Automated Microservice Identification in Modular Monolith Architectures

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Software Engineering
April 2024
Florian Dejonckheere

UNIVERSITY OF TURKU
Department of Computing

Florian Dejonckheere: Automated Microservice Identification in Modular Monolith Architectures

Master of Science (Tech) Thesis, 77 p., 4 app. p.
Software Engineering
April 2024

The modular monolith architecture has recently emerged as a harmonization of monolithic and microservices architectures, offering a balanced approach to modularity, scalability, and flexibility for software design. As simple software systems evolve into complex and hard-to-maintain monolithic applications, many actors are pivoting towards modular monoliths or full microservices architectures to meet the ever-increasing demands of modern business.

This thesis investigates the merits of adopting a modular monolith architecture for monolithic applications, and the challenges faced during the migration process. Based on the findings, an automated approach is formulated to aid the modularization process, using dependency analysis and machine learning algorithms to identify module boundaries. The proposed solution uses a four-step approach to monolith decomposition: extraction, decomposition, visualization, and evaluation. It is then implemented for a case study, and evaluated using a set of metrics to assess the effectiveness of the proposed decomposition. The results indicate that the automated approach is effective in identifying module boundaries, and can be used to aid the modularization process of monolithic applications. Based on the results of the evaluation, a number of optimizations are suggested to improve the effectiveness of the automated approach. The study concludes that using automated technologies to reduce the manual effort required for modularization can significantly improve the efficiency and accuracy of the process.

Keywords: software architecture, monolith, microservices, modular monolith, modularization

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

AST Abstract Syntax Tree

BPMN Business Process Model and Notation

DSRM Design Science Research Methodology

DSRP Design Science Research Process

SDLC Software Development Life Cycle

SLOC Source Lines of Code

SLR Systematic Literature Review

1 Introduction

In the past decade, software engineering has seen a radical shift in the way software is developed and deployed. The rise in popularity of cloud computing and containerization has led to the emergence of microservices a new software architecture paradigm. Microservices as an architectural style emphasize the development of small, distributed services that are deployed independently and communicate with each other over an internal network. This approach has several benefits, including scalability and fault tolerance. Many big and small organizations have adopted a microservices architecture to increase the flexibility of their software systems, and to enable faster development and deployment cycles.

Migrating monolithic applications to a microservices architecture is not a trivial task. It involves deep understanding of software engineering principles, the existing codebase, and the business domain. Moreover, as larger and older applications typically have more technological debt, the process of migrating to microservices can be overly complex and error-prone.

In recent years, a new software architecture paradigm has emerged that takes a hybrid approach to monolithic and microservices architectures. The modular monolith architecture aims to combine the advantages of using a microservices architecture with the simplicity of a monolithic codebase. The modular monolithic application consists of multiple independent modules encapsulating a specific set of functionality. The modules are developed

in tandem, but deployed as single units. This approach allows developers to rapidly build and deploy new features, while maintaining the flexibility and scalability of a microservices architecture. As the code resides in the same codebase, developers can easily restructure and redefine the module boundaries. This makes the modular monolith architecture an attractive option for organizations that want to migrate their monolithic software systems to a more flexible and scalable architecture.

In this thesis, we aim to investigate the potential benefits of a modular monolithic architecture, and how automated technologies can help software architects to migrate their monolithic codebases to a modular monolith architecture.

1.1 Motivation

The case study conducted in this thesis relates to a software system developed at Nipro Digital Technologies Europe, a Belgian software company that specializes in developing software solutions for the healthcare industry. The application in question is a monolithic application that has accumulated a significant amount of technical debt over the years, and might benefit from a modular monolith architecture. It is the aim of this thesis to investigate the viability of applying automated modularization techniques to this software system, in order to pivot towards a modular monolith architecture.

1.2 Scope and goal

This research is centered around three research questions:

Research Question 1: What are the challenges and opportunities of the modular monolith architecture compared to traditional monolithic and microservices architectures?

Research Question 2: What are the existing approaches and tools for automated microservice candidate identification in monolith codebases?

Research Question 3: How can static analysis of source code identify module boundaries in a modular monolith architecture that maximize internal cohesion and minimize external coupling?

The motivation behind the first research question is to investigate the potential benefits and drawbacks of the modular monolith architecture with a particular focus on its application to existing monolithic codebases. To answer this question, we will first define the modular monolith architecture using existing literature, and examine what sets it apart from monolithic and microservices architectures. Then, we will proceed to investigate the merits and drawbacks of the software architecture when applied to an existing codebase.

The second research question is motivated by the need for reducing the complexity and error-proneness of manual modularization efforts. We will explore the existing automated technologies to aid modularization of monolithic codebases in the literature. This will then help us to answer the third research question. An approach to automated modularization will be chosen based on the review of existing technologies. A prototype of the proposed solution will be implemented, and applied to a case study. The results will be evaluated using a set of quality metrics, and its effectiveness will be discussed.

The goal of this research can be summarized as follows:

1. Investigate the merits and drawbacks of the modular monolith architecture
2. Investigate the use of automated technologies to modularize a monolithic architecture

Although the proposed solution will be designed for a specific case study, the results can be generalized to other monolithic codebases.

1.3 Outline

The thesis is divided into three parts.

The first part comprises the background and related work. In Chapter 1, the scope and goal of the research is defined, and the research questions are formulated. Chapter 2 describes the research methodology used in this thesis. Then, Chapter 3 introduces the reader to the research background and necessary concepts. In Chapter 4, the existing literature on the modular monolith architecture and automated modularization is discussed.

The second part of the thesis, starting with Chapter 5, is dedicated to the first research question. The modular monolith architecture is defined, and its merits and drawbacks are discussed.

The third part aims to solve the second and third research questions. Chapter 6 takes a deep dive into the existing technologies for automated modularization. Chapter 7 then continues with a proposed solution for automated modularization. In Chapter 8, a case study is presented along with a strategy for applying the proposed solution. The results are then evaluated and discussed.

Finally, Chapter 9 summarizes the findings, and gives an outlook on future work.

2 Methodology

This chapter describes the methodology used in this thesis in detail.

To answer the first research question, an ad hoc review is conducted, picking a select number of papers that define and discuss the modular monolith architecture. An ad hoc review is a less formal review process, where the researcher discusses purposefully selected papers to gain an understanding of a specific topic [1].

For the second research question, a systematic literature review is conducted to identify and summarize the state of the art in automated modularization technologies. Systematic literature reviews are more formal than ad hoc reviews, and follow a well-defined process to reduce bias and increase the reliability of the results.

The third research question is answered by designing an approach based on the results of the systematic literature review, and implementing it for a case study. The effectiveness of the approach is then evaluated based on quantitative and qualitative metrics.

Systematic literature review

A systematic literature review is used to identify, evaluate and interpret research literature for a given topic area, or research question [2]. The systematic nature of systematic literature reviews reduces sampling bias through a well-defined sequence of steps to identify and categorize

existing literature, and applies techniques such as forward and reverse snowballing to reduce publication bias [1]. Studies directly researching the topic area are called *primary* studies, systematic studies aggregating and summarizing primary studies are called *secondary* studies. *Tertiary* studies are systematic studies aggregating and summarizing secondary studies. Systematic literature reviews often only consider primary studies as they are considered the most reliable source of information, but may also include secondary studies if the primary studies are scarce, or as a means to identify primary studies.

The systematic literature review was conducted using the three-step protocol as defined by B. Kitchenham and S. Charters:

	Step	**Activity**
1	Plan	Identify the need for the review, specifying the research questions, and developing a review protocol
2	Conduct	Identification and selection of literature, data extraction and synthesis
3	Report	Evaluation and reporting of the results

Table 1: Systematic literature review process

Case study

For the case study, a Design Science Research Methodology (DSRM) is adopted, which is a research paradigm for information systems research focused at creating and evaluating artifacts. In particular, the research and design of the proposed solution follows the six-step Design Science Research Process (DSRP) model [3]. The model is inspired by prior research and is

designed to guide researchers through the process of analysis, creation, and evaluation of artifacts in information science.

The six steps of the process are:

1. **Problem identification and motivation**: Research problem statement and justification for existence of a solution.
2. **Objectives of a solution**: Definition of the objectives, derived from the problem statement.
3. **Design and development**: Creation of the artifact.
4. **Demonstration**: Usage of the artifact to demonstrate its effectiveness in solving the problem.
5. **Evaluation**: Observation and measurement of how well the artifact supports a solution to the problem.
6. **Communication**: Transfer of knowledge about the artifact and the problem solution to the relevant audience.

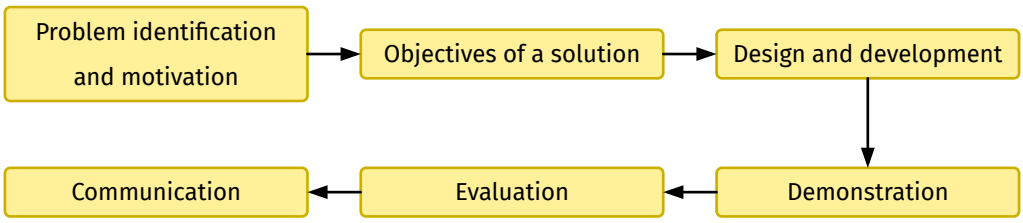


Figure 1: Design Science Research Process (DSRP)

The process is structured sequentially, however the authors suggests that researchers may proceed in a non-linear fashion, and start or stop at any step, depending on the context and requirements of the research.

In this thesis, we use the DSRP as a guideline for the design, development, and evaluation of the automated modularization approach used in the case study. We focus in particular on the design and development, demonstration, and evaluation steps.

3 Background

In this chapter background information and technical concepts related to the topic of the thesis are discussed. We start with a brief introduction to monolith software architecture, and continue with service-oriented architecture and microservices.

Monolith architecture

Modular programming

Microservice architecture

Modularization

4 Related work

5 Modular monolith architecture

5.1 Background

5.2 Challenges and opportunities

5.3 Modularization

6 Automated modularization

In this chapter, we investigate the state of the art in automated technologies for modularization of monolith codebases. Using a systematic literature review, we identified and categorized existing literature on automated modularization of monolith codebases.

6.1 Plan

Using the systematic literature review, we answered the following research question:

Research Question 2: What are the existing approaches and tools for automated microservice candidate identification in monolith codebases? The motivation for the research question is discussed in Chapter 1.

In current literature, several systematic mapping studies related to microservices architecture have been conducted [4], [5], as well as systematic literature reviews related to microservice decomposition . However, the methods discussed in these studies are mostly aimed at aiding the software architect in identifying microservice candidates, rather than providing automated solutions. Therefore, we believe that there is a need for a systematic literature review aimed at summarizing existing literature regarding automated and semi-automated methods for modularization of monolith codebases.

Automated methods for modularization are techniques that autonomously perform the entire decomposition process, without requiring intervention of a software architect. The resulting architecture is then presented to the software architect for validation and implementation. Semi-automated methods for modularization are techniques that assist the software architect in the decomposition process, but do not perform the entire process autonomously. The software architect is required to make decisions during the process, and is left with several final proposals to choose from. Automated methods are of particular interest, as they take away the manual effort required from the software architect to analyze and decompose the monolith codebase.

As a search strategy, the following platforms were queried for relevant publications:

1. IEEE Xplore[1]
2. ACM Digital Library[2]

The platforms were selected based on their academic relevance, as they contain a large number of publications in the field of software engineering. Furthermore, the platforms also contain only peer-reviewed publications, which ensures a certain level of quality in the publications.

Based on a list of relevant topics, we used a combination of related keywords to formulate the search query. We refrained from using more generic keywords, such as "architecture" or "design", as they would yield too many irrelevant results. The topics relevant for the search query are:

- **Architecture**: the architectural styles being discussed in the publications.

  Keywords: *microservice, monolith, modular monolith*

[1]https://ieeexplore.ieee.org/
[2]https://dl.acm.org/

- **Modularization**: the process of identifying and decomposing modules in a monolith architecture.

  Keywords: *service identification, microservice decomposition, monolith modularization*
- **Technology**: the technologies, algorithms, or methods for modularization.

  Keywords: *automated tool, machine learning, static analysis, dynamic analysis, hybrid analysis*

The resulting search query can be expressed as follows:

```
  (('microservice*' IN title OR abstract) OR
   ('monolith*' IN title OR abstract))
AND
  (('decompos*' IN title OR abstract) OR
   ('identificat*' IN title OR abstract))
AND
  ('automate*' IN title OR abstract)
```

Listing 1: Search query

The search query was adapted to the specific search syntax of the platform.

