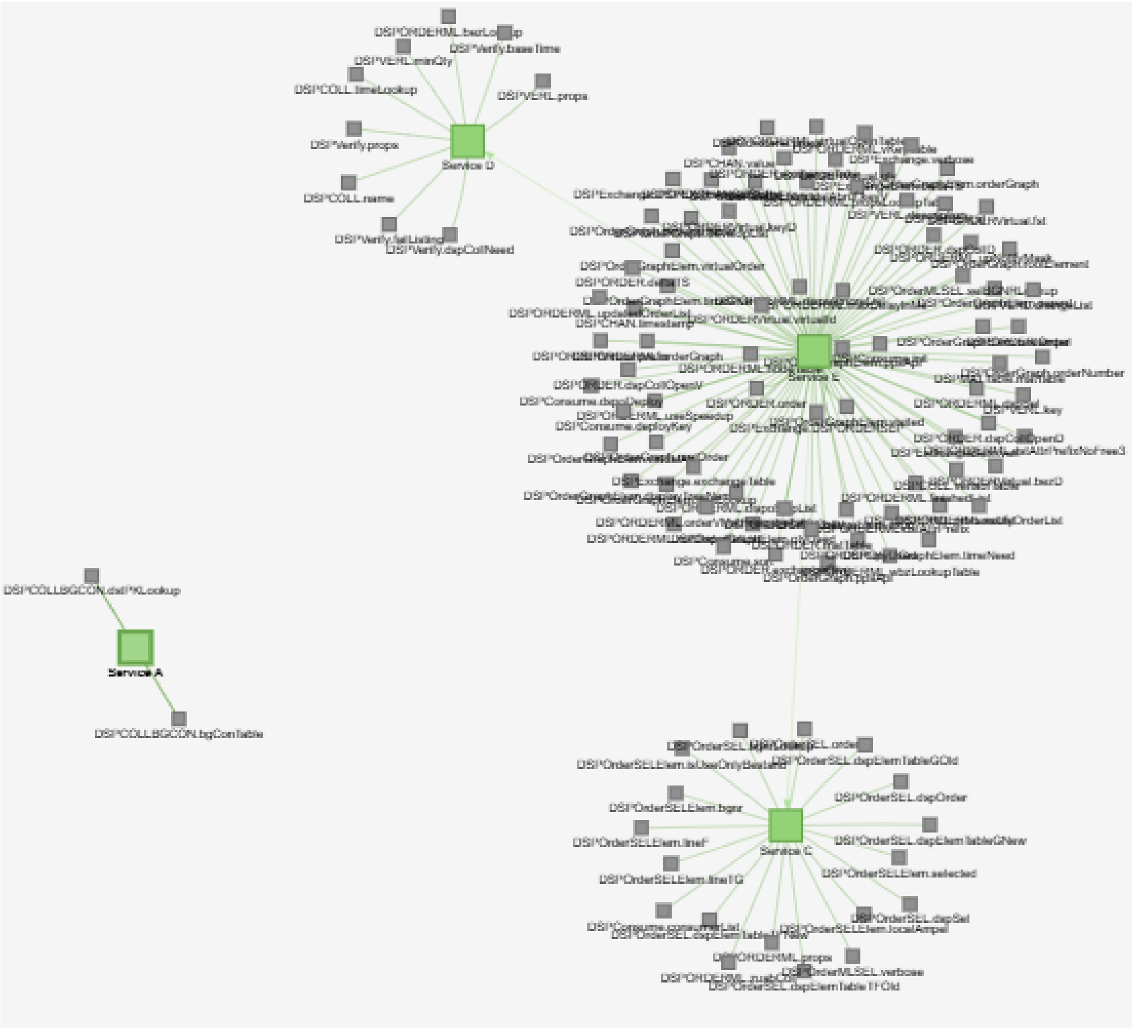

Figure 7.2: Visualization of the decomposed Dispo component, generated by the Service Cutter [GKGZ16a], using the Leung clustering algorithm. A detailed JSON representation is available under [Zen23].

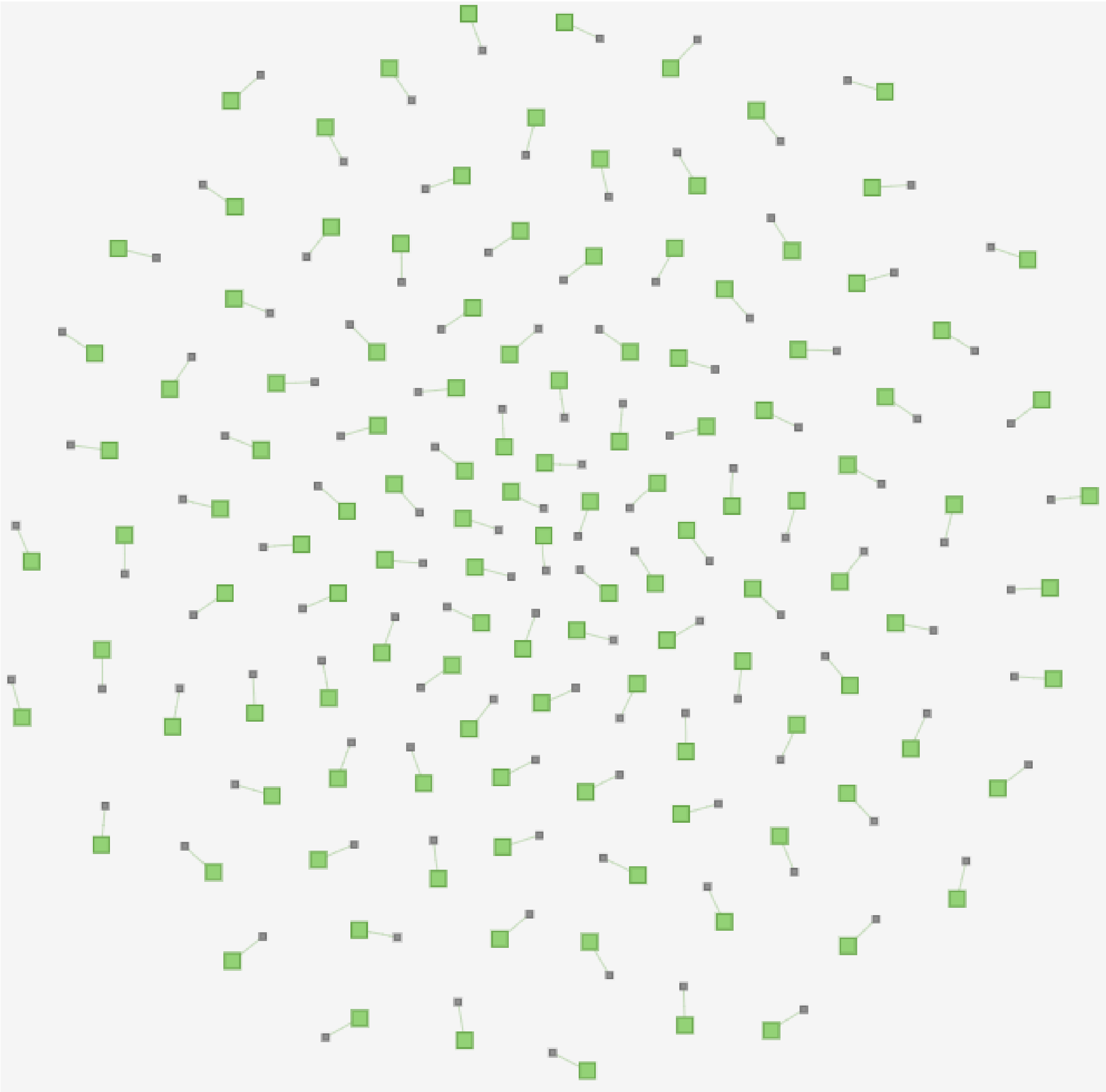

Figure 7.3: Visualization of the decomposed Dispo component with all coupling criteria ignored, generated by the Service Cutter [GKGZ16a], using the Leung, Chinese-Whispers, or Markov MCL clustering algorithm. No clusters are formed, and every nanoentity is encapsulated separately in one of 122 services. A detailed JSON representation is available under [Zen23].

Chinese Whispers Service Cuts

Because the Chinese Whispers algorithm [Bie06] is randomized, the decompositions it produces show a degree of inconsistency similar to the Leung decompositions. Under the default configuration, the candidate services generally follow a pattern similar to that of the Leung algorithm. Two additional services distinguish the results of the two algorithms. In nearly every generated service cut, one service encapsulates the `DSPOrderGraph` and `DSPOrderGraphElem` nanoentities, while another service includes the nanoentities of the `DSPORDER` and the `DSPORDERVirtual` classes, as seen in Figure 7.4.

When the weights of the coupling criteria are increased, the candidate decomposition either remains consistent with the default values, retaining the separate `DSPORDER` and `DSPORDERVirtual` service, or merges said nanoentities into the larger service that contains nanoentities referenced by all use cases. The resulting configuration is visualized in Figure 7.5.