In addition to search queries on the selected platforms, we used snowballing to identify additional relevant publications. Snowballing is a research technique used to find additional publications of interest by following the references of the selected publications .

Based the inclusion/exclusion criteria in Table 2, the results were filtered, and the relevant studies were selected for inclusion in the systematic literature review.

	Criteria
Inclusion	• Title, abstract or keywords include the search terms • Conference papers, research articles, blog posts, or other publications • Publications addressing (semi-)automated methods or technologies
Exclusion	• Publications in languages other than English • Publications not available in full text • Publications using the term "microservice", but not referring to the architectural style • Publications aimed at greenfield[3] or brownfield[4] development of systems using microservices architecture • Publications published before 2014, as the definition of "microservices" as an architectural style is inconsistent before 2014 [5] • Publications addressing manual methods or technologies • Surveys, opinion pieces, or other non-technical publications

Table 2: Inclusion and exclusion criteria

As a final step, the publications were subjected to a validation scan to ensure relevance and quality. To assess the quality, we mainly focused on the technical soundness of the method or approach described in the publication.

[3]Development of new software systems lacking constraints imposed by prior work [6]
[4]Development of new software systems in the presence of legacy software systems [6]

The quality of the publication was assessed based on the following criteria:

- The publication is peer-reviewed or published in a respectable journal
- The publication thoroughly describes the technical aspects of the method or approach
- The publication includes a validation phase or case study demonstrating the effectiveness of the method or approach

This step is necessary to ensure that the selected publications are relevant to the research question and that the results are not biased by low-quality publications.

Once a final selection of publications was made, the resulting publications were qualitatively reviewed and categorized based on the method or approach described.

6.2 Conduct

Using the search strategy outlined in the previous section, we queried the selected platforms and found a total of 507 publications.

Platform	**Search results**	**Selected publications**
IEEE Xplore	339	33
ACM Digital Library	168	9
Snowballing		6
Total	507	48

Table 3: Summary of search results

After applying the inclusion/exclusion criteria, we selected 42 publications for inclusion in the systematic literature review. Of these publications, 36 are primary studies, and 6 are secondary studies. The secondary studies were used as a starting point for the snowballing process, which resulted in 6 additional publications being included in the systematic literature review. For a list of the selected publications, see Appendix A.

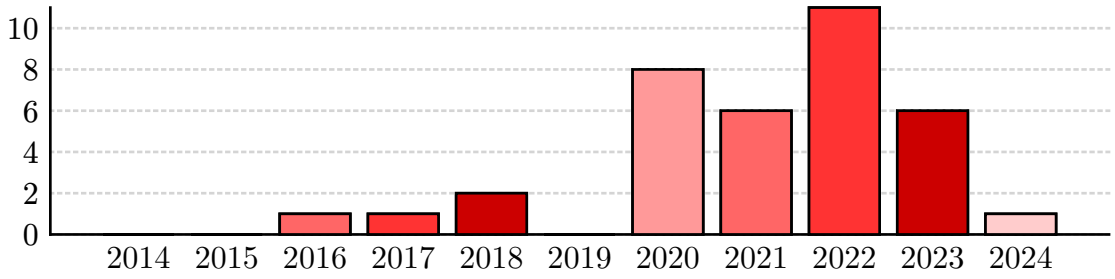


Figure 2: Distribution of selected publications by year

The selected publications range in publication date from 2014 to 2024, with a peak in 2022. Few publications were selected in the first part of the interval, picking up in the later years with a steady increase in the number of publications.

From the selected publications, we extracted relevant information, such as:

- The type of approach or technique described (automated, semi-automated)
- The input data used for the microservice candidate identification process
- The algorithms used in the microservices candidate identification process
- The quality metrics used in the evaluation of the decomposition

B. Kitchenham and S. Charters suggest that the data extraction process should be performed by at least two researchers to ensure the quality and consistency of the extracted data. However, due to resource constraints, the data extraction was performed by a single researcher. To prevent bias and ensure the quality of the data extraction, the results were validated by a re-test procedure where the researcher performs a second extraction from a random selection of the publications to check the consistency of the extracted data.

6.3 Report

The publications selected for inclusion in the systematic literature review were qualitatively reviewed and categorized in three dimensions. The categorization was only performed on the primary studies, as the secondary studies already aggregate and categorize primary studies. The secondary studies were used to perform the snowballing process, which resulted in additional primary studies being included in the systematic literature review.

First, we categorized the publications based on the Software Development Life Cycle (SDLC) artifact used as input for the microservice candidate identification algorithm. Each artifact category has an associated collection type: either static, dynamic, or hybrid. [P41]. Static collection describes a SDLC artifact that was collected without executing the software (e.g. source code or binary code), while dynamic collection describes a SDLC artifact that was collected after or during execution of the software (e.g. execution logs). Some publications describe methods or algorithms that use a combination of SDLC artifacts, which is categorized as hybrid.

Second, we categorized the publications based on the class of algorithm(s) used for microservice candidate identification. We based the classification of the algorithms on M. Abdellatif et al., who identified six types of service identification algorithms.

Third, the publications were also categorized by the quality metrics used for evaluation the proposed decompositions.

6.3.1 SDLC artifact

The identified SDLC artifact categories used as input for the microservice candidate identification algorithm are described in Table 4. The categories are based on Bajaj et al. [P41].

Artifact	**Type**	**Publications**
Requirements documents and models	Static	[P19][P25][P2][P1][P22]
Design documents	Static	[P8][P4][P6][P30][P32]
Codebase	Static	[P27][P9][P7][P10][P20] [P28][P16][P11][P18][P3] [P12][P13][P14][P26][P17] [P21][P29][P31][P32][P33] [P40][P36][P34][P35]
Execution data	Dynamic	[P20][P12][P17][P21][P23] [P5][P15][P24][P34]

Table 4: SDLC artifact categories

Of the four categories, requirements documents and models, design documents, and codebase are static artifacts, while execution data is dynamic. Hybrid approaches using both static and dynamic analysis are categorized according to the artifact used in the static and dynamic analysis. In the selected 43 publications, the majority of the approaches use the codebase as input for the algorithm (24; 55.8%), followed by execution data (9; 20.9%), and design and requirements documents (5; 11.6% each).

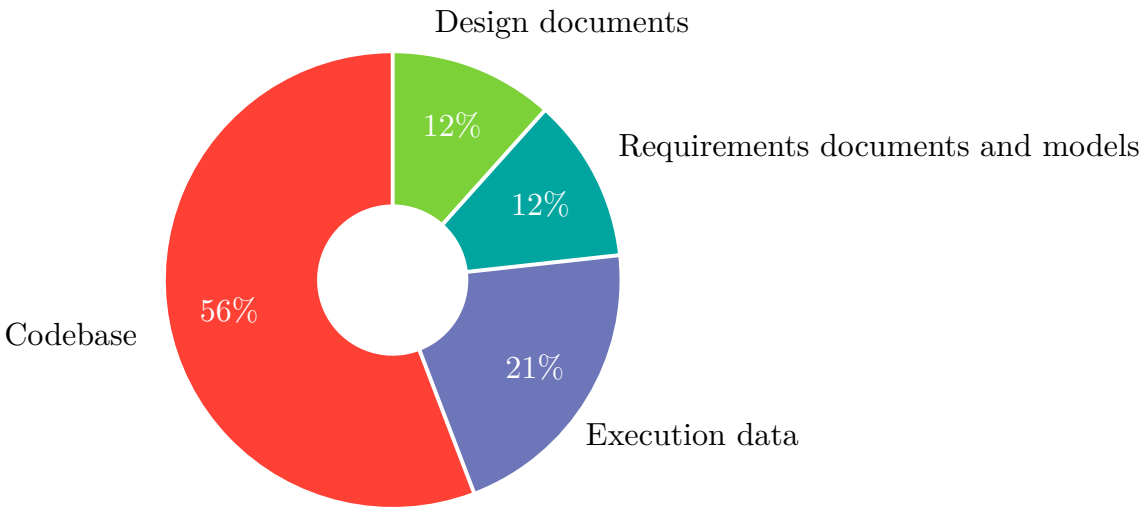


Figure 3: SDLC artifact categories

Requirements documents and models

In software engineering, requirements documents and models are used to formally describe the requirements of a software system following the specification of the business or stakeholder requirements [7]. They include functional and non-functional requirements, use cases, user stories, and business process models. Approaches using requirements documents and models as input for the microservice candidate identification algorithm often times need to pre-process the documents to extract the relevant information, as they are not intended to be directly read by a machine. In many cases, requirements documents and models for legacy systems are no longer available or outdated, which makes this approach less suitable for automated microservice identification.

Amiri [P19] and Daoud et al. [P25] model a software system as a set of business process using the industry standard Business Process Model and Notation (BPMN), using the machine-readable XML representation as input for the algorithm. Yang et al. [P2] tackle requirements engineering using problem frames [8]. Problem frames are a requirements engineering method,

which emphasizes the integration of real-world elements into the software system [P2].

Some approaches use schematic requirements documents in XML format as input for the algorithm, as described by Saidi et al. [P1]. The latter use domain-driven design techniques to extract functional dependencies from the software design as starting point in microservice identification. Li et al. [P22] employ an intermediate format containing a precise definition of business functionality, generated from validated requirements documents.

Design documents

Design documents created by software architects are machine-readable representations of the software system. They describe the software functionalities in detail and are used to guide the implementation of the software system. Design documents include API specifications, UML diagrams (such as class diagrams and sequence diagrams), and entity-relationship diagrams.

Techniques using design documents either use a domain-driven approach, or a data-driven approach. Domain-driven approaches use domain-specific knowledge to identify microservice candidates, while data-driven approaches use knowledge about data storage and data flow to identify microservice candidates. Similar to requirements documents and models, design documents for legacy systems are often not available or outdated, although some design documents can be reconstructed from the software system (e.g., reverse engineering entity-relationship diagrams from the database schema).

For example, Al-Debagy and Martinek [P8] propose a data-driven method based on the analysis of the software system's external API, specified

in the OpenAPI[5] format. The method extracts the information from the specification and converts it into vector representation for further processing.

Zhou and Xiong [P4] use readily available design documents as well, in the form of UML class diagrams, use cases, and object sequence diagrams as starting point for the microservice identification algorithm. The decomposition tool proposed by Hasan et al. [P32] uses design documents as well, although the specifications are inferred from the source code of the software system, and do not require pre-existing design documents.

Quattrocchi et al. [P6] takes a different approach to the problem, using a data-driven approach combined with a domain-driven approach. Software architects describe the software system using a custom architecture description language, and the tool developed by the authors is able to identify microservice candidates. The tool can be prompted to generate different, more efficient decompositions when given additional domain-driven requirements. Wei et al. [P30] uses a similar approach, gathering a list of features from the software architect, and proposing a microservice decomposition based on pre-trained feature tables.

Codebase

A third category of SDLC artifacts is the codebase of the software system. This can be the source code of the software system, or a binary distribution (e.g. a JAR file). For example, the implementation in [P18] accepts either source code or compiled binary code for analysis.

As the source code of the software system is the most detailed representation of how the software system works, it is most often used as input for the microservice candidate identification algorithm. The source code can be analyzed using static analysis (i.e., without executing the software system),

[5]https://www.openapis.org/

dynamic analysis (i.e., during the execution of the software system or test suite), or a combination of both. Dynamic analysis has the advantage that it can be used if the source code is not available.

Additionally, the revision history of the source code can also be used as source for valuable information about the behaviour of the software system. Mazlami et al. [P9] originally proposed the use of the revision history of the source code to identify couplings between classes. The authors suggest multiple strategies that can be used to extract information from the revision history. Others have built upon this approach, using the revision history to identify the authors of the source code, and use this information to drive the identification algorithm [P17] [P35]

Escobar et al. [P27] use the source code of the software system to construct an Abstract Syntax Tree (AST), and map the dependencies between the business and data layer. Kamimura et al. [P7] use a more data-driven approach, and statically trace data access calls in the source code.

Many publications [P10] [P18] [P12] [P13] [P14] [P36] [P34] [P35] construct a dependency graph from Java source code, and use the graph as input for a clustering algorithm. Bandara and Perera [P28] map object-oriented classes in the source code to specific microservices, but require a list of microservices to be specified before the decomposition is performed.

Filippone et al. [P16] concentrate on the API controllers as entrypoints into the software system. A later paper by the same authors [P11] builds on top of this approach by using the API endpoints as entrypoints, and then ascending into the source code by separating the presentation and logic layer. Likewise, Zaragoza et al. [P13] make a distinction between presentation, business, and data layer.