When all coupling criteria are ignored, the Chinese Whispers algorithm produces a decomposition identical to that of the Leung algorithm. This behavior is demonstrated in Figure 7.3.

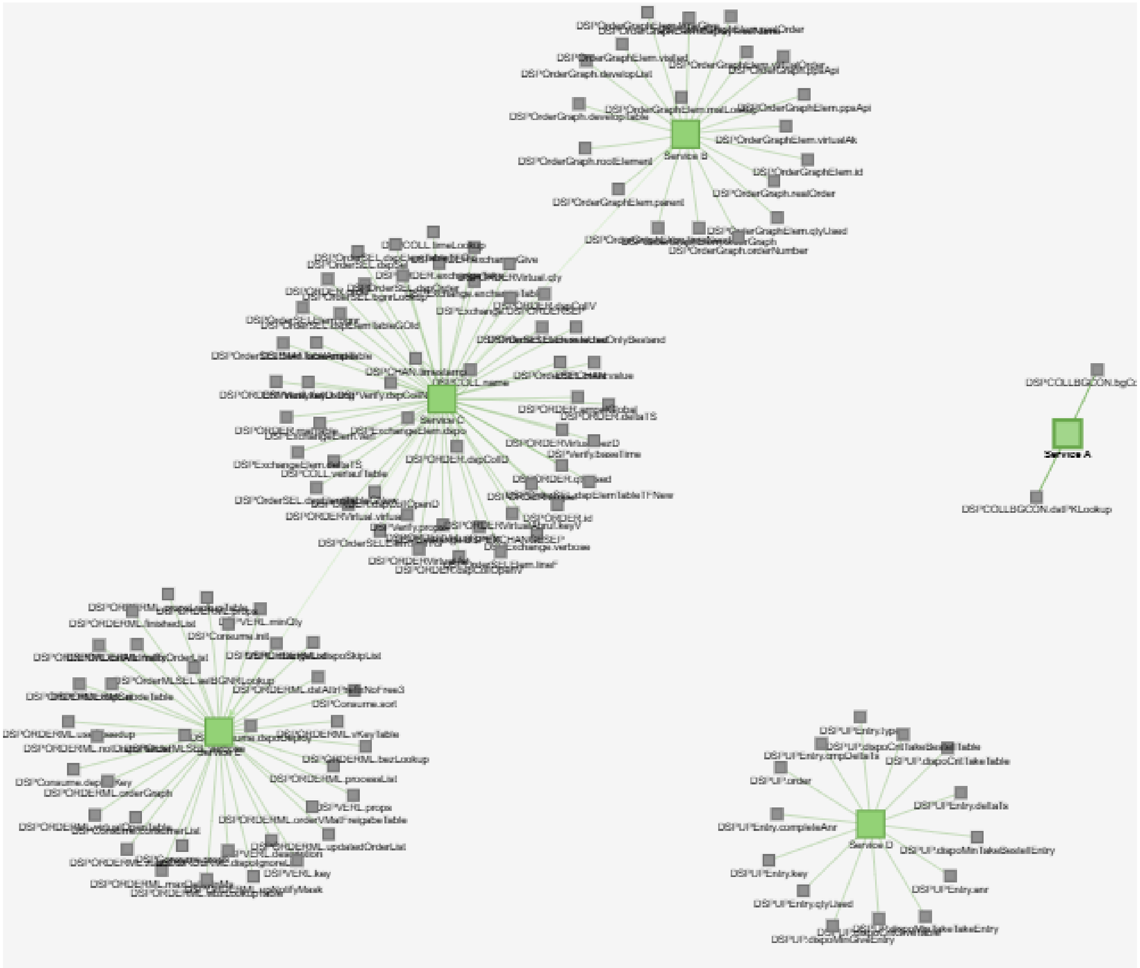

Figure 7.4: Visualization of the decomposed Dispo component, generated by the Service Cutter [GKGZ16a], using the Chinese-Whispers clustering algorithm. A detailed JSON representation is available under [Zen23].

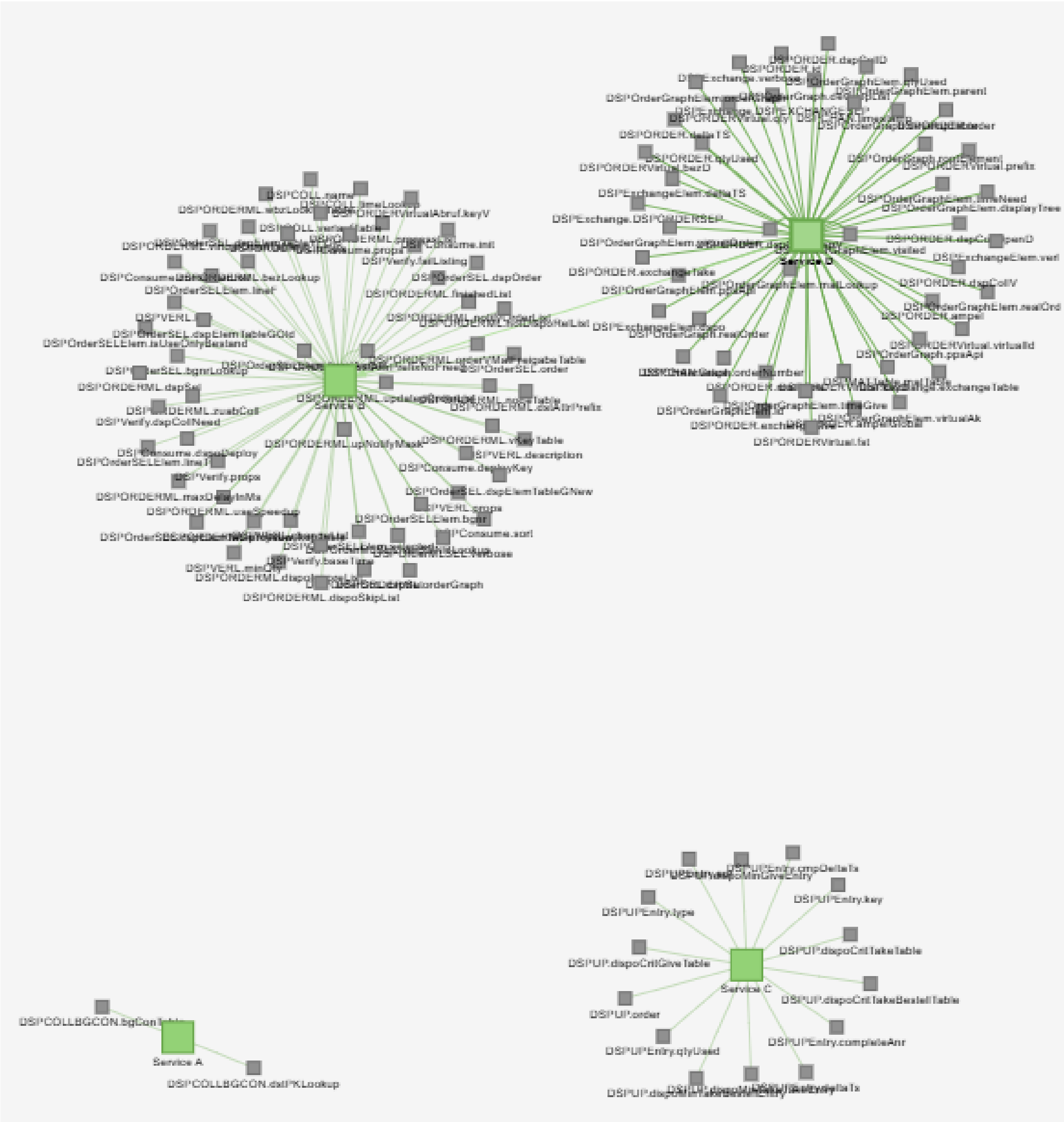

Figure 7.5: Visualization of the decomposed Dispo component with all coupling criteria set to their maximum weight, generated by the Service Cutter [GKGZ16a], using Chinese-Whispers clustering algorithm. A detailed JSON representation is available under [Zen23].

Markov MCL Service Cuts

In contrast to the Chinese Whispers and Leung, the Markov MCL clustering algorithm [Van08] deterministically produces service cuts for the Dispo component.

The algorithm again isolates the `DSPUP` nanoentities and divides the `DSPCOLLBGCON` nanoentities into two separate services, thereby distinguishing them from the nanoentities relevant to the use cases. In addition, the remaining four services are organized into two services for the use cases, one data model service, and one service for `DSPOrderSEL` nanoentities. This decomposition is illustrated in Figure 7.6.

Neither an increase nor a decrease in the coupling criteria weights affected the resulting service cuts, except when all coupling criteria were ignored. In that situation, the algorithm produced a fully decomposed candidate configuration, shown in Figure 7.3.