Most of the publications tracing dependencies between classes (or modules) do this at the level of the classes (or modules). As Mazlami et al. [P9] remark, using a more granular approach at the level of methods (or functions) and attributes has the potential to improve the quality of the decomposition. Carvalho et al. [P20] use a more granular approach, identifying dependencies between methods in the source code. On the other hand, Kinoshita and Kanuka [P3] do not automatically extract information from the source code, but rely on a software architect to decompose the software system on the basis of business capability.

Romani et al. [P26] propose a data-centric microservice candidate identification method based on knowledge gathered from the database schema. The authors extract table and column methods from the database schema, and use the semantically enriched information as input for the identification algorithm. Hao et al. [P21] construct access patterns from both the database schema (static) and the database calls during execution of the software system (dynamic).

A unique approach to constructing a call graph is proposed by Nitin et al. [P33], who make a distinction between context-insensitive and context-sensitive dependency graphs. While the former captures the dependencies between classes using simple method calls, the latter also includes the context (i.e., the arguments) of the method call in the dependency graph.

Execution

As the last category, information about the behaviour of the system can also be collected during the runtime of the software system. Execution data includes log files, execution traces, and performance metrics. This category is often combined with static analysis on source code, as the execution data can provide additional information to the identification algorithm. In dynamic

languages such as Java, dynamic analysis can trace access patterns that static analysis cannot (e.g., due to late binding and polymorphism). Additionally, execution data can be collected when the source code of the software system is not available.

Examples of approaches using execution traces are Jin et al. [P23] and Eyitemi and Reiff-Marganiec [P24]. Using software probes inserted into the bytecode of respectively Java and .NET applications, the authors are able to monitor execution paths. Zhang et al. [P5] collect the execution traces of the software system, in combination with performance logs.

Ma et al. [P15] use a data-centric approach based on the analysis of database access requests.

Hybrid approach

Some publications suggest a hybrid approach using both static and dynamic analysis. For instance, Wu and Zhang [P12], Carvalho et al. [P20] and Cao and Zhang [P34] collect information statically from the source code (entity classes and databases), as well as dynamically from the execution of the software system (execution traces). The approach proposed by Lourenço and Silva [P17] uses either static of the source code or dynamic analysis of the system execution to gather access patterns.

Hao et al. [P21] use both static and dynamic analysis, albeit aimed at the database schema and database calls, respectively.

6.3.2 Algorithms

Microservice candidate identification is a problem that is often solved by representing the architecture as a directed graph. The graph exposes the relationship between the elements of the software architectures. The nodes of the graph represent the classes, modules, or components, and the edges the function or method calls between them. Often the edges are weighted, representing the frequency or cost of the calls. Based on the information contained within, the graph is then divided into several clusters, each encapsulating a microservice candidate. The goal is to find a partitioning of the graph that minimizes the number of edges between clusters and maximizes the number of edges within clusters.

The identified classes of microservice candidate identification algorithms are described in Table 5.

Type	**Example algorithms**	**Publications**
Clustering algorithms	K-Means, DBSCAN, Hierarchical Agglomerative Clustering, Affinity Propagation	[P25][P1][P8][P10] [P28][P12][P13][P14] [P26][P17][P21][P15] [P31][P33][P40][P36] [P34]
Evolutionary algorithms	NSGA-II, NSGA-III	[P19][P22][P4][P20] [P3][P23][P5]
Graph algorithms	Kruskal, Louvain method, Leiden algorithm, Label Propagation	[P2][P9][P11][P29] [P32][P33][P34]
Other algorithms	Linear optimization, custom algorithms	[P6][P27][P7][P10] [P16][P18][P24][P30] [P32][P35]

Table 5: Microservice candidate identification algorithm

We categorized 41 algorithms in the literature into three main classes: clustering algorithms, evolutionary algorithms, and graph algorithms. Publications proposing a custom algorithm that does not fit into one of these categories are grouped in a single category. The majority of the algorithms identified in the literature are clustering algorithms (17; 41.5%), followed by evolutionary algorithms (7; 17.1%) and graph algorithms (7; 17.1%). The remaining algorithms are grouped in the "Other algorithms" category (10; 24.4%).

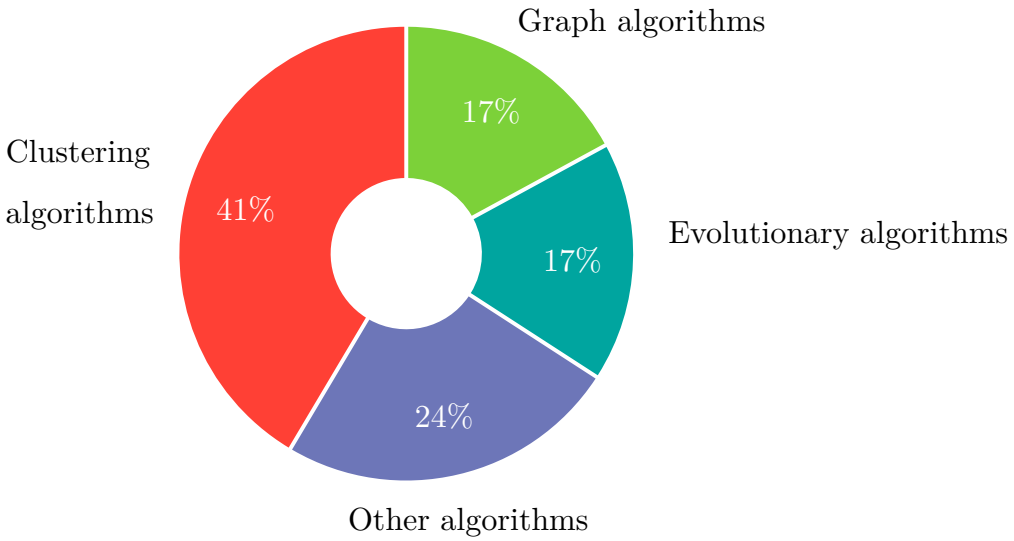


Figure 4: SDLC algorithm categories

Clustering algorithms

The first class of algorithms identified in the literature is clustering algorithms. Clustering algorithms are unsupervised machine learning algorithms that aim to find an optimal partitioning of the graph. Typical clustering algorithms used for this purpose are K-Means clustering and agglomerative clustering.

Examples of publications using K-Means clustering to identify microservice candidates are Saidi et al. [P1], Wu and Zhang [P12], Romani et al. [P26], and Hao et al. [P21].

Al-Debagy and Martinek [P8] use Affinity Propagation [9] to cluster vector representations of operation names in a software system. Affinity Propagation is a clustering algorithm that identifies exemplars in the data, which are used to represent the clusters.

Hierarchical clustering approaches are used in various publications [P10] [P14] [P17] [P15] [P13] [P28]. Lourenço and Silva [P17] uses similarity between domain entities accesses and development history of source code

files as a guiding measure for the clustering algorithm, while Zaragoza et al. [P13] uses structural and data cohesion of microservices. Daoud et al. [P25] extend the hierarchical agglomerative clustering (HAC) algorithm [10] with a collaborative approach, where the clustering is performed by multiple homogenous clustering nodes, each responsible for a subset of the data.

Selmadji et al. [P10] propose two possible algorithms for microservice identification: a hierarchical clustering algorithm, and a clustering algorithm based on gravity centers.

Sellami et al. [P31] use the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm [11] to identify microservices.

Evolutionary algorithms

Evolutionary algorithms are the second class of algorithms present in the literature. Evolutionary algorithms, and in particular genetic algorithms, are algorithms aimed at solving optimization problems by borrowing techniques from natural selection and genetics. These algorithms typically operate iteratively, selecting the best solutions from a population at each iteration (called a generation), and then combining the selected solutions to create new combinations for the next generation. The process is then repeated until certain criteria are met, for example a maximum number of generations, convergence of the population, or a quality indicator.

Examples of publications using Non-Dominated Sorting Algorithm II (NGSA-II) as multi-objective optimization algorithm to identify microservice candidates are Zhou and Xiong [P4], Kinoshita and Kanuka [P3], Zhang et al. [P5], Jin et al. [P23], and Li et al. [P22]. Carvalho et al. [P20] use the next generation of NSGA, NSGA-III, in order to find a solution for the problem.

Amiri [P19] rely on a genetic algorithm using Turbo-MQ [12] as fitness function.

Graph algorithms

Another common approach to identify microservice candidates is to use classical algorithms from graph theory.

For example, Mazlami et al. [P9] and Yang et al. [P2] use Kruskal's algorithm [13] to partition the graph into connected clusters. Kruskal's algorithm is a greedy algorithm that finds the minimum spanning forest for an undirected weighted graph.

Filippone et al. [P11] apply the Louvain community detection algorithm [14] to obtain the granularity of the microservices, and high-cohesive communities of nodes. The Louvain method is a greedy optimization algorithm that aims to extract non-overlapping communities from a graph, using the modularity value as optimization target. Hasan et al. [P32] use the Leiden algorithm [15], an improvement of the Louvain method that uses a refinement step to improve the quality of the communities.

Cao and Zhang [P34] use both the Leiden algorithm and the hierarchical clustering algorithm to identify microservice candidates. First, the Leiden algorithm is used to detect cohesive communities in static and dynamic analysis data, and then the hierarchical clustering algorithm is used to merge the communities into microservice candidates based on a call relation matrix.

Nitin et al. [P33] use Context sensitive Label Propagation (CARGO), an algorithm built on the principles of the Label Propagation algorithm [16]. CARGO is a community detection algorithm that is able to leverage the context embedded in the dependency graph to increase the cohesiveness of the communities.

Other algorithms

Other publications using algorithms that do not fit into one of the previous categories are grouped in a single category.

For example, the authors of Quattrocchi et al. [P6] incorporated a Mixed Integer Linear Programming (MILP) solver in their solution. The MILP solver is used to find a solution for an optimization problem that decomposes the software system into microservices, based on the placement of operations and data entities according to the users' needs. Filippone et al. [P16] use a linear optimization algorithm to solve a combinatorial optimization problem.

The approach taken by Kamimura et al. [P7] is to use a custom clustering algorithm named SArF [17], that aims at identifying software subsystems without the need for human intervention. Escobar et al. [P27] also use a custom clustering algorithm, detecting optimal microservices based on a meta-model of the class hierarchy.

Agarwal et al. [P18] propose an algorithm based on seed expansion. The seed classes are detected by using formal concept analysis. Then, using a seed expansion algorithm, clusters are created around the seeds by pulling in related code artefacts based on implementation structure of the software system [P18].

Eyitemi and Reiff-Marganiec [P24] use a rule-based approach to microservice candidate identification. The 6 proposed rules are based on the principles of high cohesion and low coupling, and using a step-based protocol can be used to manually decompose a monolithic system into microservices.

6.3.3 Metrics

The quality metrics used in the publications are summarized in Table 6. The metrics are used to quantitatively evaluate the quality of the generated microservice decomposition. Some of the algorithms require the use of a specific metric to guide the process, such as the fitness function in genetic algorithms.

Metric	**Publications**
Cohesion	[P25][P22][P8][P4][P6][P10][P20][P28][P16] [P11][P12][P13][P14][P17][P23][P5][P29] [P30][P31][P32][P33][P40][P36][P34]
Coupling	[P25][P22][P4][P10][P20][P28][P16][P11] [P18][P12][P14][P17][P23][P5][P29][P30] [P31][P32][P33][P40][P36][P34]
Network overhead	[P4][P6][P20][P16]
Complexity	[P8][P14][P17][P32]
CPU and memory usage	[P6][P5][P33]
Modularity	[P22][P28][P23][P12][P29][P31][P40][P36]
Other metrics	[P2][P1][P22][P9][P7][P10][P11][P17][P21] [P15][P30][P31][P32][P33][P40][P36][P34] [P35]
No metrics	[P27][P3][P26][P24]

Table 6: Quality metrics

We identified 87 metrics used in the publications, and categorized them in 6 categories. Publications using undisclosed quality metrics, and publications

using no metrics at all, are categorized into separate categories. Cohesion (24; 27.6%) and coupling (22; 25.3%) are the most frequently used metrics, followed by modularity (8; 9.2%), network overhead and complexity (4; 4.6% each), and CPU and memory usage (3; 3.4%). Publications using other metrics (18; 20.7%) account for the remaining metrics. Finally, the 4 publications that do not mention any quality metrics account for 4.6%.