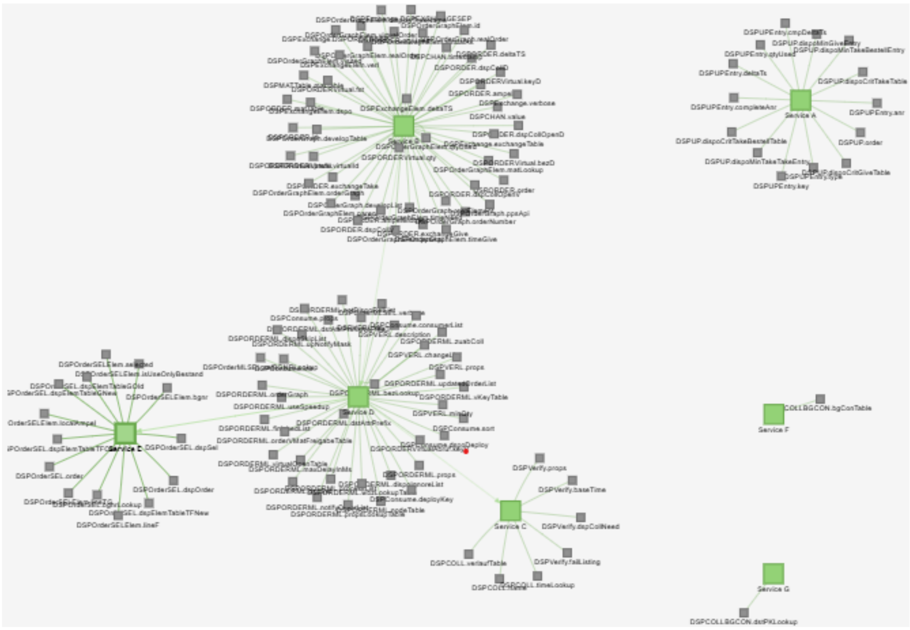

Figure 7.6: Visualization of the decomposed Dispo component with all coupling criteria weights on default, generated by the Service Cutter [GKGZ16a], using the Markov MCL clustering algorithm. A detailed JSON representation is available under [Zen23].

Girvan-Newman Service Cuts

The Girvan-Newman clustering algorithm [GN02; NG04] has a property that none of the other clustering algorithm has. You can select a desired number of services for the algorithm to generate. The Service Cutter tool [GKGZ16a] lets you choose any number between 0 and 30. To cover the appropriate range of services, we set a lower boundary of three, as we found that the algorithm yields three services for a requested service quantity of one, two, and three. For the upper limit, we generated service cuts until newly generated services had low intra-service cohesion, which started to occur at a service quantity larger than six. Similar to all other candidate service generations, except for the split `DSPCOLLBGCON` service, this algorithm also generates the two services that are independent of the use cases. The remaining nanoentities are encapsulated within a third service. This minimal Girvan-Newman decomposition is displayed in Figure 7.7.

Adding more services results in the use case service being iteratively decomposed. First, the `DSPVerify` nanoentities get extracted, afterward the `DSPOrderSel`, lastly the sixth service encapsulates the `DSPExchange` nanoentities. The maximum Girvan-Newman decomposition is shown in Figure 7.8.

Similar to the Markov MCL algorithm, changing the coupling criteria weights did not change the output candidate services. Ignoring all coupling criteria results in the algorithm not terminating.

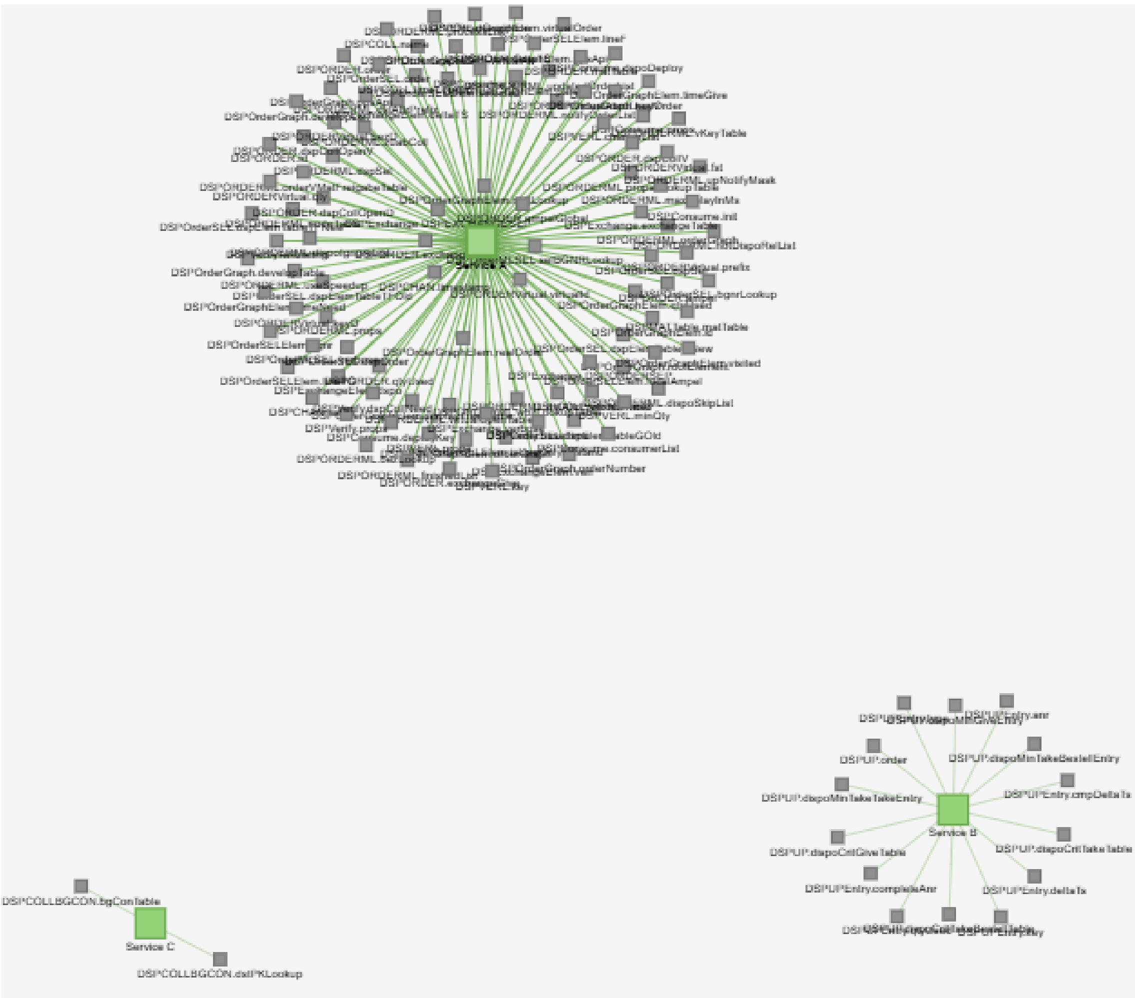

Figure 7.7: Visualization of the decomposed Dispo component, generated by the Service Cutter [GKGZ16a], using the Girvan-Newman clustering algorithm with a requested service quantity of three. A detailed JSON representation is available under [Zen23].

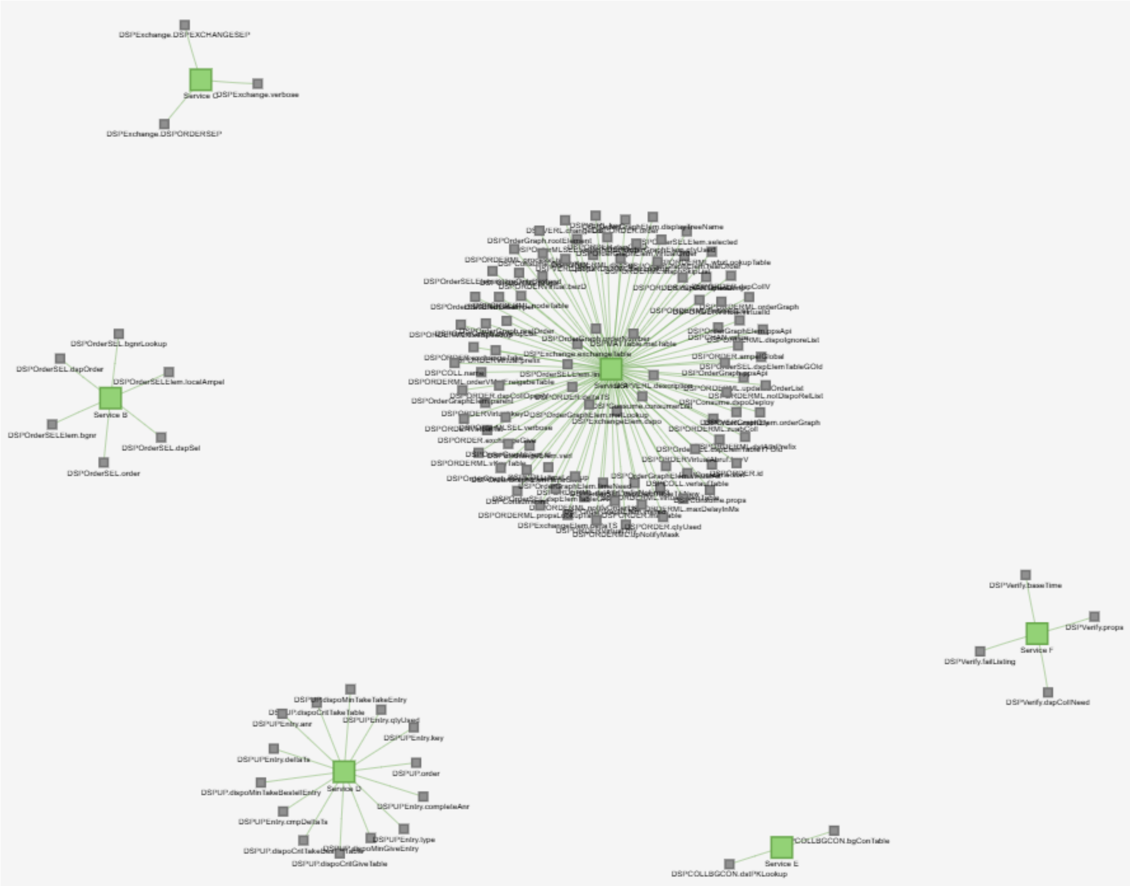

Figure 7.8: Visualization of the decomposed Dispo component, generated by the Service Cutter [GKGZ16a], using the Girvan-Newman clustering algorithm with a requested service quantity of six. A detailed JSON representation is available under [Zen23].