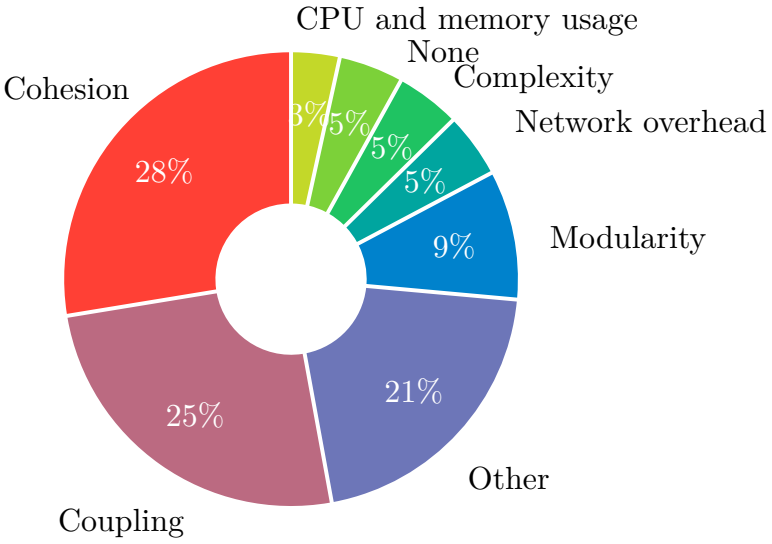


Figure 5: SDLC metric categories

Cohesion and coupling

The quality metrics most frequently mentioned in the literature are cohesion and coupling. The behaviour of information systems has been studied with the help of these metrics and others such as size and complexity since the 1970s [18]. As object-oriented programming became more popular, the concepts of cohesion and coupling were adapted to the new paradigm [19].

Throughout the years, many definitions of cohesion and coupling have been proposed both for procedural and object-oriented systems. For example, L. Briand, S. Morasca and V. Basili define cohesion as the tightness with which

related program features are grouped together, and coupling as the amount of relationships between the elements belonging to different modules of a system.

The publications in this review use different definitions for cohesion and coupling, and different methods of calculating them. For example, Selmadji et al. [P10] define (internal) cohesion as the number of direct connections between the methods of the classes belonging to a microservice over the number of possible connections between the methods of the classes. The authors then define internal coupling as the number of direct method calls between two classes over the total number of method calls in the application.

Others [P20] [P16] [P4] [P5] [P18] [P28] use a similar definition of cohesion, but they define (individual) coupling as the number of method calls from a microservice class to another class outside of the service boundary. The total coupling of the solution is the sum of the coupling of all microservices. Similarly, Filippone et al. [P11] define average cohesion and average coupling as ratio of the total cohesion and coupling respectively, to the number of microservices in the decomposition.

Jin et al. [P23] introduce the concept of inter-service cohesion and inter-call percentage (ICP) as coupling metrics. Several other publications use the metrics introduced by W. Jin et al. in their research [P12] [P29] [P31] [P33].

Another approach to cohesion and coupling is that of Santos and Silva [P14] and Lourenço and Silva [P17], who define cohesion as the percentage of entities accessed by a functionality. If all entities belonging to a microservice candidate are accessed each time a microservice candidate is accessed, the service is strongly cohesive. Coupling is defined as the percentage of the entities exposed by a microservice candidate that are accessed by other microservice candidates.

Al-Debagy and Martinek [P8] use the inverse of cohesion as a metric, named lack of cohesion (LCOM). It is calculated by the number of times a microservice uses a method from another microservice, divided by the number of operations multiplied by the number of unique parameters. This metric quantifies how the operations in a service are related to each other in terms of functionality.

Network overhead

Microservices are distributed systems, and communication between services is done over a network. The network overhead is the extra cost of this communication, and many authors consider it an important metric to consider when designing a microservice architecture.

Filippone et al. [P16] and others [P20] [P4] calculate the value based using a heuristic function that uses the size of primitive types of method call arguments to predict the total network overhead of a microservice decomposition. Carvalho et al. [P20] also includes the protocol overhead in the calculation, which is the cost of the communication protocol used to send messages between services (for example, TCP headers, HTTP headers, etc.).

Quattrocchi et al. [P6] measure network overhead as part of their operational cost metric. The metric also includes data management costs (CPU and memory).

Complexity

The complexity of a microservice candidate is another metric that can impact the quality of the microservice decomposition. Al-Debagy and Martinek [P8] defines complexity based on Number of Operations, a metric that uses Weighted Methods per Class (WMC), summing the number of methods in a class.

Santos and Silva [P14] define the complexity metric in terms of the functionality redesign effort, rather than the complexity of the microservice candidates. The metric is associated with the cognitive load of the software architect when considering a migration from monolith to microservice.

In another publication by the same co-author, Lourenço and Silva [P17] define complexity as the effort required to perform the decomposition, and expand the concept to uniform complexity, which is calculated by dividing the complexity of a decomposition by the maximum possible complexity.

CPU and memory usage

A non-functional metric that is considered by some authors is the CPU and/or memory usage of the microservices. Zhang et al. [P5] use this metric to evaluate the quality of the microservice decomposition, by predicting the average CPU and memory usage of the microservices. The prediction is made based on performance logs collected by executing the monolith application.

Quattrocchi et al. [P6] define operational costs as metric to minimize, which includes communication (network) and data management (CPU and memory) costs.

Nitin et al. [P33] do not utilize the CPU and memory usage directly as a metric, but instead assume the latency and throughput as indicators of performance.

Modularity

Modularity is a measure of independence of services, and can be divided into many dimensions, such as structure, concept, history, and dynamism [20]. Some definitions of modularity rely on the concepts of cohesion and coupling, and the balance between them.

Jin et al. [P23] use modularity as a metric to evaluate potential decompositions. The authors use Modularity Quality [21] and extend the concept with structural and conceptual dependencies to assess the modularity of microservice candidates.

Carvalho et al. [P20] introduce a metric named feature modularization, which maps a list of features supplied by the software architect onto classes and methods, determining the set of predominant features per microservice.

Other metrics

Lourenço and Silva [P17] introduce the concept of Team Size Reduction (TSR), which indicates if the average team size is shorter after the decomposition, by comparing the average number of authors per microservice to the total number of authors. A Team Size Reduction value of 1 indicates that the microservices architecture has the same number of authors as the monolith, while a value less than 1 indicates a reduction in the number of authors. Mazlami et al. [P9] make use of the TSR metric, as well as the Average Domain Redundancy (ADR) metric, which represents the amount of domain-specific duplication or redundancy between the microservices. The ADR metric uses a scale from 0 to 1, where 0 indicates no redundancy and 1 indicates that all microservices are redundant.

Carvalho et al. [P20] propose a metric called reuse, which measures the reusability of a microservice. Reuse is calculated as the number of times a microservice is called by the user, relying on dynamic analysis to collect this information.

The usage metric of an object-oriented software system, defined as the sum of the inheritance factor (is-a) and the composition factor (has-a) is used by Bandara and Perera [P28] as a part of the fitness function for the clustering algorithm.

Saidi et al. [P1] use the intra-domain and inter-domain data dependency metrics to delineate microservice boundaries, based on the read and write access pattern of the operations. In a similar fashion, Selmadji et al. [P10] talk about data autonomy determined by the internal and external data access of a microservice candidate.

Kamimura et al. [P7] introduce a metric called dedication score, which measures the relationships between services as a function of access frequency. Along with a modularity metric, the dedication score is used in their custom SArF dependency-based clustering algorithm [17].

The correlation metric is used by Yang et al. [P2] and indicates the degree of correlation between the microservices. The authors calculate the correlation in two ways: the number of co-occurrence of the problem domain, and the adjacency relationship between problem domains.

Ma et al. [P15] use the Adjusted Rand Index (ARI) as clustering evaluation criterion. The metric measures the similarity between two clusters in a decomposition, and ranges from -1 to 1, with 0 being the optimal value.

Hao et al. [P21] use the Matching Degree metric as quality indicator. The metric is calculated by dividing the number of intersections of database tables in a given microservice and a given cluster by the total number of tables used in the microservice.

Hasan et al. [P32] and Kalia et al. [P36] use the Size metric to evaluate the quality of the microservice decomposition. The metric measures how evenly the size of the proposed microservices is. The size metric was originallly proposed by Wu et al. [P38].

Santos and Paula [P35] use the silhouette coefficient originally proposed by P. Rousseeuw as evaluation metric. The silhouette coefficient assesses clustering consistency by comparing the average dissimilarity within the cluster.

No metrics

Some of the publications do not mention any quality metrics used in the evaluation of the proposed decomposition. These methods typically rely on the selection or approval of a software architect to choose the best decomposition, based on their experience and knowledge of the system. This is the case of Eyitemi and Reiff-Marganiec [P24], Romani et al. [P26], Amiri [P19], and Escobar et al. [P27].

The evaluation method by Kinoshita and Kanuka [P3] also does not rely on quantifying the quality of the microservice decomposition using metrics, but rather relies on the software architect's judgement to choose a qualitative decomposition.

7 Proposed solution

In this chapter, we propose **Modular Optimization to Service-oriented Architecture Integration Kit (MOSAIK)**, our solution for identification of microservice candidates in a monolithic application. The approach is based on the analysis of a dependency graph, that aggregates information from the static and evolutionary analysis of the source code.

7.1 Problem statement

The goal of this solution is to identify a set of microservice candidates that can be extracted from the source code of the given monolithic application, in order to automate the migration to a microservices architecture. The problem can be formulated as a graph partitioning problem, where the vertices correspond to the modules or classes in the monolithic application, and the edges represent the dependencies between them. The input of the algorithm is a representation M of the monolithic application, which exposes a set of functionalities M_F through a set of classes M_C, and history of modifications M_H. The triplet is described by Equation 1.

$$M_i = \left\{ M_{F_i}, M_{C_i}, M_{H_i} \right\} \tag{1}$$

The set of functionalities M_{F_i}, the set of classes M_{C_i}, and the set of historical modifications M_{H_i} are described by Equation 2.

$$
\begin{aligned}
M_{F_i} &= \{f_1, f_2, ..., f_j\} \\
M_{C_i} &= \{c_1, c_2, ..., c_k\} \\
M_{H_i} &= \{h_1, h_2, ..., h_l\}
\end{aligned} \tag{2}
$$

The output of the algorithm is a set of microservices S, according to Equation 3, where m is the number of microservices in the proposed decomposition.

$$
S_i = \{s_1, s_2, ..., s_m\} \tag{3}
$$

As each class belongs to exactly one microservice, the proposed decomposition S can be written as a surjective function f of M_{C_i} onto S_i as in Equation 4, where $f(c_i) = s_j$ if class c_i belongs to microservice s_j.

$$
f : M_{C_i} \rightarrow S_i \tag{4}
$$

A microservice that contains only one class is called a *singleton microservice*. Singleton microservices typically contain classes that are not used by any other class in the monolithic application. As an optimization of the microservice decomposition, these classes can be omitted from the final decomposition, as they do not have any functional contribution.

7.2 Design

We start by identifying the functional and non-functional requirements for the solution. Then, we propose a four-step approach to decomposition adapted from the microservice identification pipeline by T. Lopes and A. Silva.

- **Extraction**: the necessary information is extracted from the application and its environment.
- **Decomposition**: using the collected data, a decomposition of the application into microservices is proposed.
- **Visualization**: the proposed decomposition is visualized to facilitate the understanding of the architecture.
- **Evaluation**: the proposed decomposition is evaluated according to a set of quality metrics.

An overview of the architecture of the proposed solution is shown in Figure 6. The extraction step is comprised of two smaller steps: static analysis and evolutionary analysis. From the extracted information, a dependency graph is visualized. The decomposition step is based on the graph partitioning algorithm, which is used to identify the microservice candidates. Finally, the proposed decomposition is evaluated using a set of quality metrics.

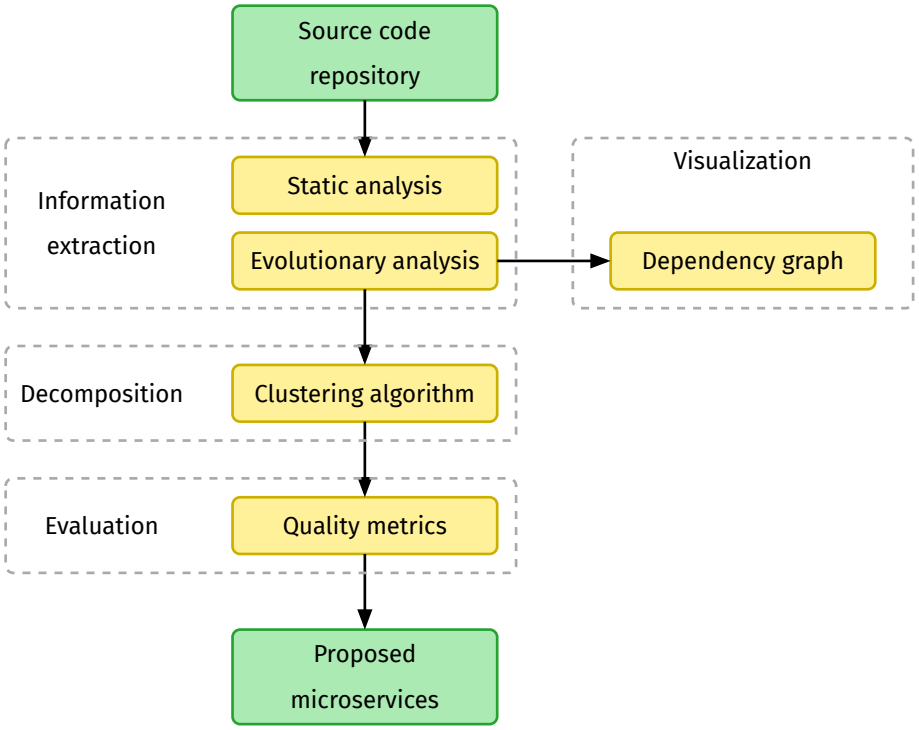


Figure 6: Architectural overview of the proposed solution

The next sections detail each of these steps, providing a comprehensive overview of the proposed solution. The process we describe is generic and not tied to any specific programming language or paradigm. We implemented a prototype of the proposed solution in Ruby, as the monolithic application we use for evaluation is written in Ruby. The implementation is available online[6].

[6]https://github.com/floriandejonckheere/mosaik

7.3 Requirements

Our approach needs to fulfill certain requirements. We make a distinction between functional and non-functional requirements. In software engineering, functional requirements describe requirements that impact the design of the application in a functional way . Non-functional requirements are additional requirements imposed at design-time that do not directly impact the functionality of the application .

The functional requirements we pushed forward for our proposed solution are as follows:

1. **Quality**: the solution provides a high-quality decomposition of the monolithic application, based on a set of quality metrics
2. **Automation**: the solution automates the decomposition process as much as possible
3. **Technology**: the solution can analyze applications written in the Ruby programming language[7]
4. **Visual**: the solution can output the proposed decomposition in a visual manner, to aid understanding of the process

The non-functional requirements identified for our solution are:

1. **Usability**: a software architect or senior software engineer can reasonably quickly get started with the solution
2. **Performance**: the solution performs the analysis, decomposition, and evaluation reasonably fast on the source code of a larger application
3. **Reuse**: The solution can successfully be reused for untested monolithic applications

[7]https://www.ruby-lang.org/

7.4 Extraction

Software development is typically done in multiple steps, either using the waterfall model, or using an iterative approach [22]. Analysis and design are two steps of early software development which often yield software development lifecycle artifacts in the form of use cases, process models, and diagrams. However, after the completion of the development and the subsequent deployment, these documents are often not kept up to date, and sometimes even lost. Hence, it is not always possible to use design documents for the information extraction phase. A software development artifact that is usually available is the source code repository of the application. Hence, we chose the source code repository as the starting point of the information extraction.

G. Mazlami, J. Cito and P. Leitner propose a microservice extraction model that includes three possible extraction strategies: *logical coupling* strategy, *semantic coupling* strategy, and *contributor coupling* strategy. In this thesis, we concentrate on the logical coupling strategy, and the contributor coupling strategy. The next sections describe in detail how these strategies are used for extracting information from the source code repository.

7.4.1 Structural coupling

Structural coupling is a measure of the dependencies between software components. The dependencies can take the form of control dependencies, or data dependencies . Control dependencies are dependencies between software components that are related to the flow of control of the software system (e.g. interleaving method calls). Data dependencies relate to the flow of data between software components (e.g. passing parameters). MOSAIK extracts structural coupling information using static analysis of the source

code . As MOSAIK is intended to collect information from monolithic applications written in the Ruby programming language, the static analysis is limited to the information that is embedded in the source code. Ruby is a dynamic language, which means that only incomplete type information can be extracted using static analysis. In particular, some techniques like meta-programming and dynamic class loading may affect the accuracy of the extracted information.

Our solution analyzes the source code of the monolithic application using the `parser` library[8]. The library is written in Ruby and can be used to parse Ruby source code files and extract the AST of the source code. Iterating over the AST of the monolithic application, MOSAIK extracts the references between classes.

Using this information, a call graph is constructed that represents the structural coupling of the monolithic application. For each class in the monolithic application $c_i \in M_C$, a vertex is created in the call graph. References between classes are represented as directed edges between the vertices.

A directed edge is created for each reference between two classes c_i, c_j. This edge describes three types of references: (i) static method calls between two methods m_i and m_j of the classes c_i, c_j *(method-to-method)*, (ii) references from method m_i to an object of class c_j *(method-to-entity)*, and (iii) associations between entities of class c_i and c_j *(entity-to-entity)* [P16].

Hence, the structural coupling N_s for each pair of classes $c_i, c_j \in M_C$ is defined as the sum of the number of references between the classes, as described in Equation 5.

[8]https://github.com/whitequark/parser

$$N_s(c_i, c_j) = \sum_{m_i \in c_i, m_j \in c_j} ref_{mm}(m_i, m_j) + ref_{mc}(m_i, c_j) + ref_{cc}(c_i, c_j) \quad (5)$$

The ref_{mm}, ref_{mc}, and ref_{cc} functions return the number of references between the two methods m_i and m_j, method m_i and class m_j, and classes c_i and c_j respectively.

As L. Carvalho et al. note, the choice of granularity is an important decision in the extraction of microservices. Existing approaches tend to use a more coarse-grained granularity (e.g. on the level of files or classes) rather than a fined-grained granularity (e.g. on the level of methods). Using a coarse-grained granularity can lead to a smaller number of microservices that are responsible for a larger number of functionalities. A fine-grained granularity can lead to a much larger number of microservices, which can decrease the maintainability of the system. Hence, a trade-off between the two granularities must be made. MOSAIK uses a coarse-grained granular approach, using the classes of the monolithic application as the starting point for the extraction of microservices.

Consider the extraction algorithm in pseudocode in Algorithm 1. The algorithm first initializes an empty three-dimensional call matrix, which stores the number and type of references between classes in the monolithic application. The algorithm iterates over all classes in the monolithic application, and for each method in the class, it parses the method body. All references from the method body are extracted, and the receiver and type of reference are stored in the call graph.

Algorithm 1: Structural coupling extraction algorithm

```
calls ← array[][][]

for each ( class : classes )
  for each ( method : class.methods )
    for each ( reference : method.references )
      receiver ← reference.receiver
      type ← reference.type
      calls[class][receiver][type] ← 1

return calls;
```

Algorithm 1: Structural coupling extraction algorithm

7.4.2 Logical coupling

The logical coupling strategy is based on the Single Responsibility Principle [23], which states that a software component should only have one reason to change. Software design that follows the Single Responsibility Principle groups together software components that change together. Hence, it is possible to identify appropriate microservice candidates by analyzing the history of modifications of the classes in the source code repository. Classes that change together, should belong in the same microservice. Let M_H be the history of modifications of the source code files of the monolithic application M. Each change event h_i is associated with a set of associated classes c_i that were changed during the modification event at timestamp t_i, as described by Equation 6 [P9].

$$h_i = \{c_i, t_i\} \tag{6}$$

If c_1, c_2 are two classes belonging to the same change event h_i, then the logical coupling is computed as follows in Equation 7 [P9].

$$\Delta(c_1, c_2) = \sum_{h \in M_H} \delta_{h(c_1, c_2)} \tag{7}$$

Where δ is the change function.

$$\delta(c_1, c_2) = \begin{cases} 1 \text{ if } c_1, c_2 \text{ changed in } h_i \\ 0 \text{ otherwise} \end{cases} \tag{8}$$

Then, Equation 7 is calculated for each change event $h_i \in M_H$, and each pair of classes c_1, c_2 in the change event. Thus, the logical coupling N_c for each pair of classes $c_i, c_j \in M_C$ is defined as the sum of the logical coupling for each change event $h_i \in M_H$.

$$N_c(c_1, c_2) = \Delta(c_1, c_2) \tag{9}$$

Consider the extraction algorithm in pseudocode in Algorithm 2. First, a co-change matrix is initialized, which stores the number of times two files have changed together in a two-dimensional matrix. The algorithm then iterates over all commits in the source code repository, and for each commit, retrieves the changes between the commit and its parent. Then, it iterates over each pair of files in the changelist, and increments the co-change matrix for the pair of files.

Algorithm 2: Logical coupling extraction algorithm

```
cochanges ← array[][]

for each ( commit : git.log )
    parent ← commit.parent
    parent_diff ← diff ( commit, parent )

    for each ( file_one : parent_diff.files )
        for each ( file_two : parent_diff.files )
            cochanges[file_one][file_two] ← 1

return cochanges;
```

Algorithm 2: Logical coupling extraction algorithm

7.4.3 Contributor coupling

Conway's law states that the structure of a software system is a reflection of the communication structure of the organization that built it [24]. The contributor coupling strategy is based on the notion that the communication structure can be recovered from analyzing the source code repository [P9]. Grouping together software components that are developed in teams that have a strong communication paradigm internally can lead to less communication overhead when developing and maintaining the software system. Hence, identifying microservice candidates based on the communication structure of the organization can lead to more maintainable software systems.

Let M_H be the history of modifications of the source code files of the monolithic application M. Each change event h_i is associated with a set of associated classes c_i that were changed during the modification event at

timestamp t_i. The change event h_i is also associated with a set of developers $d_i \in M_D$, as stated in Equation 10 [P9]. M_D is the set of developers that have contributed to the source code repository of the monolithic application.

$$h_i = \{c_i, t_i, d_i\} \tag{10}$$

$H(c_i)$ is a function that returns the set of change events that have affected the class c_i, and $D(c_i)$ returns the set of developers that have worked on the class c_i.

$$H(c_i) = \{h_i \in M_H \mid c_i \in h_i\} \tag{11}$$

$$D(c_i) = \{d_i \in M_D \mid \forall h_i \in H(c_i) : d_i \in h_i\} \tag{12}$$

Then, Equation 12 is calculated for each class $c_i \in M_C$ in the monolithic application.

Finally, the contributor coupling N_d for each pair of classes $c_i, c_j \in M_C$ is defined as the cardinality of the intersection of the sets of developers that have contributed to the classes c_i, c_j [P9].

$$N_d(c_1, c_2) = |D(c_i) \cap D(c_j)| \tag{13}$$

Consider the extraction algorithm in pseudocode in Algorithm 3. The algorithm first initializes the co-authorship matrix, which is a two-dimensional array that stores the (unique) authors of each file in the source code repository. Then, it iterates over all commits in the source code repository, and for each commit, retrieves the changes between the commit and its parent. Finally, iterating over each file in the changelist, the algorithm adds the author(s) of the commit to the file's entry in the co-authorship matrix.

Algorithm 3: Contributor coupling extraction algorithm

```
coauthors ← array[][]

for each ( commit : git.log )
    parent ← commit.parent
    parent_diff ← diff ( commit, parent )

    for each ( file : parent_diff.files )
        coauthors[file] ← commit.authors

return coauthors;
```

Algorithm 3: Contributor coupling extraction algorithm

7.4.4 Dependency graph

As a final step in the information extraction phase, an edge-weighted graph $G = (V, E)$ is constructed, where V is the set of classes in the monolithic application, and E is the set of edges between classes that have an interdependency based on the discussed information extraction strategies. The weight for the edge e_i between classes $c_j, c_k \in V$ is calculated as the weighted sum of the call graph N_s representing the structural coupling, the co-change matrix N_c representing the logical coupling, and the co-authorship matrix N_d representing the contributor coupling. The weights $\omega_s, \omega_c, \omega_d \in [0, 1]$ are used to balance the contribution of the structural, logical, and contributor coupling respectively, as described in Equation 14. This makes the strategy adaptive and flexible [P35].

$$w(e_i) = w(c_j, c_k) = \omega_s N_s(c_j, c_k) + \omega_c N_c(c_j, c_k) + \omega_d N_d(c_j, c_k) \quad (14)$$

An illustration of the graph G is presented in Figure 7.