7.4 Result Discussion

Since we generated candidate service lists for only one SIA, our comparison focuses solely on the service cuts produced by the Service Cutter approach described by Gysel et al. [GKGZ16a] rather than comparing service candidates across the three selected approaches. The discussion of the service cuts took place with the Head of Development and Support, who has extensive knowledge of the 3LiterPPS-Server®, and the Dispo component itself.

We first consider the candidate service list for the fully decomposed Dispo component. Because the algorithms were provided no information on nanoentity relationships, each nanoentity was treated as fully independent of every other. The result was 122 services, each containing a single nanoentity. This outcome not only fails to improve the component's existing architecture but also leads to impractical and highly coupled services. We therefore discard this candidate service list and do not include it in further comparisons. It is mentioned only for the sake of completeness.

For all other generated decompositions of the Dispo component, we observe that the nanoentities for the DSPUP, DSPUPEntry, and DSCOLLBGCON classes were separated from services associated with the use cases. This separation occurred because the use cases do not directly read or write these nanoentities. An unexpected outcome was that the DSPCOLLBGCON nanoentities were divided into two separate services without an apparent linkage between them when using the Markov MCL algorithm. Apart from this irregularity, we agree with the separation of these nanoentities from those accessed in the use cases. These elements are instead employed by other components' use cases, providing information that is encapsulated within the Dispo component.

We found the service granularity produced by the Girvan-Newman candidate service lists unsatisfactory. Beginning with the minimal decomposition of three services, shown in Figure 7.7, we obtained one service for all use case nanoentities, with the nanoentities of individual classes being removed one after another as the target service quantity increased, rather than grouping nanoentities from different use cases into distinct services. This approach resulted in an imbalance with many small services alongside on large service.

The outputs of the Chinese Whispers algorithm separated the DSPOrderGraph nanoentities associated with the production order-specific "Supplying to" and "Covered by" use cases. In addition, these nanoentities are semantically connected through their collection-element relationship. With lower-weighted coupling criteria, the nanoentities referenced solely in the "Recalculate Dispo" use case were isolated into another service. This resulted in a service cut where the responsibilities for the use cases were distributed into separate services connected by common nanoentities. We regard this method of partitioning as superior to the candidate services produced by the Girvan-Newman algorithm.

The service cuts generated by the Leung clustering algorithm also isolated the responsibility for the "Recalculate Dispo" use case into a separate service. However, they grouped the shared nanoentities with those of the production order-specific use cases into a single service, thereby decreasing the cohesion of that service. We support the practice of encapsulating collection-element classes into separate services, present in the Leung service cuts, for higher weighted coupling criteria, if they are logically separable from the use case services.

Our preferred decomposition was produced by the Markov MCL algorithm. This approach not only separated the nanoentities used by the "Supplying to" and "Covered by" use cases from those used by the "Recalculate Dispo" use case and from the shared nanoentities, but it also distinguished portions of the data model involving the DSPCOLL nanoentities. This result could be further improved by extracting all DSPCOLL, DSPVERL, and DSPCHAN nanoentities, since they provide the joint data structure for changes in material stock. In addition, other data model classes and their nanoentities might be grouped into one or more cohesive data model services.

Based on our analysis, we propose a fusion of ideas from multiple clustering algorithms. We suggest decomposing the Dispo component into data model services, similar to the partitioning observed in the Markov MCL decomposition, while also extracting collection-element structures as seen in the Leung algorithm. We recommend separating use case responsibilities by grouping the corresponding nanoentities into distinct services, as observed in the outputs of the Chinese Whispers and Markov MCL algorithms. Finally, we propose isolating the nanoentities of the DSPUP, DSPUPEntry, and DSPCOLLBGCON classes from those accessed by the provided use cases. By decomposing the Dispo component in this manner, we obtain a component architecture that is subjectively well granular, properly separates use case concerns, and is sufficiently decoupled to justify its decomposition.

8 Evaluation

We evaluate the results of this thesis by separating it into three sections. Starting with the evaluation of the Architecture Refactoring Helpers [Fri24a], by elucidating its guiding role, and addressing areas for improvement. Moreover, we classify the impact of the organizational context in which this thesis was conducted, before covering the threats to validity.

8.1 Architecture Refactoring Helper

We evaluate the Architecture Refactoring Helper in three distinct steps. First, we elaborate on how it supported the selection of suitable Service Identification Approaches, based on stakeholder concerns. Afterward, we point out limitations that have arisen in the course of this thesis, before we suggest adjustments to the tool's functionality that would improve its value for constrained environments.

8.1.1 Strengths and General Usability

The Architecture Refactoring Helper [Fri24a] provided a structured approach that supported the development of a migration plan for the 3LiterPPS-Server®, by guiding the user towards the selection of a Service Identification Approach suitable for their needs. Its three-phase structure, including "System comprehension", "Strategy Definition", and "Architecture Definition", allowed a clear and systematic breakdown of the evaluation process. This study did not aim to perform a full migration to microservices, but instead focused on evaluating the ARH's capabilities and developing a viable migration plan through the identification and selection of a suitable SIA.

The tool's scenario-based quality attribute selection was particularly effective in translating stakeholder concerns into actionable filtering preferences. This helped narrow down SIAs that fit organizational goals and highlighted the impact of architectural decisions.

8.1.2 Usability Issues and Filtering Mechanics

Despite its benefits, the ARH has several usability issues. Some tooltips in the preference filtering interface were misleading, displaying explanations for different entries than the ones hovered over, while others had no explanation at all. This made it difficult to confidently set up filtering preferences, as definitions had to be looked up in external sources that potentially differ from the definitions used when classifying the repository's SIAs.

The “Matches” metric, a central element of SIA ranking, is skewed by the way quality attributes are aggregated. Selecting a quality sub-attribute like Modularity for a scenario automatically includes the parent attribute Maintainability. If an SIA is categorized with only the parent attribute, it still receives matches for all related sub-QAs, queried for by the user. This inflates the perceived alignment between SIAs and user requirements.

Additionally, some lower-level quality attributes were not represented by SIAs at all, meaning that specific stakeholder concerns could not be reflected in the filtering process. Furthermore, the repository contains Service Identification Approaches that were not assigned any, or just upper-level quality attributes, resulting in them performing worse when querying as intended, even though one might argue that microservice architectures inherently reflect quality attributes like scalability or modularity. Moreover, some SIAs are linked to tools that are not publicly maintained or executable in their current form, making them unusable despite their high ranking.

Moreover, we identified misclassifications in the repository. For example, the SIA 2101 [DEF+21] is listed as requiring source code, even though it does not. SIAs 1706 [CLL17] and 1907 [LZJ+19] are incorrectly shown as using Service Cutter [GKGZ16a], although they only mention it in their related works or evaluation.

8.1.3 Tool Potential and Recommendations

Despite the limitations observed, the ARH has strong potential as a tool for structured architecture migration planning. Its phased process, integration of quality attributes, and broad repository of SIAs address a gap in current migration tooling. To improve its usability and relevance in constrained industrial environments, we propose the following enhancements:

Total exclusion option for preferences

Currently, preferences marked as “excluded” do not eliminate SIAs relying on them. A strict exclusion feature would allow users to differentiate between undesirable and strictly ruled out preferences, allowing users to fully filter out SIAs that depend on artifacts such as source code, or specific diagrams when these are not available or usable.

Independent weighting for scenario unrelated preferences

Attributes like automation level, tool support, or used techniques are treated uniformly. A feature to assign custom weights would allow for prioritized preferences, reflecting the actual preferences more accurately.

Revised metrics

Overhauling the current metrics, which inflate the suitability of certain SIAs through artificial matches, would improve the alignment of organizational requirements and recommended SIAs. Therefore, we suggest the “Matches” metric to attribute matches to those, and only those, preference filters that were attributed to the Service Identification Approaches during their categorization.

Improved categorization and consistency

A standardized categorization of SIAs is necessary to ensure representative recommendations. We propose re-categorizing the current SIAs once this standard process is enabled, as misclassified requirements like source code dependencies, missing quality attributes, or wrongly associated attributes and tools, reduce the meaningfulness of recommended approaches.

8.2 Organizational Context and Constraints

Due to confidentiality concerns, we excluded source code from the analysis. This disqualified all SIAs that require direct code inspection or code-based analysis, without intermediate artifacts, as we could not provide the necessary inputs for a reproducible study. Likewise, database-driven SIAs could not be applied since the 3LiterPPS-Server® does not use a conventional database schema for persistence. The existing backend architecture relies on custom data storage mechanisms that do not fit standard database assumptions. Furthermore, the system's API is not an HTTP API and, therefore, not documented using standards like OpenAPI. As a result, SIAs that depend on such specifications were excluded as well. These constraints significantly limited the pool of applicable SIAs and, in combination with unmaintained SIA tools, ultimately required us to select an approach with fewer matching quality attributes simply because no other options fit the technical boundaries.

8.3 Threats to Validity

We structure the Threats to Validity into the four Validity types described by Runeson and Höst [RH09]: Construct Validity, Internal Validity, External Validity, and Reliability.

8.3.1 Construct Validity

Construct validity concerns whether the study truly measures the intended concepts [RH09]. This study's main objective was to evaluate the Architecture Refactoring Helper and identify a suitable Service Identification Approach under real-world constraints. Thus, construct validity concerns whether the steps taken and data used truly support this evaluation. One threat is the interpretation of quality attributes in ARH. Because the tool automatically includes parent attributes when selecting sub-quality attributes, the filtering process may give a false impression of precise alignment between stakeholder concerns and SIA characteristics. This automation simplifies input but can lead to inflated match scores and a mismatch between what the user intended to prioritize and what the ARH processes. Another issue arises from inconsistent or incomplete SIA categorization. Some SIAs have no quality attributes assigned, while others are labeled only with more generic upper-level QAs. Since the evaluation aimed to filter SIAs based on quality attribute alignment, these inconsistencies limit the credibility of the filtering mechanism and introduce doubt as to whether the final recommendations truly reflect the intended priorities. Lastly, while scenario-based inputs were used effectively, there's a risk that stakeholders interpreted architectural terms differently, especially if not all participants shared a strong technical background. This could impact the accuracy of the scenarios used to guide preference selection, introducing subtle mismatches between what was intended and what ARH filtered for.

8.3.2 Internal Validity

Internal validity pertains to establishing credible causal relationships [RH09]. In exploratory case studies, this refers to the degree to which the observations and interpretations are trustworthy and not influenced by hidden or uncontrollable factors. Although this study did not aim to establish causal relationships, it involved subjective interpretation of stakeholder scenarios, tool outputs, and preference configurations, all of which could introduce unintended bias. The idea of moving towards a microservice-based architecture had been discussed within the organization prior to the start of this thesis. Although the goal here was not to justify a decision but to evaluate a tool and generate a structured migration plan, this context might have introduced bias in the way stakeholders prioritized certain quality attributes or interpreted scenarios. Another internal threat is tied to the way the Architecture Refactoring Helper calculates Service Identification Approach matches. Due to the previously described skewing, users can be misled by the displayed number of matches compared to the actual classification. This effect is worsened through the presumably unstandardized classification of SIAs. Additionally, not all relevant stakeholder perspectives could be represented during the architecture review session. Due to an unforeseen operational need, the Head of Development and Support was unable to attend. While the session still included valuable input from various roles, the absence of a senior technical leadership voice may have limited the completeness of architectural considerations in the scenario collection and prioritization. Additionally, the limited number of applicable SIAs due to project constraints narrowed the selection field. While these constraints are real, they also limit the opportunity to explore alternatives. The final SIA was chosen from a small selection pool, and while justified, the selection could still reflect availability rather than sought optimality. These factors suggest that while the study was carefully conducted, some findings may have been subtly influenced by contextual or tool-specific dynamics rather than by objective technical superiority.

8.3.3 External Validity

External validity addresses the generalizability of the finding to other contexts [RH09]. The constraints in our case study ruled out a substantial number of SIAs. In other projects with full access to these artifacts, a very different subset would be viable. This restricts the broader applicability of the recommendation results, which are inherently tied to our specific application.

8.3.4 Reliability

Reliability reflects the repeatability of the study under the same conditions [RH09]. In this case, steps were taken to ensure high reliability. All JSON files used with the Service Cutter tool were preserved and made available, enabling others to reproduce the service decomposition exactly as it was conducted. Additionally, all filtering decisions and preference settings in the ARH were explicitly documented and justified, ensuring transparency throughout the process.

However, there are still minor reliability concerns tied to tool usability. For instance, several ARH tooltips displayed incorrect or unrelated descriptions, which could lead to user confusion during preference configuration. Moreover, the absence of a strict exclusion option in ARH's preference filtering means that even explicitly unwanted approaches may still appear in results, potentially affecting reproducibility when strict constraints are assumed. Moreover, the reproduction of results within the Service Cutter tool [GKGZ16a] is impacted due to the application of non-deterministic clustering algorithms.

Overall, while the study setup is well-documented and reproducible, minor improvements in tool interface clarity and filtering control would further strengthen its reliability.

9 Conclusion

We conclude our thesis, summarizing our findings and their beneficiaries, stating limitations, and lessons learned, as well as giving an outlook on future work.

9.1 Summary

This thesis evaluated the Architecture Refactoring Helper [Fri24a] in an industrial context, by performing a collaborative case study with the LF CONSULT GmbH [LF 24], to support the migration planning towards a microservices architecture for the 3LiterPPS-Server® application in a structured way. Rather than conducting a full migration, the study focused on assessing the tool's practical value and identifying a fitting Service Identification Approach under real-world constraints. Doing so, we performed a proof of concept decomposition of a component within the monolithic backend application, the 3LiterPPS-Server®.

The ARH proved to be a helpful and structured tool for guiding architectural decisions. Its phased methodology, quality attribute-driven filtering, and integration of stakeholder input provided a systematic approach to narrowing down migration strategies. Despite a limited selection of applicable SIAs, the process led to a well-reasoned choice, namely the Service Cutter approach [GKGZ16a], thus affirming **RQ1**. Input artifacts for the service identification steps were created manually and documented thoroughly to ensure reproducibility.

However, the evaluation revealed several limitations in the tool's functionality (**RQ2**). These included the aggregation of matches for quality attributes, reflected in a limited expressiveness of the SIA ranking. Additionally, the ARH user interface presented usability issues, such as incorrect tooltips, the lack of preference weighting, and exclusion. These aspects collectively impacted the precision of the filtering and recommendation process. We proposed suggestions to eliminate the limitations found in this case study in Section 8.1.3 (**RQ2**).

9.2 Benefits

The results of this thesis provide contributions that can benefit both practitioners and researchers working in the field of software architecture, especially in the context of legacy system modernization.

For software architects, the thesis offers a documented and structured process for conducting migration planning using high-level system knowledge. It demonstrates how to work with scenario-driven quality attribute filtering and how to select a Service Identification Approach even in the absence of source code or standard HTTP APIs, due to confidentiality or availability concerns. Serving as an example for architectural evaluation in constrained, real-world settings.

For organizations, the work shows that migration planning is still feasible even when key system artifacts like source code or database schemas are unavailable. The thesis provides a reusable and reproducible process, including the selection of a Service Identification Approach, Service Cutter, that is applicable in similarly restricted environments.

For tool users and evaluators, this thesis contributes a critical assessment of the Architecture Refactoring Helper, identifying its strengths and exposing practical limitations such as misleading metrics, flawed QA propagation, and usability issues. These findings can help teams make informed decisions when relying on architecture decision-support tools.

For the research community, the study delivers a real-world case study that highlights the gap between theory and practice in the selection and application of SIAs. It adds to the understanding of how architecture migration tools perform under practical constraints and offers a foundation for refining both methodologies and tool support.

For the collaborating organization, LF CONSULT GmbH [LF 24], the study delivers a proof of concept decomposition for a component of their monolithic backend application, which, at a later point in time, will be implemented as a separate service. Furthermore, the decomposition process can be extended to the whole application, to create an intermediate modulith [Sil22], that can be decomposed into a microservices architecture iteratively, applying the Strangler Pattern [Fow04].

Rather than only demonstrating the use of the ARH, this thesis delivers a broader contribution by surfacing practical challenges in using the tool and provides recommendations for improvements from an industry use case point of view.

9.3 Limitations

While the evaluation produced meaningful insights, several limitations affect the scope and generalizability of this work. The methodology and outcomes were closely tied to the specific context of the 3LiterPPS-Server®, a system with a constrained set of accessible artifacts, a unique architecture, as well as the historical evolution resulting in custom in-house developed solutions predating standardized language features. Most notably, the unavailability of certain input artifacts significantly reduced the number of applicable Service Identification Approaches.