$G = (V, E)$

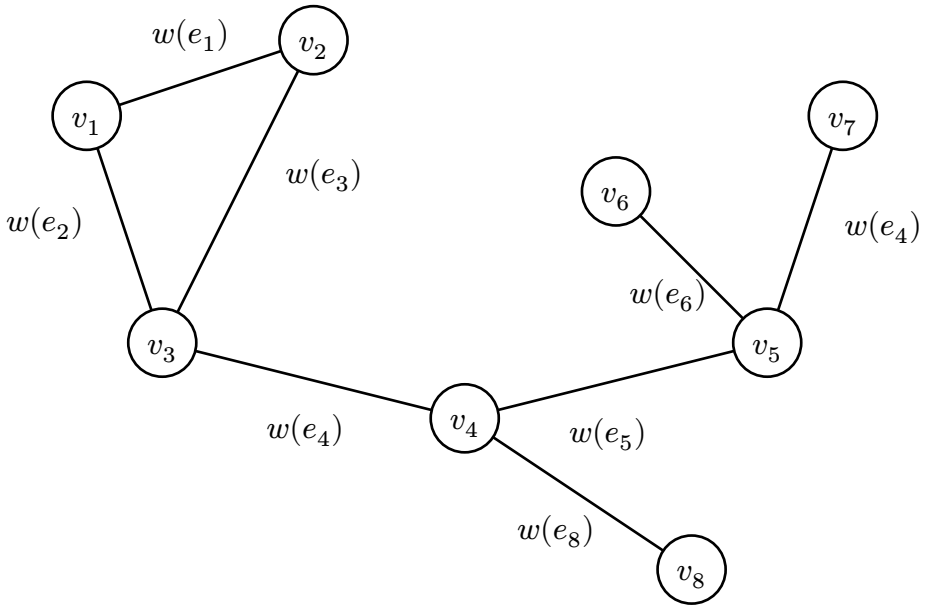


Figure 7: Dependency graph

7.5 Decomposition

Monolith decomposition is the process of identifying microservice candidates in a monolithic application. The goal of this process is to split the monolith into smaller, more manageable software components which can be deployed independently [25]. Traditionally, monolith decomposition is a manual process that requires a deep understanding of the software architecture and business requirements. However, the burden of manual decomposition can be alleviated by using automated tools and algorithms. The knowledge of software architects should be leveraged where possible to guide the decomposition process, without imposing the requirement of a deep understanding of the software architecture. For example, Y. Li et al. propose a method that utilizes expert knowledge, however requires the recommendations to be written in a domain-specific language, increasing the burden on the architect.

MOSAIK implements an automated identification of microservice candidates in a monolithic application using clustering algorithms. The decomposition process can be fine-tuned by assigning an importance to the different types of coupling strategies. This way, the software architect can decide which coupling strategies are most relevant to the decomposition process.

Clustering algorithms group similar elements together based on one or multiple criteria. Generally these algorithms work iteratively either top-down or bottom-up. Top-down algorithms start by assigning all elements to one big cluster, and then progressively split it into smaller clusters until a stopping criterion is met. Bottom-up algorithms on the other hand, start by assigning each element to its own cluster, then merge suitable clusters together in succession, until a stopping criterion is met.

Selection of algorithm

In Chapter 6.3.2, we performed an analysis of the state of the art in clustering algorithms used for microservice candidate identification. We considered the following criteria when selecting the most suitable algorithm for our task:

1. **Automation**: The algorithm should not require architectural knowledge up-front (e.g. number of clusters)
2. **Performance**: The algorithm should be able to handle large datasets efficiently

The first criteria disqualifies algorithms that require specifying the number of clusters up-front, such as Spectral Clustering, K-Means, and Agglomerative Clustering. Search-based algorithms (e.g. genetic, linear optimization) were considered as well, due to their inherent ability to optimize multiple objectives [P20]. However, they require a lot of computing resources, and proper fine-tuning of parameters such as population size, mutation rate, and crossover rate, which makes them less suitable. Affinity Propagation is an algorithm that doesn't require specifying the number of clusters up-front, but it is computationally expensive as well [9].

We found that the Louvain [14] and Leiden [15] algorithms are the most suitable for this task, as they are designed for optimizing modularity in networks. The algorithms are iterative and hierarchical, which makes them fast and efficient.

Similarly, in 2019 S. Rahiminejad, M. Maurya and S. Subramaniam performed a topological and functional comparison of community detection algorithms in biological networks. They analyzed six algorithms based on certain criteria such as appropriate community size (not too small or too large), and performance speed. The authors found that the Louvain algorithm [14] performed best in terms of quality and speed.

The Louvain/Leiden algorithm

The Louvain algorithm, introduced by V. Blondel et al., is an algorithm for extracting non-overlapping communities in large networks. The algorithm uses a greedy optimization technique to maximize the modularity of the network.

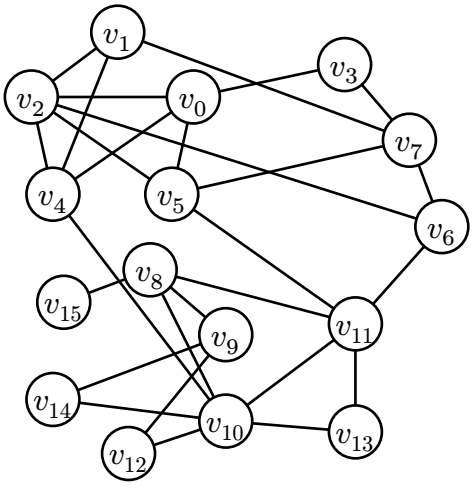
Modularity is a measure of the strength of division of a network. Networks with high modularity have dense connections between the internal vertices of a community, and sparse connections between vertices of different communities. The domain of the metric is between -0.5 (non-modular clustering) and 1 (fully modular clustering). Optimizing the modularity theoretically results in the best possible clustering of the network, though for numerical computing reasons, the algorithm uses heuristics to approach the optimal solution.

The modularity of a network is defined as follows [27]:

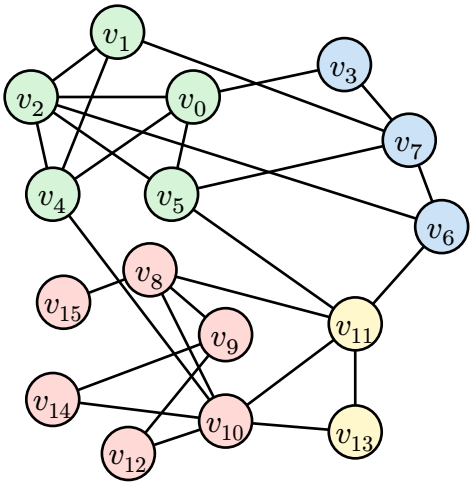
$$Q = \frac{1}{2m} \sum_{i=1}^{N} \sum_{j=1}^{N} \left[A_{\text{ij}} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j) \tag{15}$$

Where:

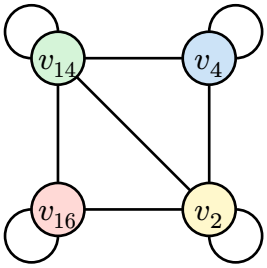
- A is the adjacency matrix
- k_i and k_j are the degrees of the vertices i and j respectively
- m is the number of edges in the network
- N is the total number of vertices in the network
- c_i and c_j are the communities to which vertices i and j belong
- $\delta(c_i, c_j)$ is 1 if c_i and c_j are in the same cluster, and 0 otherwise



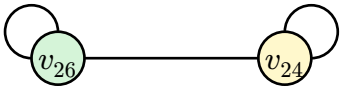
Original network



Modularity optimization



Community aggregation



Second iteration

Figure 8: Louvain algorithm intermediate steps

Algorithm 4: Louvain algorithm

graph ← original network

loop

  for each *vertex* **in** *graph*

    Put *vertex* in its own community

  // Phase 1: local modularity optimization

  for each *neighbour* **in** *vertex.neighbours*

    Move *vertex* to community of *neighbour*

    if modularity gain

      break

  // Phase 2: community aggregation

  for each (*community* : *graph*)

    Reduce *community* to a single vertex

  if modularity no longer increases

    break

Algorithm 4: Louvain algorithm pseudocode

The Louvain algorithm operates in two phases. In the first phase, the algorithm optimizes the modularity locally by moving each vertex into the community of their neighbour that yield the best modularity gain. This step is repeated for each vertex until a local maximum is reached.

Then, the algorithm aggregates each community in a single vertex, while preserving the network structure. The algorithm can then be applied iteratively to the new network, until the modularity cannot be further increased.

A visualization of the intermediate steps of the Louvain algorithm is shown in Figure 8.

A major disadvantage of the Louvain algorithm is that it can only detect non-overlapping communities [14]. This means that a software component can only belong to one microservice, which is not in line with the principle of reuse in software engineering. The algorithm has also been proven to generate small and disconnected communities [15], which is not desirable in the context of microservices [28].

In 2019, V. Traag, L. Waltman and N. van Eck introduced the Leiden algorithm, an improvement of the Louvain algorithm that addresses the disconnected community problem. Similarly to the Louvain algorithm, the Leiden algorithm optimizes the quality of the network using the Constant Potts Model [29]:

$$\mathcal{H}(G, \mathcal{P}) = \sum_{C \in \mathcal{P}} |E(C, C)| - \gamma \binom{\|C\|}{2} \tag{16}$$

The Leiden algorithm operates in three phases. The first and last phases equal those of the Louvain algorithm (i.e., local modularity optimization and community aggregation).

In the second phase, the algorithm performs a refinement of partition on each small community. The refinement ensures that the algorithm does not get stuck in a local optimum using a probability distribution. The Leiden algorithm has been shown to outperform the Louvain algorithm in terms of quality and speed [15].

Algorithm 5: Leiden algorithm (refinement)

```
// Phase 2: partition refinement
for each ( community : graph )
    partition ← community
    for each ( well connected vertex : partition )
        if vertex is a singleton
            assign vertex to new community
            using probability distribution P
```

Algorithm 5: Leiden algorithm (refinement)

Although the Leiden algorithm is more complex than the Louvain algorithm, it is more suitable for our task due to its ability to detect overlapping communities.

7.6 Visualization

MOSAIK is able to generate visualizations of the microservice decomposition process. The information extraction step can be visualized as a dependency graph where the vertices represent the classes of the monolithic application, and the edges represent the dependencies between the classes. The edges are weighted according to the calculated coupling factor between the classes. Depending on the weights assigned to the structural, logical, and contributor coupling, the generated graph can visualize a different view of the monolithic application.

Visualization of the decomposition process yields a dependency graph where the vertices represent the microservice candidates, and the edges represent the dependencies between the microservice candidates.

The graphs generated by the different steps in the decomposition process are saved as CSV files. The visualizations are generated using a custom implementation, that converts the graphs into the DOT language[9], before rendering them as images using the `dot` tool from the Graphviz package[10].

[9]https://graphviz.org/doc/info/lang.html
[10]https://graphviz.org/

7.7 Evaluation

Several quality metrics were considered to evaluate the quality of the microservice decomposition. L. Carvalho et al. conducted an analysis of the criteria used to evaluate the quality of microservice-based architectures and found that the most common metrics are cohesion and coupling, as does our literature review win Chapter 6. In addition to this, they conclude that in many cases the decomposition process is guided only by these two metrics. In other cases, the software architects use other metrics (e.g. network overhead, CPU usage), although coupling and cohesion remain the dominant ones. According to experts, using four quality metrics is a good balance between the number of metrics and the quality of the solution [30].

I. Candela et al. studied a large number of metrics to evaluate the quality of microservice-based architectures, including network overhead, CPU, and memory. They affirmed the need for more than two metrics to ensure the quality of the decomposition.

We distinguish between functional and non-functional metrics.

7.7.1 Functional metrics

Coupling

Coupling is a measure of the degree of interdependence between modules in a software system [31]. In the context of microservices, individual coupling is defined as the sum of static calls from methods within a microservice candidate M_c in a solution S to another microservice candidate $M_c \in S$ [P20].

$$coup(M_c) = \sum_{v_i \in M_c, v_j \notin M_c} calls(v_i, v_j) \tag{17}$$

Where v_i and v_j are methods belonging and not belonging to M_c respectively, and $calls(v_i, v_j)$ returns the number of method calls present in the body of method v_i made to method v_j.

The total coupling of a solution is the sum of the individual couplings of all microservice candidates M_c.

$$Coupling = \sum_{M_c \in S} coup(M_c) \tag{18}$$

A lower total coupling indicates a better decomposition.

Cohesion

Cohesion is a measure of the degree to which internal elements of a module in a software system are related to each other [31]. Cohesion of microservice candidates is defined as the number of static calls between methods within a microservice candidate M_c in a solution S, divided by the total number of possible static method calls in M_c [P20]. The metric indicates how strongly related the methods internal to a microservice candidate are.

To compute the individual cohesion of a microservice candidate M_c, we first introduce the boolean function ref, which indicates the existence of at least one method call between methods v_i and v_j in M_c.

$$ref(v_i, v_j) = \begin{cases} 1 \text{ if } calls(v_i, v_j) > 0 \\ 0 \text{ otherwise} \end{cases} \tag{19}$$

The cohesion of a microservice candidate M_c is then calculated as described in Equation 20, where $|M_c|$ is the cardinality of method calls in M_c.

$$coh(M_c) = \frac{\sum_{v_i \in M_c, v_j \in M_c} ref(v_i, v_j)}{|M_c| \ \frac{|M_c| - 1}{2}} \tag{20}$$

The total cohesion of a solution is the sum of the individual cohesion of all microservice candidates M_c.

$$Cohesion = \sum_{M_c \in S} coh(M_c) \tag{21}$$

A higher total cohesion indicates a better decomposition.

Size

Size of a microservice candidate can be defined in several different ways. In Chapter 6, we identified several publications that use the size metric as introduced by Wu et al. [P38], who define size as the number of source code files or classes in a microservice candidate. Other definitions of size include the number of methods, or the number of lines of code. However, these definitions have the disadvantage that they only describe the size of a microservice candidate superficially, without considering the structure of the code.

Fitzpatrick [32] developed the ABC size metric, which takes into account the number of assignments, branches, and conditions in a method. Using not only the number of lines of code, but also the complexity of the code, the ABC size metric describes the size of a method more accurately. Methods with a high ABC size are harder to understand, and more prone to errors and bugs.

The ABC size of a method consists of a vector $\langle A, B, C \rangle$, where:

- A is the number of assignments (explicit transfer of data into a variable)
- B is the number of branches (explicit branch out of the current scope)
- C is the number of conditions in the method (logical test)

ABC sizes are written as an ordered triplet of numbers, in the form $\langle A, B, C \rangle$, for example $\langle 7, 12, 3 \rangle$. To convert the ABC size vector into a scalar value, the magnitude of the vector is calculated using the Euclidean norm.

$$|\text{ABC}| = \sqrt{A^2 + B^2 + C^2} \tag{22}$$

The ABC size of a method can vary between programming languages due to semantic differences in the language constructs. As such, the interpretation of ABC size values is language-dependent. For example, in Ruby an ABC value of ≤ 17 is considered satisfactory, a value between 18 and 30 unsatisfactory, and > 30 is considered dangerous[11]. In this study we do not intend to evaluate the quality of individual methods, but rather the quality of the decomposition as a whole. As such, we use the average of the ABC sizes of all methods in a microservice candidate to calculate the size of the microservice candidate.

Formalized, the ABC size metric can be defined as in Equation 23. The functions $asgn(v_i)$, $brch(v_i)$, and $cond(v_i)$ return the number of assignments, branches, and conditions in method v_i respectively.

$$abc(v_i) = \sqrt{asgn(v_i)^2 + brch(v_i)^2 + cond(v_i)^2} \tag{23}$$

To compute the individual size of a microservice candidate M_c, we sum the ABC sizes of all methods in M_c, and divide by the number of methods in M_c

$$size(M_c) = \frac{\sum_{v_i \in M_c} abc(v_i)}{|v_i|} \tag{24}$$

The total size of a solution is the sum of the individual sizes of all microservice candidates M_c.

$$Size = \sum_{M_c \in S} size(M_c) \tag{25}$$

A lower total size indicates a better decomposition, as smaller microservices are easier to understand and maintain. However, a very low size may indicate

[11] https://docs.rubocop.org/rubocop/cops_metrics.html

that the microservice candidates are too small, and that the decomposition is too fine-grained.

Complexity

Complexity is a measure of the number of operations performed by a method in a class [P8]. We use the number of operations to compute the individual complexity of a microservice candidate M_c in a solution S. The metric is based on the Weighted Methods per Class (WMC) metric, which is the sum of the complexities of all methods in a class [33]. Classes and methods with lower complexity are associated with better maintainability and understandability.

$$numops(M_c) = \sum_{v_i \in M_c} ops(v_i) \tag{26}$$

Where *ops* returns the number of operations performed by method v_i.

The total complexity of a solution is the sum of the individual complexities of all microservice candidates M_c.

$$Complexity = \sum_{M_c \in S} numops(M_c) \tag{27}$$

A lower total complexity indicates a better decomposition.

7.7.2 Non-functional metrics

Network overhead

Microservices communicate over a network with one another, which introduces overhead in terms of latency and bandwidth. In order to keep this overhead low, it is important that method calls between microservices are kept to a minimum, and that the size of the data exchanged (e.g. parameters and return values) is kept small. Using the source code of the monolith application, we can estimate the network overhead of a method call

by inspecting the primitive types of the parameters and return values of the methods involved in the call [P16].

The heuristic function $h(v_i, v_j)$ estimates the network overhead of a method call from a method v_i to a method v_j, as described by Equation 28. The value for $h(v_i, v_j)$ is calculated by summing the size of the primitive types of the parameters and return values of the method call. The heuristic does not take into account the overhead of the communication protocol (e.g. HTTP headers) or data management overhead (e.g. (de-)serialization).

$$h(v_i, v_j) = \sum_{p \in \mathcal{P}(v_i)} size(p) + \sum_{r \in \mathcal{R}(v_j)} size(r) \tag{28}$$

$\mathcal{P}(v_i)$ returns the set of parameters of method v_i, and $\mathcal{R}(v_j)$ returns the return value(s) of method v_j. The *size* function returns the size of the primitive type of the given parameter.

The individual network overhead of a microservice candidate M_c can be written as the sum of the network overheads of all method calls between methods in M_c and methods not in M_c.

$$ovh(M_c) = \sum_{v_i \in M_c, v_j \notin M_c} h(v_i, v_j) \tag{29}$$

The total network overhead of a solution is the sum of the individual network overheads of all method calls between microservice candidates M_c and M_c in solution S.

$$Overhead = \sum_{M_c \in S} ovh(M_c) \tag{30}$$

However, this method has severe limitations in a dynamic language such as Ruby, where the types of the parameters and return values are not explicitly declared and may vary at runtime. In some cases, the primitive types can

be inferred from the method body, but in general, this is a difficult problem to solve.

8 Case study

In this chapter, we present a case study to evaluate the proposed solution in the context of a real-word use case. We aim to answer the following research question:

Research Question 3: How can static analysis of source code identify module boundaries in a modular monolith architecture that maximize internal cohesion and minimize external coupling?

We start by presenting the background information about the use case, followed by a description of the experimental setup. Next, we utilize MOSAIK on the use case application, evaluate the effectiveness, and present the results. Finally, we discuss the results and analyze the implications of the proposed solution in a broader context.

8.1 Background

The case study is based on an application written called NephroFlow Link. The application is developed by Nipro Digital Technologies Europe NV[12], a subsidiary of Nipro Europe Group Companies[13]. Nipro Group is a leading global manufacturer of medical devices, specialized in renal care

[12]"https://niprodigital.com"
[13]"https://www.nipro-group.com"

products. NephroFlow Link, part of the NephroFlow Product Suite[14], is a monolithic application that allows the NephroFlow Healthcare Platform to communicate with the dialysis machines installed at dialysis centers, and vice versa. NephroFlow Link is responsible for collecting data from the dialysis machines, processing it, and sending it to the NephroFlow Healthcare Platform for storage and visualization.

Dialysis machines typically measure data essential for the dialysis treatment, such as vital signs, blood flow rate, and dialyzer efficiency. Nurses and practitioners use this information to evaluate the condition of the patient, and the effectiveness of the treatment.

Nipro Group has deployed NephroFlow Link in several dialysis centers and hospitals across Europe, Central America, and India, collectively ensuring connection to hundreds of dialysis machines. To ensure the patient's information security, the application is usually deployed per dialysis center, and the data is stored in a secure cloud environment.

The application is written in the Ruby programming language[15] as a single-threaded process, deployed as a single unit. In theory, it is not a stateful application, as it only stores transitional data (e.g. for rate-limiting purposes) using the Redis key-value datastore[16]. The source code of the application is hosted in a private Github repository[17], and is not publicly available.

The codebase of the application is rapidly becoming increasingly complex, which decreases the development velocity of new features and device integrations. When it is deployed at bigger sites with up to 400 dialysis machines, the throughput and latency suffer and performance issues arise.

[14]https://www.nipro-group.com/en/our-offer/products-services/nephroflowtm-product-suite

[15]https://www.ruby-lang.org

[16]https://www.redis.com

[17]https://github.com

For these reasons, the application would benefit from a software architectural overhaul. While microservices-based architecture would allow the application to scale efficiently, it also introduces a maintenance overhead for the software developers. Since the number of developers working on NephroFlow Link is limited, the extra burden on the software developers should be low. Hence, decomposing this application into a modular monolith architecture would prove beneficial.

8.2 Experimental setup

J. Lourenço and A. Silva analyzed multiple source code repositories and concluded that repositories with a large number of committers perform better when considering the contributor coupling in various scenario's. Approaches using contributor coupling achieve comparable results as approaching using a structural coupling on source code repositories with a large number of committers. Since the number of committers to NephroFlow Link is limited, we use multiple coupling strategies to decompose the application. Seven test scenario's were designed by combining configurations obtained through varying the weights of the coupling strategy [P35]. The weights ω_s, ω_c, and ω_d refer to the structural, logical, and contributor coupling respectively. Refer to Table 7 for a list of the test configurations.

ω_s	ω_c	ω_d	**Scenario**
1	0	0	*structural*
0	1	0	*logical*
0	0	1	*contributor*
1	1	0	*structural-logical*
1	0	1	*structural-contributor*
0	1	1	*logical-contributor*
1	1	1	*structural-logical-contributor*

Table 7: Test configurations

The source code repository of NephroFlow Link is hosted in a private Github[18] organization. The application contains 208 Ruby source code files, with a total of 9288 Source Lines of Code (SLOC), as measured by the `cloc` tool[19]. Only the application code is considered, excluding the test code and configuration files.

The repository contains a `main` branch with the latest code, and several branches for released and maintained versions. For the purpose of this study, we only consider the `main` branch, from the release of NephroFlow Link version 5.0 on up until the release of NephroFlow Link version 5.2 on , which is the most recent release at the time of writing. The static analysis is performed on the source code as it appears in the most recent commit. The commits from the `dependabot` contributor are omitted, as they

[18]https://www.github.com
[19]https://github.com/AlDanial/cloc

are automatically generated by Github to update the dependencies of the application[20].

We identified nine software developers that have contributed to the software in the analyzed timespan, although only five developers have more than ten commits attributed to them. The top two contributors are responsible for 78% of the commits, while the other seven contributors count for remaining 21%.

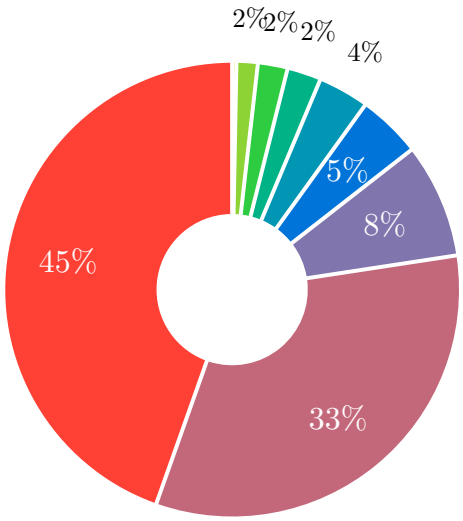


Figure 9: Contributor statistics

An overview of the source code repository is presented in Table 8.

SLOC	Classes	Methods	Commits	Contributors
9288	287	820	332	9

Table 8: Source code statistics

[20]https://github.com/features/security

8.3 Evaluation and results

8.4 Discussion

8.5 Threats to validity

9 Conclusion

9.1 Future work

References

[1] P. Ralph and S. Baltes, "Paving the Way for Mature Secondary Research: The Seven Types of Literature Review," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Singapore Singapore: ACM, Nov. 2022, pp. 1632–1636. doi: 10.1145/3540250.3560877.

[2] B. Kitchenham and S. Charters, "Guidelines for performing Systematic Literature Reviews in Software Engineering." 2007.

[3] K. Peffers, M. Rothenberger, T. Tuunanen, and S. Chatterjee, "A Design Science Research Methodology for Information Systems Research," vol. 24, no. 3. pp. 45–77, 2007.

[4] N. Alshuqayran, N. Ali, and R. Evans, *A Systematic Mapping Study in Microservice Architecture*. 2016, pp. 44–51.

[5] C. Pahl and P. Jamshidi, *Microservices: A Systematic Mapping Study*. 2016, pp. 137–146.

[6] R. M. Gupta, *Project Management*. Prentice-Hall of India Pvt.Limited, 2011.

[7] "IEEE Guide for Software Requirements Specifications." pp. 1–26, 1984.

[8] M. Jackson, *Problem frames: analyzing and structuring software development problems*. USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[9] B. J. Frey and D. Dueck, “Clustering by Passing Messages Between Data Points,” *Science*, vol. 315, no. 5814, pp. 972–976, 2007, doi: 10.1126/science.1136800.

[10] F. Murtagh and P. Legendre, “Ward's Hierarchical Agglomerative Clustering Method: Which Algorithms Implement Ward's Criterion?,” *Journal of Classification*, vol. 31, no. 3, p. 274–275, 2014, doi: 10.1007/s00357-014-9161-z.

[11] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, “A density-based algorithm for discovering clusters in large spatial databases with noise,” in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, in KDD'96. Portland, Oregon: AAAI Press, 1996, p. 226–227.

[12] B. Mitchell, M. Traverso, and S. Mancoridis, “An Architecture for Distributing the Computation of Software Clustering Algorithms,” in *Proceedings Working IEEE/IFIP Conference on Software Architecture*, Amsterdam, Netherlands: IEEE Comput. Soc, 2001, pp. 181–190. doi: 10.1109/WICSA.2001.948427.

[13] J. Kleinberg and É. Tardos, *Algorithm Design*. Pearson Education, 2006, pp. 142–151.

[14] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, “Fast unfolding of communities in large networks,” *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, 2008, doi: 10.1088/1742-5468/2008/10/P10008.

[15] V. A. Traag, L. Waltman, and N. J. van Eck, “From Louvain to Leiden: guaranteeing well-connected communities,” *Scientific Reports*, vol. 9, no. 1, Mar. 2019, doi: 10.1038/s41598-019-41695-z.

[16] X. Zhu and Z. Ghahramani, "Learning from Labeled and Unlabeled Data with Label Propagation," p. , 2003.

[17] K. Kobayashi, M. Kamimura, K. Kato, K. Yano, and A. Matsuo, "Feature-gathering dependency-based software clustering using Dedication and Modularity," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, IEEE, 2012. doi: 10.1109/icsm.2012.6405308.

[18] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," vol. 15, no. 12, 1972.

[19] J. Eder, G. Kappel, and M. Schre, "Coupling and Cohesion in Object-Oriented Systems," 1995.

[20] I. Candela, G. Bavota, B. Russo, and R. Oliveto, "Using Cohesion and Coupling for Software Remodularization: Is It Enough?," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 3, 2016, doi: 10.1145/2928268.

[21] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*, 1998, pp. 45–52. doi: 10.1109/WPC.1998.693283.

[22] W. W. Royce, "Managing the Development of Large Software Systems," in *Technical Papers of Western Electronic Show and Convention*, Aug. 1970, pp. 1–9.

[23] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. USA: Prentice Hall PTR, 2003.

[24] M. E. Conway, "How do committees invent?," 1968.

[25] Z. Dehghani, “How to break a Monolith into Microservices.” Apr. 24, 2018. Accessed: Apr. 10, 2024. [Online]. Available: https://martinfowler.com/articles/break-monolith-into-microservices.html

[26] S. Rahiminejad, M. R. Maurya, and S. Subramaniam, “Topological and Functional Comparison of Community Detection Algorithms in Biological Networks,” *BMC Bioinformatics*, vol. 20, no. 1, p. 212–213, Dec. 2019, doi: 10.1186/s12859-019-2746-0.

[27] S. H. Hairol Anuar *et al.*, “Comparison between Louvain and Leiden Algorithm for Network Structure: A Review,” *Journal of Physics: Conference Series*, vol. 2129, no. 1, p. 12028–12029, Dec. 2021, doi: 10.1088/1742-6596/2129/1/012028.

[28] S. Fortunato and M. Barthélemy, “Resolution Limit in Community Detection,” *Proceedings of the National Academy of Sciences*, vol. 104, no. 1, pp. 36–41, Jan. 2007, doi: 10.1073/pnas.0605965104.

[29] V. A. Traag, P. Van Dooren, and Y. Nesterov, “Narrow scope for resolution-limit-free community detection,” *Physical Review E*, vol. 84, no. 1, Jul. 2011, doi: 10.1103/physreve.84.016114.

[30] L. Carvalho, A. Garcia, W. K. G. Assunção, R. De Mello, and M. Julia De Lima, “Analysis of the Criteria Adopted in Industry to Extract Microservices,” in *2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*, Montreal, QC, Canada: IEEE, May 2019, pp. 22–29. doi: 10.1109/CESSER-IP.2019.00012.

[31] “ISO/IEC/IEEE International Standard - Systems and software engineering: Vocabulary,” *ISO/IEC/IEEE 24765:2017(E)*, vol. 0, no. , pp. 1–541, 2017, doi: 10.1109/IEEESTD.2017.8016712.

[32] J. Fitzpatrick, “Applying the ABC Metric to C, C++, and Java,” 1997.

[33] S. Chidamber and C. Kemerer, “A Metrics Suite for Object Oriented Design,” *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994, doi: 10.1109/32.295895.

[34] T. Lopes and A. R. Silva, “Monolith Microservices Identification: Towards An Extensible Multiple Strategy Tool,” in *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*, L'Aquila, Italy: IEEE, Mar. 2023, pp. 111–115. doi: 10.1109/ICSA-C57050.2023.00034.

[35] M. Abdellatif *et al.*, “A Taxonomy of Service Identification Approaches for Legacy Software Systems Modernization.” p. 110868–110869, 2021.

[36] P. J. Rousseeuw, “Silhouettes: A graphical aid to the interpretation and validation of cluster analysis,” *Journal of Computational and Applied Mathematics*, vol. 20, pp. 53–65, 1987, doi: https://doi.org/10.1016/0377-0427(87)90125-7.

[37] L. Briand, S. Morasca, and V. Basili, “Property-Based Software Engineering Measurement,” *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 68–86, 1996, doi: 10.1109/32.481535.

A Systematic Literature Review (SLR) publications

Primary studies

ID	Publication
[**P1**]	M. Saidi, A. Tissaoui and S. Faiz, *A DDD Approach Towards Automatic Migration To Microservices*, 2023
[**P2**]	Z. Yang, S. Wu and C. Zhang, *A Microservices Identification Approach Based on Problem Frames*, 2022
[**P3**]	T. Kinoshita and H. Kanuka, *Automated Microservice Decomposition Method as Multi-Objective Optimization*, 2022
[**P4**]	X. Zhou and J. Xiong, *Automated Microservice Identification from Design Model*, 2022
[**P5**]	Y. Zhang et al., *Automated Microservice Identification in Legacy Systems with Functional and Non-Functional Metrics*, 2020
[**P6**]	G. Quattrocchi et al., *Cromlech: Semi-Automated Monolith Decomposition Into Microservices*, 2024
[**P7**]	M. Kamimura et al., *Extracting Candidates of Microservices from Monolithic Application Code*, 2018
[**P8**]	O. Al-Debagy and P. Martinek, *Extracting Microservices' Candidates from Monolithic Applications: Interface Analysis and Evaluation Metrics Approach*, 2020
[**P9**]	G. Mazlami, J. Cito and P. Leitner, *Extraction of Microservices from Monolithic Software Architectures*, 2017
[**P10**]	A. Selmadji et al., *From Monolithic Architecture Style to Microservice One Based on a Semi-Automatic Approach*, 2020
[**P11**]	G. Filippone et al., *From Monolithic to Microservice Architecture: An Automated Approach Based on Graph Clustering and Combinatorial Optimization*, 2023

[P12]	S. Wu and C. Zhang, *Identification of Microservices through Processed Dynamic Traces and Static Calls*, 2022
[P13]	P. Zaragoza et al., *Leveraging the Layered Architecture for Microservice Recovery*, 2022
[P14]	S. Santos and A. Silva, *Microservices Identification in Monolith Systems: Functionality Redesign Complexity and Evaluation of Similarity Measures*, 2022
[P15]	S. Ma, T. Lu and C. Li, *Migrating Monoliths to Microservices Based on the Analysis of Database Access Requests*, 2022
[P16]	G. Filippone et al., *Migration of Monoliths through the Synthesis of Microservices Using Combinatorial Optimization*, 2021
[P17]	J. Lourenço and A. Silva, *Monolith Development History for Microservices Identification: A Comparative Analysis*, 2023
[P18]	S. Agarwal et al., *Monolith to Microservice Candidates Using Business Functionality Inference*, 2021
[P19]	M. Amiri, *Object-Aware Identification of Microservices*, 2018
[P20]	L. Carvalho et al., *On the Performance and Adoption of Search-Based Microservice Identification with toMicroservices*, 2020
[P21]	J. Hao, J. Zhao and Y. Li, *Research on Decompostion Method of Relational Database Oriented to Microservice Refactoring*, 2023
[P22]	Y. Li et al., *RM2MS: A Tool for Automatic Identification of Microservices from Requirements Models*, 2023
[P23]	W. Jin et al., *Service Candidate Identification from Monolithic Systems Based on Execution Traces*, 2021
[P24]	F. Eyitemi and S. Reiff-Marganiec, *System Decomposition to Optimize Functionality Distribution in Microservices with Rule Based Approach*, 2020
[P25]	M. Daoud et al., *Towards an Automatic Identification of Microservices from Business Processes*, 2020
[P26]	Y. Romani, O. Tibermacine and C. Tibermacine, *Towards Migrating Legacy Software Systems to Microservice-based Architectures: A Data-Centric Process for Microservice Identification*, 2022
[P27]	D. Escobar et al., *Towards the Understanding and Evolution of Monolithic Applications as Microservices*, 2016
[P28]	C. Bandara and I. Perera, *Transforming Monolithic Systems to Microservices - An Analysis Toolkit for Legacy Code Evaluation*, 2020
[P29]	M. Brito, J. Cunha and J. Saraiva, *Identification of Microservices from Monolithic Applications through Topic Modelling*, 2021
[P30]	Y. Wei et al., *A Feature Table Approach to Decomposing Monolithic Applications into Microservices*, 2020

[**P31**]	K. Sellami, M. Saied and A. Ouni, *A Hierarchical DBSCAN Method for Extracting Microservices from Monolithic Applications*, 2022
[**P32**]	M. Hasan et al., *AI-based Quality-driven Decomposition Tool for Monolith to Microservice Migration*, 2023
[**P33**]	V. Nitin et al., *CARGO: AI-Guided Dependency Analysis for Migrating Monolithic Applications to Microservices Architecture*, 2022
[**P34**]	L. Cao and C. Zhang, *Implementation of Domain-oriented Microservices Decomposition Based on Node-attributed Network*, 2022
[**P35**]	A. Santos and H. Paula, *Microservice Decomposition and Evaluation Using Dependency Graph and Silhouette Coefficient*, 2021
[**P36**]	A. Kalia et al., *Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices*, 2021
[**P37**]	S. Eski and F. Buzluca, *An Automatic Extraction Approach: Transition to Microservices Architecture from Monolithic Application*, 2018
[**P38**]	J. Wu, A. Hassan and R. Holt, *Comparison of Clustering Algorithms in the Context of Software Evolution*, 2005
[**P39**]	L. Baresi, M. Garriga and A. De Renzis, *Microservices Identification Through Interface Analysis*, 2017
[**P40**]	A. Kalia et al., *Mono2Micro: An AI-based Toolchain for Evolving Monolithic Enterprise Applications to a Microservice Architecture*, 2020

Table 9: Selected publications (primary studies)

Secondary studies

ID	Publication
[**P41**]	D. Bajaj et al., *A Prescriptive Model for Migration to Microservices Based on SDLC Artifacts*, 2021
[**P42**]	Y. Abgaz et al., *Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review*, 2023
[**P43**]	I. Oumoussa and R. Saidi, *Evolution of Microservices Identification in Monolith Decomposition: A Systematic Review*, 2024
[**P44**]	R. Schmidt and M. Thiry, *Microservices Identification Strategies : A Review Focused on Model-Driven Engineering and Domain Driven Design Approaches*, 2020
[**P45**]	J. Kazanavicius and D. Mazeika, *Migrating Legacy Software to Microservices Architecture*, 2019
[**P46**]	A. Mparmpoutis and G. Kakarontzas, *Using Database Schemas of Legacy Applications for Microservices Identification: A Mapping Study*, 2022
[**P47**]	J. Fritzsch et al., *From Monolith to Microservices: A Classification of Refactoring Approaches*, 2019

Table 10: Selected publications (secondary studies)