The selected SIA, while justifiable within this environment, may not represent the most ideal approach in a setting with broader access to technical inputs.. This means that the findings, particularly regarding the suitability of Service Cutter [GKGZ16a], may not hold in contexts where source-level or data-driven decomposition techniques are viable.

Another limitation stems from the ARH itself. Its matching mechanism and quality attribute handling introduced some ambiguity in interpreting the output. The inclusion of non-present quality attributes, along with inconsistencies in SIA categorization, created a degree of uncertainty around the precision of the filtering process. Although these issues were critically examined, they still affect the clarity of the resulting rankings and could influence outcomes in other applications of the tool.

In addition, while the architecture review process was informed by multiple stakeholder perspectives, it did not include all relevant roles, due to unforeseen operational demands. As a result, some higher-level technical or strategic priorities may not have been fully represented in the scenario creation, discussion, and prioritization, potentially limiting the completeness of the input used to guide SIA filtering.

These limitations do not invalidate the approach or the conclusions drawn, but they do define the boundaries within which the results can be confidently interpreted. This evaluation remains valuable for constrained, industry-like settings, but should be complemented with further studies in different organizational and technical contexts.

9.4 Lessons Learned

Throughout this thesis, several valuable lessons emerged regarding both the practical application of migration tooling and the process of conducting an architectural evaluation in a real-world setting.

One major insight was the need to critically assess the practical applicability of Service Identification Approaches. While many methods presented in academic literature appear promising, some are no longer maintained, lack available implementations, or rely on tools that cannot be executed at this point. This emphasized the importance of verifying not just the conceptual soundness of an approach, but also its usability and availability.

Another takeaway came from preparing inputs for the selected approach, Service Cutter [GKGZ16a]. While the tool offers a clear structure, creating valid input artifacts involved significant manual effort and human expertise regarding the targeted component. This highlighted how small inconsistencies or misunderstandings during modeling can affect downstream results. At the same time, it showed the value of carefully documenting inputs and decisions to support transparency and reproducibility.

A further realization concerned the subtle ways bias can be introduced, especially when working with tools that rely on stakeholder input or abstract configuration layers. For example, selecting and prioritizing quality attributes in the ARH was not purely objective, as it depended on how scenarios were framed and how stakeholders interpreted architectural terms, as well as the underlying attitude towards the migration process itself. Similarly, tool mechanisms like implemented metrics can shape outcomes in ways the users might not be fully aware of. This demonstrated the importance of remaining conscious of potential bias and regularly questioning how decisions are influenced.

In summary, this thesis reaffirmed the importance of pairing structured methods with critical reflection, staying aware of where assumptions and bias may enter the process, and being prepared to adapt to practical limitations.

9.5 Future Work

Several promising directions exist for future work, both at a practical level within the organization and in terms of further development of the Architecture Refactoring Helper.

A natural continuation would be to migrate the Dispo component by implementing the microservice decomposition proposed during this thesis. This would not only validate the selected Service Identification Approach in a live setting but also serve as a foundation for further refinements of the overall migration strategy, while validating the third phase of the Architecture Refactoring Helper in suggesting recommended patterns and best practices.

Expanding the scope of the evaluation, it would be valuable to generate service cuts for the entire 3LiterPPS-Server® application, not just for one selected component. This would enable the introduction of more semantic input in the form of user representations like shared owner groups, predefined services, or separated security zones [GKGZ15f].

Further, we suggest collaboration with Jonas Fritzsch, the developer of the Architecture Refactoring Helper [Fri24a], to translate the insights and identified shortcomings into concrete improvements. Based on this thesis, there is strong potential for refining the quality attribute matching logic, improving SIA categorization, and extending the filtering configuration to better support constrained environments.

Lastly, given the effectiveness of the ARH as a guiding framework, and a rightful interest by organization internal component owners, it would be worthwhile to apply it to other components or applications within the broader 3LiterPPS® ecosystem. This could help standardize migration processes across the organization and create internal best practices informed by repeated structured evaluations, based on academic approaches.

These directions aim to transition this thesis from a planning and evaluation effort into actionable improvements and organizational impact, while also contributing to the evolution of tool-supported architecture refactoring.

Bibliography

[AIE19] M. Abdullah, W. Iqbal, A. Erradi. "Unsupervised learning approach for web application auto-decomposition into microservices". In: Journal of Systems and Software 151 (2019), pp. 243–257. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2019.02.031. URL: https://www.sciencedirect.com/science/article/pii/S0164121219300408 (cit. on pp. 12, 42).

[ART21] A. B. Alaasam, G. Radchenko, A. Tchernykh. "Refactoring the monolith workflow into independent micro-workflows to support stream processing". In: Programming and Computer Software 47.8 (2021), pp. 591–600 (cit. on p. 13).

[BDW99] L. C. Briand, J. W. Daly, J. K. Wust. "A unified framework for coupling measurement in object-oriented systems". In: IEEE Transactions on software Engineering 25.1 (1999), pp. 91–121 (cit. on p. 47).

[BGD17] L. Baresi, M. Garriga, A. De Renzis. "Microservices identification through interface analysis". In: Service-Oriented and Cloud Computing: 6th IFIP WG 2.14 European Conference, ESOCC 2017, Oslo, Norway, September 27-29, 2017, Proceedings 6. Springer. 2017, pp. 19–33 (cit. on p. 42).

[Bie06] C. Biemann. "Chinese whispers-an efficient graph clustering algorithm and its application to natural language processing problems". In: Proceedings of TextGraphs: the first workshop on graph based methods for natural language processing. 2006, pp. 73–80 (cit. on pp. 16, 48, 51, 53, 56).

[CCRZ23] M. Camilli, C. Colarusso, B. Russo, E. Zimeo. "Actor-driven decomposition of microservices through multi-level scalability assessment". In: ACM Transactions on Software Engineering and Methodology 32.5 (2023), pp. 1–46 (cit. on p. 45).

[CLL17] R. Chen, S. Li, Z. Li. "From monolith to microservices: A dataflow-driven approach". In: 2017 24th Asia-Pacific Software Engineering Conference (APSEC). IEEE. 2017, pp. 466–475 (cit. on p. 66).

[DB22] H. Dinh-Tuan, F. Beierle. "MS2M: A message-based approach for live stateful microservices migration". In: 2022 5th Conference on Cloud and Internet of Things (CIoT). IEEE. 2022, pp. 100–107 (cit. on p. 13).

[DBFP18] A. A. C. De Alwis, A. Barros, C. Fidge, A. Polyvyanyy. "Discovering microservices in enterprise systems using a business object containment heuristic". In: On the Move to Meaningful Internet Systems. OTM 2018 Conferences: Confederated International Conferences: CoopIS, C&TC, and ODBASE 2018, Valletta, Malta, October 22-26, 2018, Proceedings, Part II. Springer. 2018, pp. 60–79 (cit. on p. 42).

[DBPF18] A. A. C. De Alwis, A. Barros, A. Polyvyanyy, C. Fidge. "Function-splitting heuristics for discovery of microservices in enterprise systems". In: Service-Oriented Computing: 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings 16. Springer. 2018, pp. 37–53 (cit. on p. 42).

[De 19] L. De Lauretis. "From monolithic architecture to microservices architecture". In: 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE. 2019, pp. 93–96 (cit. on p. 43).

[DEF+21] M. Daoud, A. El Mezouari, N. Faci, D. Benslimane, Z. Maamar, A. El Fazziki. "A multi-model based microservices identification approach". In: Journal of Systems Architecture 118 (2021), p. 102200. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2021.102200. URL: https://www.sciencedirect.com/science/article/pii/S1383762121001442 (cit. on pp. 11, 44, 66).

[FBH+22] J. Fritzsch, J. Bogner, M. Haug, S. Wagner, A. Zimmermann. "Towards an architecture-centric methodology for migrating to microservices". In: International Conference on Agile Software Development. Springer Nature Switzerland Cham. 2022, pp. 39–47 (cit. on pp. 6, 7).

[FFC23] F. Freitas, A. Ferreira, J. Cunha. "A methodology for refactoring ORM-based monolithic web applications into microservices". In: Journal of Computer Languages 75 (2023), p. 101205 (cit. on p. 45).

[Fow04] M. Fowler. Strangler Fig Application. Accessed: 2025-03-16. 2004. URL: https://martinfowler.com/bliki/StranglerFigApplication.html (cit. on pp. 12, 72).

[Fow13] M. Fowler. D D D Aggregate. Accessed: 2025-03-16. 2013. URL: https://martinfowler.com/bliki/DDD_Aggregate.html (cit. on p. 53).

[Fow25] M. Fowler. Branch By Abstraction. Accessed: 2025-03-16. 2025. URL: https://martinfowler.com/bliki/BranchByAbstraction.html (cit. on p. 12).

[Fri24a] J. Fritzsch. "Architectural refactoring to microservices: a quality-driven methodology for modernizing monolithic applications". In: (2024) (cit. on pp. iii, v, 1, 3, 9–11, 15, 16, 34, 45, 65, 71, 74, 81).

[Fri24b] J. Fritzsch. Architecture Refactoring Helper. Accessed: 2024-07-02. 2024. URL: https://github.com/jfr609/architecture-refactoring-helper (cit. on p. 6).

[GCL+20] M. Gao, M. Chen, A. Liu, W. H. Ip, K. L. Yung. "Optimization of Microservice Composition Based on Artificial Immune Algorithm Considering Fuzziness and User Preference". In: IEEE Access 8 (2020), pp. 26385–26404. DOI: 10.1109/ACCESS.2020.2971379 (cit. on pp. 13, 43).

[GKGZ15a] M. Gysel, L. Kölbener, W. Giersche, O. Zimmermann. Service Cutter: A Structured Way to Service Decomposition. Accessed: 2025-04-04. 2015. URL: https://github.com/ServiceCutter/ServiceCutter (cit. on pp. 16, 50, 51).

[GKGZ15b] M. Gysel, L. Kölbener, W. Giersche, O. Zimmermann. Service Cutter: Aggregates. Accessed: 2025-04-04. 2015. URL: https://github.com/ServiceCutter/ServiceCutter/wiki/Aggregates (cit. on pp. 16, 53).

[GKGZ15c] M. Gysel, L. Kölbener, W. Giersche, O. Zimmermann. Service Cutter: Coupling-Criteria. Accessed: 2025-04-04. 2015. URL: https://github.com/ServiceCutter/ServiceCutter/wiki/Coupling-Criteria (cit. on p. 53).

[GKGZ15d] M. Gysel, L. Kölbener, W. Giersche, O. Zimmermann. Service Cutter: ERM. Accessed: 2025-04-04. 2015. URL: https://github.com/ServiceCutter/ServiceCutter/wiki/ERM (cit. on pp. 16, 51).

[GKGZ15e] M. Gysel, L. Kölbener, W. Giersche, O. Zimmermann. Service Cutter: Use-Cases. Accessed: 2025-04-04. 2015. URL: https://github.com/ServiceCutter/ServiceCutter/wiki/Use-Cases (cit. on pp. 16, 53).

[GKGZ15f] M. Gysel, L. Kölbener, W. Giersche, O. Zimmermann. Service Cutter: User-Representations. Accessed: 2025-04-04. 2015. URL: https://github.com/ServiceCutter/ServiceCutter/wiki/User-Representations (cit. on pp. 51, 53, 74).

[GKGZ16a] M. Gysel, L. Kölbener, W. Giersche, O. Zimmermann. "Service cutter: A systematic approach to service decomposition". In: Service-Oriented and Cloud Computing: 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings 5. Springer. 2016, pp. 185–200 (cit. on pp. iii, v, 1, 11, 16, 42, 45, 47, 50, 51, 55–61, 66, 69, 71–73).

[GKGZ16b] M. Gysel, L. Kölbener, W. Giersche, O. Zimmermann. Service Cutter: Usage Scenarios. Accessed: 2025-02-11. 2016. URL: https://github.com/ServiceCutter/ServiceCutter/wiki/Usage-Scenarios (cit. on p. 42).

[GM20] M. H. Gomes Barbosa, P. H. M. Maia. "Towards Identifying Microservice Candidates from Business Rules Implemented in Stored Procedures". In: 2020 IEEE International Conference on Software Architecture Companion (ICSA-C). 2020, pp. 41–48. DOI: 10.1109/ICSA-C50368.2020.00015 (cit. on p. 43).

[GN02] M. Girvan, M. E. Newman. "Community structure in social and biological networks". In: Proceedings of the national academy of sciences 99.12 (2002), pp. 7821–7826 (cit. on pp. 16, 51, 53, 54, 59).

[Goo25] Google Scholar. Google Scholar. https://scholar.google.com/. Accessed: 2025-04-24. 2025 (cit. on p. 9).

[Her24] A. Herrmann. "Toolunterstütztes Refactoring von Microservices-Architekturen: Eine industrielle Fallstudie". B.S. thesis. 2024 (cit. on pp. iii, v, 1, 9–12, 19, 50).

[IBM25] IBM. IBM - Official Website. Accessed: 2025-03-16. 2025. URL: https://www.ibm.com/ (cit. on p. 12).

[Int11] International Organization for Standardization/International Electrotechnical Commission. ISO/IEC 25010:2011—Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models. Geneva, Switzerland: ISO, 2011 (cit. on pp. 11, 20).

[IT21] N. Ivanov, A. Tasheva. "A hot decomposition procedure: Operational monolith system to microservices". In: 2021 International Conference Automatics and Informatics (ICAI). IEEE. 2021, pp. 182–187 (cit. on pp. 12, 44).

[KBAW94] R. Kazman, L. Bass, G. Abowd, M. Webb. “SAAM: A method for analyzing the properties of software architectures”. In: Proceedings of 16th International Conference on Software Engineering. IEEE. 1994, pp. 81–90 (cit. on p. 19).

[KKB+98] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere. “The architecture tradeoff analysis method”. In: Proceedings. fourth ieee international conference on engineering of complex computer systems (cat. no. 98ex193). IEEE. 1998, pp. 68–78 (cit. on pp. 3, 5, 6, 19).

[KN10] M. Kramer, P. H. Newcomb. “Legacy system modernization of the engineering operational sequencing system (eoss)”. In: Information Systems Transformation. Elsevier, 2010, pp. 249–281 (cit. on p. 4).

[Kno23] M. Knodel. “Migration monolithischer Anwendungen in Microservices-basierte Architekturen: Fallstudie einer Service/Sales-Applikation”. MA thesis. 2023 (cit. on pp. iii, v, 1, 9–12, 19).

[Koc22] D. Koch. “Migrating monolithic architectures to microservices: a study on software quality attributes”. MA thesis. 2022 (cit. on pp. 5, 9–11, 20).

[KXK+21] A. K. Kalia, J. Xiao, R. Krishna, S. Sinha, M. Vukovic, D. Banerjee. “Mono2micro: a practical and effective tool for decomposing monolithic java applications to microservices”. In: Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering. 2021, pp. 1214–1224 (cit. on pp. 11, 12, 44).

[L-m25] L-mobile. L-mobile: Digital Solutions for Manufacturing and Logistics. https://l-mobile.com/. Accessed: 2025-04-24. 2025 (cit. on p. 11).

[LF 24] LF Consult GmbH. LF Consult GmbH. Accessed: 2024-08-04. 2024. URL: https://www.lfconsult.de/de/ (cit. on pp. iii, v, 1, 71, 72).

[LF14] J. Lewis, M. Fowler. “Microservices”. In: (2014). Accessed: 2024-06-27. URL: https://martinfowler.com/articles/microservices.html (cit. on pp. 3, 4).

[LML20] C.-Y. Li, S.-P. Ma, T.-W. Lu. “Microservice migration using strangler fig pattern: A case study on the green button system”. In: 2020 International Computer Symposium (ICS). IEEE. 2020, pp. 519–524 (cit. on p. 11).

[LO20] J. Löhnertz, A. Oprescu. “Steinmetz: Toward Automatic Decomposition of Monolithic Software Into Microservices”. In: Seminar on Advanced Techniques and Tools for Software Evolution. 2020. URL: https://api.semanticscholar.org/CorpusID:229366857 (cit. on pp. 13, 16, 43–45, 47, 48, 50).

[Löh25] L. Löhnertz. Steinmetz: A Tool for ... (add a brief description if needed). GitHub repository. 2025. URL: https://github.com/loehnertz/Steinmetz (cit. on pp. 13, 47, 48).

[LXXW22] J. Li, H. Xu, X. Xu, Z. Wang. “A novel method for identifying microservices by considering quality expectations and deployment constraints”. In: International Journal of Software Engineering and Knowledge Engineering 32.03 (2022), pp. 417–437 (cit. on p. 13).

[LZJ+19] S. Li, H. Zhang, Z. Jia, Z. Li, C. Zhang, J. Li, Q. Gao, J. Ge, Z. Shan. “A dataflow-driven approach to identifying microservices from monolithic applications”. In: Journal of Systems and Software 157 (2019), p. 110380 (cit. on pp. 43, 66).

[Mal24] P.-J. Malandrino. “Architecture Style: Modulith (vs Microservices)”. In: Level Up Coding (Jan. 2024). URL: https://levelup.gitconnected.com/architecture-style-modulith-vs-microservices-90c7c75713db (cit. on p. 4).

[MCF+20] T. Matias, F. F. Correia, J. Fritzsch, J. Bogner, H. S. Ferreira, A. Restivo. “Determining microservice boundaries: A case study using static and dynamic software analysis”. In: Software Architecture: 14th European Conference, ECSA 2020, L’Aquila, Italy, September 14–18, 2020, Proceedings 14. Springer. 2020, pp. 315–332 (cit. on p. 44).

[Men20] N. Menard. “Decision criteria between microservice and monolithic architecture”. B.S. thesis. N. Menard, 2020 (cit. on p. 3).

[NARK22] V. Nitin, S. Asthana, B. Ray, R. Krishna. “Cargo: Ai-guided dependency analysis for migrating monolithic applications to microservices architecture”. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. 2022, pp. 1–12 (cit. on pp. 11, 13).

[New21] S. Newman. Building microservices. Ö’Reilly Media, Inc.", 2021 (cit. on p. 4).

[NG04] M. E. Newman, M. Girvan. “Finding and evaluating community structure in networks”. In: Physical review E 69.2 (2004), p. 026113 (cit. on pp. 16, 51, 53, 54, 59).

[NUEH18] R. Nakazawa, T. Ueda, M. Enoki, H. Horii. “Visualization Tool for Designing Microservices with the Monolith-First Approach”. In: 2018 IEEE Working Conference on Software Visualization (VISSOFT). 2018, pp. 32–42. DOI: 10.1109/VISSOFT.2018.00012 (cit. on p. 42).

[Ope24] OpenAPI Initiative. OpenAPI Specification. Accessed: 2025-02-12. 2024. URL: https://swagger.io/specification/ (cit. on p. 42).

[PKY20] J. Park, D. Kim, K. Yeom. “An Approach for Reconstructing Applications to Develop Container-Based Microservices”. In: Mobile Information Systems 2020.1 (2020), p. 4295937 (cit. on p. 13).

[RAK07] U. N. Raghavan, R. Albert, S. Kumara. “Near linear time algorithm to detect community structures in large-scale networks”. In: Physical Review E—Statistical, Nonlinear, and Soft Matter Physics 76.3 (2007), p. 036106 (cit. on pp. 16, 51, 53, 54).

[RH09] P. Runeson, M. Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: Empir. Softw. Eng. 14.2 (2009), pp. 131–164 (cit. on pp. 67, 68).

[Rob03] C. M. Robert. “Agile software development”. In: Martin: Prentice Hall (2003) (cit. on p. 13).

[SBH+24] X. Sun, S. Boranbaev, S. Han, H. Wang, D. Yu. “Expert system for automatic microservices identification using API similarity graph”. In: Expert Systems 41.5 (2024), e13158 (cit. on p. 13).

[Sil22] K. Silz. “Spring Modulith Structures Spring Boot 3 Applications with Modules and Events”. In: InfoQ (Nov. 2022). URL: https://www.infoq.com/news/2022/11/spring-modulith-launch/ (cit. on pp. 4, 72).

[SL24] R. Su, X. Li. "Modular Monolith: Is This the Trend in Software Architecture?" In: arXiv preprint arXiv:2401.11867 (Jan. 2024). DOI: 10.48550/arXiv.2401.11867. URL: https://arxiv.org/abs/2401.11867 (cit. on p. 4).

[SM07] M. Svahnberg, F. Mårtensson. "Six years of evaluating software architectures in student projects". In: Journal of Systems and Software 80.11 (2007), pp. 1893–1901 (cit. on pp. 1, 11, 15, 19, 20).

[SOMS19] I. Saidani, A. Ouni, M. W. Mkaouer, A. Saied. "Towards automated microservices extraction using muti-objective evolutionary search". In: Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings 17. Springer. 2019, pp. 58–63 (cit. on p. 13).

[Son25] Sonatype. Maven Central Repository. https://repo1.maven.org/maven2/. Accessed: 2025-04-01. 2025 (cit. on p. 48).

[SP21] A. Santos, H. Paula. "Microservice decomposition and evaluation using dependency graph and silhouette coefficient". In: Proceedings of the 15th Brazilian Symposium on Software Components, Architectures, and Reuse. 2021, pp. 51–60 (cit. on p. 44).

[SSR19] N. Singhal, U. Sakthivel, P. Raj. "Efficient Hybrid Research for QoS-Aware Mcroservice Compostion". In: International Journal of Recent Technology and Engineering 8.2 (2019), pp. 5251–5255 (cit. on p. 43).

[Tek04] B. Tekinerdogan. "ASAAM: Aspectual software architecture analysis method". In: Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004). IEEE. 2004, pp. 5–14 (cit. on pp. 3, 5, 6, 19).

[Van08] S. Van Dongen. "Graph clustering via a discrete uncoupling process". In: SIAM Journal on Matrix Analysis and Applications 30.1 (2008), pp. 121–141 (cit. on pp. 16, 51, 53, 58).

[VGC+15] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, S. Gil. "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud". In: 2015 10th computing colombian conference (10ccc). IEEE. 2015, pp. 583–590 (cit. on pp. iii, v, 1, 3).

[Viv24] I. M. Viveros. microservicesbacklog. https://github.com/ivanmviveros/microservicesbacklog. Accessed: 2025-04-04. 2024 (cit. on p. 50).

[VPAG21] F. H. Vera-Rivera, E. Puerto, H. Astudillo, C. M. Gaona. "Microservices Backlog–A Genetic Programming Technique for Identification and Evaluation of Microservices From User Stories". In: IEEE Access 9 (2021), pp. 117178–117203. DOI: 10.1109/ACCESS.2021.3106342 (cit. on pp. 11, 16, 44, 45, 47–50).

[Zen23] Zenodo Initiative. Zenodo Record 15280694. Accessed: 2025-04-27. 2023. DOI: 10.5281/zenodo.15291397. URL: https://doi.org/10.5281/zenodo.15291397 (cit. on pp. 9, 10, 20, 21, 34, 52, 53, 55–61, 81).

[ZSS+21] P. Zaragoza, A.-D. Seriai, A. Seriai, H.-L. Bouziane, A. Shatnawi, M. Derras. "Refactoring Monolithic Object-Oriented Source Code to Materialize Microservice-Oriented Architecture." In: ICSOFT 117 (2021), p. 126 (cit. on p. 12).

All links were last followed on April 28, 2025.

.1 Architecture Refactoring Helper Repository Data

To recreate our findings regarding the querying of the Architecture Refactoring Helper [Fri24a], in its second phase, we provide the Service Identification Approaches Repository file under [Zen23].

.2 Quality Attribute Lists

In this section, we provide the quality attribute lists handed out at the start of the architecture review. As the review was held in German, we gave the participants a German version, which is available under [Zen23]. For reference, we also provide an English version in Table 1.

Upper Attribute	Lower Attribute	Description
Reliability		The degree to which a system, product, or component performs specified functions under specified conditions for a specified period of time. This attribute is about how failures are handled.
	Availability	This describes that if a system or component is required, the degree to which it can be used, or is accessible.
	Fault Tolerance	The degree to which a system, product, or component operates as intended despite the presence of hardware or software faults.
	Recoverability	The degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system.
Compatibility		The extent to which a system or component is able to interact with other systems in a homogeneous environment.
	Interoperability	The extent to which a system or component is able to interact with other systems in heterogeneous environments.
	Co-Existence	The degree to which a product functions in a shared environment without having an unwanted influence on other parts of the system.
Security		The extent to which the data and other information of a system are protected, which therefore also includes which systems and persons have access to this information.

Upper Attribute	Lower Attribute	Description
	Confidentiality	Describes how and to what extent is it ensured that only authorized persons or systems have access to the information of a system.
	Accountability	The degree to which actions performed by an entity in or with a system can be traced.
	Authenticity	Whether and to what extent a system is able to recognize and confirm the actual identity of an entity.
	Integrity	The degree to which a system, product, or component prevents unauthorized access to, or modification of, computer programs or data.
Performance		The degree of capability of a system under certain conditions in relation to the resources used.
	Time Behavior	The degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements.
	Resource Utilization	The degree to which the resources of a system are used in the execution of its functions.
Scalability		The ability of a system to add or remove resources so that they can be used effectively.
Portability		How easily the platform or environment of a software or system can be changed.
	Installability	The degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software, or other operational or usage environments.
	Adaptability	The extent to which a system can adapt to changing or evolving environments or platforms.
	Replaceability	The degree to which another system or software in the same environment can be replaced by another system or software to ensure the same functionality.
	Deployability	The way a software product or system is delivered to the host or target platform, how it is integrated, and how subsequent possible updates are handled.

Upper Attribute	Lower Attribute	Description
	Agility	The degree to what extent it is possible to carry out simple adjustments.
Independence		The fundamental separation of the individual services from each other, so that they only communicate with one another via interfaces.
Maintainability		The ability of a system or software product to be adapted, modified, or corrected.
	Modifiability	The degree of how simple it is to change something within the system.
	Modularity	The degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
	Reusability	The degree to which an asset can be used in more than one system, or in building other assets.
	Testability	The degree of effectiveness and efficiency with which test criteria can be established for a system.
	Analysability	The degree of effectiveness and efficiency with which it is possible to assess the impact on a system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
	Monitorability	The degree to what extent the system or software product can be controlled or monitored during execution.
	Manageability	The degree of centralization.
	Understandability	The degree to which a system or organizational factors of a system can be understood in relation to the context of the system.
Business		This includes attributes that do not describe the quality of the microservice architecture itself, but those that address external factors in the immediate context.
	Execution Cost	The cost associated with the invocation of this service.

Upper Attribute	Lower Attribute	Description
	Organizational Alignment	The extent to which the structure of the teams and the communication between them can be mapped to the structure and dependencies within the microservice architecture.

Table 1: Attribute Description

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature