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, 98 p., 4 app. p.
Software Engineering
April 2024

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

This thesis investigates the modular monolith architecture and its benefits for monolithic applications, as well as the challenges faced during the migration process. It explores the concept of automated modularization, and the techniques for identification of module boundaries in software systems. Based on a literature review, a four-step approach to automated modularization is designed, and a case study is conducted to evaluate its effectiveness using the coupling and cohesion quality metrics. The results of the case study indicate that the automated approach is effective in identifying module boundaries, and can be used to modularize monolithic applications. Reflecting upon 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

API	Application Programming Interface
AST	Abstract Syntax Tree
BPMN	Business Process Model and Notation
DDD	Domain-Driven Design
DSRM	Design Science Research Methodology
DSRP	Design Science Research Process
SDLC	Software Development Life Cycle
SLOC	Source Lines of Code
SLR	Systematic Literature Review
SOA	Service-Oriented Architecture

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 monolith 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 monolith and microservices architectures. The modular monolith architecture aims to combine the flexibility of using a microservices architecture with the simplicity of a monolith codebase. The modular monolith 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 monolith software systems to a more flexible and scalable architecture.

In this thesis, we aim to investigate the potential benefits of a modular monolith architecture, and how automated technologies can help software architects to migrate their monolith codebases to a modular monolith or microservices 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 monolith 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 is the modular monolith architecture, and what sets it apart from monolith 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 merits and demerits of the modular monolith architecture with a particular focus on its application to migrating existing monolith codebases. To answer this question, we will first define the modular monolith architecture using existing literature, and examine what sets it apart from monolith and microservices architectures. Then, we will proceed to investigate the merits and demerits 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 monolith codebases in the literature.

Finally, the third research question is motivated by the fact that a lot of useful information is embedded in the source code repository of a software system, and how this information can be extracted to aid in the modularization process. An approach for microservice candidate identification will be designed 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 demerits of the modular monolith architecture
2. Investigate the use of automated technologies to modularize a monolith architecture

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

1.3 Outline

The thesis is structured as follows.

In Chapter 1, the motivation behind the research is explained. The scope and goal are defined, and the research questions are formulated. Chapter 2 describes the research methodology used in this thesis. In Chapter 3 the reader is introduced to the research background and necessary concepts. Chapter 4 discusses the related work in the field of the research topic.

Chapter 5 is dedicated to answering the first research question. The modular monolith architecture is defined, and its merits and demerits are discussed. The chapter also touches the concept of modularization and its relevance in software architecture.

The next chapters aim to solve the remaining research questions. Chapter 6 takes a deep dive into the existing technologies for automated modularization. Based on this study, Chapter 7 then continues with a proposed solution for automated modularization. In Chapter 8, a case study

is presented that applies the proposed solution to a real-world software system. The results are then evaluated and discussed.

Finally, Chapter 9 summarizes the findings, discusses the validity of the research, 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 of grey literature is conducted, picking a select number of publications that define and discuss the modular monolith architecture. An ad hoc review is a less formal review process, where the researcher discusses purposefully selected publications to gain an understanding of a specific topic [4].

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 [5]. 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 [4]. 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 Kitchenham and Charters [5]:

	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 [1]. 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.

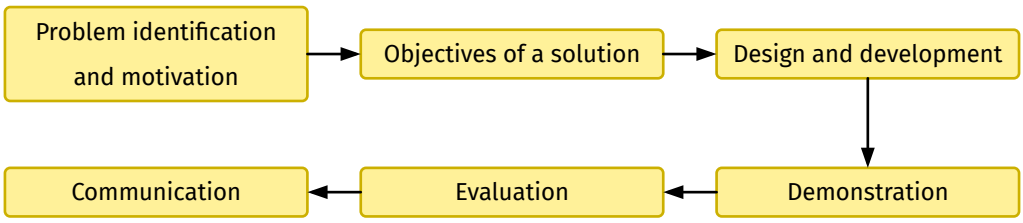


Figure 1: Design Science Research Process [1]

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.

Domain-driven design

Domain-Driven Design (DDD) is a software design approach that focuses on modeling business requirements and domain concepts in the design [6]. The goal of DDD is to align the concepts in the software model with the concepts in the business domain, to allow better understanding and knowledge transfer between domain experts and developers. For example, a e-commerce application may have domain concepts such as “Shopping cart” and “Order”, which can be directly translated to software entities and classes.

Monolith architecture

In software engineering, a monolith application describes a software system that consists of a single, indivisible unit [7]. Monolith applications encompass all the components of the system, including the presentation layer, the business logic layer, and the data access layer [8]. Software components within a monolith are interdependent and tightly coupled, which makes development and maintenance more challenging as the system grows in size and complexity. Changes and updates to the source code affect the entire system, and require redeployment of the entire application. Monolith

architectures are typically more prone to failure, as they have a single point of failure, and are less scalable than distributed systems.

Modular programming

Modular programming is a software engineering technique that emphasizes the separation of concerns by dividing a software system into smaller, independent modules [9]. Each module is responsible for a specific functionality of the system, and exposes a well-defined interface for communication with other modules. Changes to one module do not affect other modules, which makes the system more maintainable and allows multiple teams to work on different modules simultaneously. Modular programming does not make any assumptions about the underlying architecture of the system, and can be applied to both monolith and distributed systems.

Service-oriented Architecture

Service-Oriented Architecture (SOA) is a software architectural style that focuses on service orientation [10]. Service orientation is a design paradigm that encourages thinking about a problem domain in terms of services and service-based development. A service is a logical software unit that represents a business activity with a specified outcome, is self-contained, and its inner workings are opaque to the consumer of the services. Domain knowledge and heuristics may be used to define services.

SOA is an architectural style, and is not tied to any specific technology or implementation. Services can be implemented as standalone applications, web services, or microservice components. Following the principles of SOA, services can be implemented in the same or in different programming languages, and can execute on the same or on different platforms.

Microservice architecture

Microservices are a more recent architectural style that builds on the principles of SOA and modular programming [11]. In a microservice architecture, a software system is decomposed into a set of small, independent deployment units that communicate with each other using lightweight protocols such as HTTP or messaging queues. Microservices promote the separation of concerns and decentralized governance [11]. Each microservice is a separate codebase, which makes it easier to develop and deploy the system incrementally. The architecture is designed to be resilient to failures, and allows for scaling individual microservices independently. In modern software engineering, microservices are a key concept in building flexible, scalable, and maintainable software systems.

While SOA and microservices share many similarities, there are some key differences between the two architectural styles. SOA can still be developed as a monolith application, while microservices are inherently distributed [12]. Microservices architecture puts more emphasis on the autonomy of development, allowing teams to make independent decisions about technology stack and implementation details. SOA is heavily focused on the reuse of code and the abstraction of business functionality, while microservices are more focused on bounded contexts and not sharing code between services [13].

Modularization

Modularization, or microservice decomposition, is the process of dividing a monolith application into smaller, loosely coupled modules or microservices [14]. The goal of modularization is to decrease the complexity and maintenance burden of tightly coupled, interconnected software components. Modularization can be an objective in the design of new software systems,

or as an objective in the migration of monolith applications to a distributed architecture. Cohesion and coupling are two key concepts in modularization [15]. Modules in a modularized system are cohesive, meaning that they only encapsulate code that is related to a specific functionality or domain concept. Modules are loosely coupled, which entails that they have minimal dependencies on other modules, and only communicate through the defined interface contracts.

Modularization typically requires a deep understanding of the existing system architecture, bundled with knowledge about the business domain and the requirements of the system. The process of modularization can be very time-consuming and error-prone [16]. Automated tools and techniques for microservice identification and decomposition can help to accelerate the process, and allow software architects with limited knowledge about the software system to initiate the modularization process.

Quality metrics

Quality metrics are quantitative measures that provide insight into the quality of software systems [17]. In the context of microservices, quality metrics can be used to evaluate how modular the system is, and identify areas for improvement. Many different types of quality metrics exist, including size, complexity, and coupling metrics [16]. Microservices are typically evaluated based on the cohesion and coupling of the individual services.

4 Related work

In this chapter, we present related work on the topic of automated microservice identification in monolith systems, with a focus on concrete implementations of these methods and tools. A number of tools have been developed to aid the modularization of monolith applications into microservices or modular monolith architecture.

Gysel et al. [18] developed a tool called ServiceCutter. The tool, implemented as a web application[1], extracts information from the domain model of the application, and uses it to identify microservice candidates. The software architect can customize the process by changing the algorithm and several parameters used for the identification. ServiceCutter implements a visual approach to the identification of microservices, and provides a starting point for the modularization of monolith applications.

Pereira da Rocha [19] proposed a monolith decomposition tool called Monolysis, and introduces the MonoBreak algorithm used in the tool to identify microservice candidates based on the functionalities of the application. The tool collects execution trace data from the monolith application, and uses similarity analysis to group functionalities that are likely to be part of the same microservice. The software architect can specify the granularity of the microservices, which provides a level of customization.

[1] https://servicecutter.github.io/

In 2019, Saidani et al. introduced MSExtractor, an automated tool to extract microservices from legacy applications written in the Java programming language[2] [P41]. Using a genetic algorithm, the authors demonstrate the ability of the tool to outperform other state-of-the-art approaches in terms of the quality of the decomposition.

Mono2Micro is an AI-based toolchain for the decomposition of monolith applications into microservices, developed by Kalia et al. [P40]. The toolchain uses a combination of static and dynamic analysis to identify microservice candidates, and uses machine learning to generate recommendations for the decomposition. Mono2Micro has the ability to analyze monolith applications written in Java and provides a visual interface to interact with the tool.

In 2023, Petrovic and Sawhney introduced ServiceWeaver [20], a framework for the Go programming language[3] that aims to introduce the concept of a modular monolith architecture. Applications written in Go are compiled into a single, statically linked binary, which is then deployed as a single unit. ServiceWeaver maintains the same development process as a monolith application, but the application is deployed as a set of microservices that communicate with each other over the network. The framework leverages the Go runtime to modularize the application at runtime, by using information about the structure and dependencies of the application. It also provides integration with cloud providers, and a set of tools to monitor and manage the deployed microservices.

The Structural Quality (S-Quality) framework, introduced by Hasan et al. [P32], is a tool that uses static analysis to identify microservice candidates in monolith applications. The tool uses structural design properties, alongside customizable architectural quality objectives, to generate recommendations

[2]https://www.java.com
[3]https://go.dev

for the decomposition of the monolith application. The S-Quality application was developed using the Django[4] framework.

Lopes and Silva [21] implemented an extensible multiple strategy tool for the identification of microservice candidates in monolith applications. The application implements multiple strategies for identification in order to promote the comparison of modularization approaches. The authors wrote the tool in multiple programming languages, exposing a web application interface to interact with the tool.

In summary, several tools and frameworks have already been developed to aid the modularization of monolith applications into microservices or modular monolith architecture. These tools use a variety of approaches, and usually only support a specific programming language or framework. The level of automation varies between the tools, with some only providing recommendations or visualizations of the microservice candidates, and others implementing fully automated solutions that can deploy the microservices.

[4]https://www.djangoproject.com/

5 Modular monolith architecture

In this chapter, we will discuss the modular monolith architecture. We will start by defining the architectural style, and then continue by discussing the merits and demerits of this architecture. We aim to answer the following research question:

Research Question 1: What is the modular monolith architecture, and what sets it apart from monolith and microservices architectures?

5.1 Definition

While a traditional monolith architecture is a single-tiered software architecture that tightly couples the three layers (presentation, business logic and data access), the modular monolith architecture focuses on separation of concerns by partitioning the application into modules or components based on their functionality [3].

The three layers are present in each module, but they are not directly accessible from outside of the module. Instead, modules expose a well-defined interface that describes the capabilities and limitations of the module. Hence, the modules of a modular monolith are loosely coupled.

The architecture emphasizes interchangeability and potential reuse of modules, while maintaining an explicit interface between them [22]. Focusing on business domains rather then technical capabilities improves the organization of the code, and increases comprehensibility.

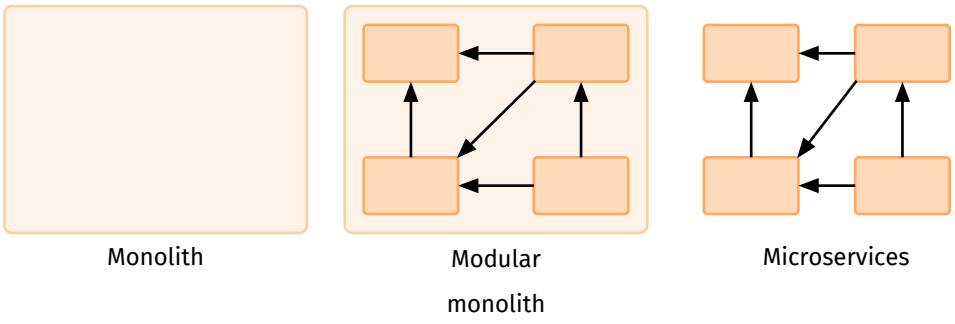


Figure 2: Modular monolith architecture [2]

While traditional monolith architecture is not a bad choice for small applications, it becomes difficult to develop new functionality and maintain existing as the application grows. Larger applications are likely to turn into a big ball of mud, where the code is tangled and difficult to understand. A big ball of mud, a term coined by Foote and Yoder in 1997, is a software system that lacks a perceivable architecture [23].

Modular monolith architecture can also be used as a stepping stone towards a microservices architecture. Once the application is modularized and the interfaces are well-defined and stable, individual modules can be extracted and turned into external microservices. If the agreed upon interface is respected, the external microservice can be swapped out entirely by another implementation without affecting the rest of the application. When

all modules have been extracted from the monolith, the application has effectively been transformed into a microservices architecture.

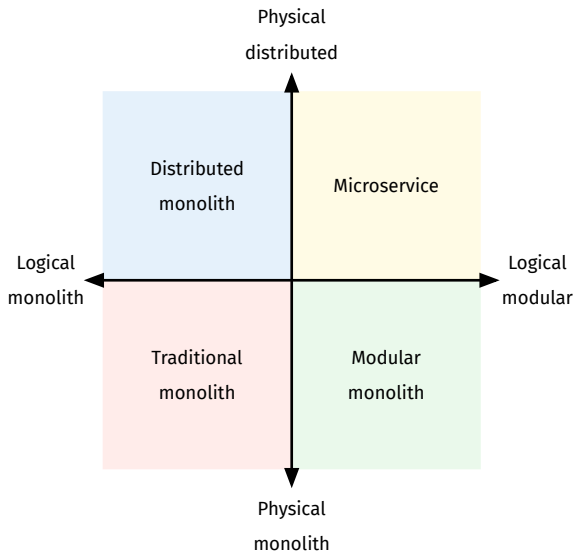


Figure 3: Physical and logical architectures [3]

Contrary to microservices, modular monolith applications are built as a single deployable unit. The modules of the software system are separated logically (not physically), and are deployed together. Horizontal scaling of a modular monolith architecture is more difficult than in a microservices architecture.

Su and Li [22] identified six characteristics of modular monolith architecture in the literature:

1. **Segregation** of modules: each module is independent and includes all three application layers. Modules are autonomously developed, tested and deployed.
2. **Modularity**: modules are highly internally cohesive and loosely externally coupled. Communication between modules is done using well-defined interfaces, preferably asynchronously.
3. **Unified database**: the database schema is shared by all modules, in contrast with microservices where each microservice has its own database and schema.

4. **Monolith deployment**: the application is deployed as a single unit, and although modules can be distributed across multiple hosts.
5. **Unified application process**: the application functions as a singular process, scaling uniformly depending on the requirements.
6. **Maintainability and scalability**: the architecture can efficiently manage increasing complexity, and facilitates growth.

In summary, the modular monolith architecture aims to find a middle ground between the monolith and microservices architectures by reaping the benefits of both approaches. While opting for a modular monolith architecture already improves flexibility and comprehensibility, it can also be used as a step in the migration towards a microservices architecture.

5.2 Merits and demerits

In the 2013 book "Software architecture for developers", Brown defined architectural drivers as a set of requirements that have significant influence over software architecture [24]. The author argues that architectural drivers are the most important requirements that shape the architecture of a software system. Architectural drivers are often classified in four categories: functional requirements, quality attributes, technical, and business constraints [25]. Any realized software architecture is a trade-off between several architectural drivers. Hence, the choice of architecture depends on the context it is being designed in.

In this section, we qualitatively compare the architectural drivers of modular monolith architecture with traditional monolith and microservices architectures. Table 2 provides an overview of the comparison. A star rating system is used to indicate the performance of each architecture with respect to the architectural driver, with one star indicating the lowest performance,

and three stars indicating the highest performance. The ratings are based on several sources, including Grzybek [25], Fowler [26], Kodja [27], Küçükoğlu [3], and Su and Li [22].

Architectural driver	**Monolith**	**Modular monolith**	**Microservices**
Complexity	★★★	★★	★
Structure	★★	★★	★
Testing	★	★★	★★
Productivity	★★★	★★★	★
Performance	★	★★	★★★
Fault tolerance	★	★★★	★★★

Table 2: Comparison of modular monolith architecture

Complexity

The complexity of a software system is related to the number of modules and their interactions [25]. In a monolith architecture, there is only one module (the monolith application itself), and no interactions. This makes the architecture simple, and easy to deploy. Microservices architecture improves the coupling by separating modules into independent microservices, but introduces complexity due to the communication between the microservices, and the additional tools and infrastructure required to manage them. The modular monolith architecture uses the decoupled approach of microservices, but keeps the complexity down by bundling the modules together in a single deployable unit. Modules in a modular monolith architecture have two ways of communicating: externally through Application Programming

Interface (API) calls, and internally through abstracted interfaces [22]. The external API calls can introduce additional complexity, although some tools (e.g. ServiceWeaver) hide this complexity behind abstractions of internal communication.

Structure

The source code of monolith and modular monolith applications is stored in the same repository. This makes enacting changes to the codebase easier, as developers can modify multiple modules or layers at the same time. In contract, microservices are usually stored in separate repositories, due to the independent nature of the services [3].

In 1968, Conway observed that the structure of a software system is often influenced by the communication structure of the organization that develops it [28]. The modular nature of modular monolith and microservices architectures makes it easier to align the architecture with the organization structure, while the inherent structure of a monolith architecture does not provide this ability.

Testing

Testing a monolith application can be difficult and tedious, as code is tightly coupled and changes in one module can affect other modules [22]. Functionality cannot be tested independently, but must be tested in the context of the entire application. In modular monolith and microservices architectures, functionality is separated into modules or microservices, which can be tested in isolation. To ensure that the application works as expected, integration tests can be performed that test the integration between the modules or microservices.

Productivity

When designing a software system from scratch, a monolith architecture is often the most productive choice [26]. Developers can focus on the business logic of the application, without having to worry about the underlying infrastructure or deployment model. Using microservices architecture incurs additional complexity, as developers must define the interfaces between the microservices, and manage the communication between them. Modular monolith architecture compromises between the two, by allowing developers to focus on the business logic first, while still providing the benefits of a distributed architecture. The "Monolith First" approach proposed by Fowler suggests initiating new software systems with a monolith architecture, and only moving to a microservices approach when the monolith architecture shows its limitations [26].

Performance

Traditional monolith applications offer better performance than distributed architectures, due to the lower overhead incurred when processing requests [25]. However, monolith applications can only handle a limited number of requests until the application becomes a bottleneck. Distributed architectures, such as microservices and modular monoliths, can offer a solution for this problem by allowing the application to scale horizontally with ease, at the cost of increased complexity and overhead. Care must be taken when designing the distributed application to ensure that the transactional context of a request is not spread across multiple modules or microservices, as this can lead to performance issues.

Additionally, a microservices architecture is truly distributed in that every microservice is bundled with its own external dependencies, such

as databases and caches. This allows for better performance, as the dependencies can be scaled alongside the microservices [27].

Fault tolerance

The impact of a failure in a monolith application is greater than in modular architectures, as the entire application is affected [25]. The risk of a system failure can be mitigated by replicating the application, but this comes with a significant cost. Distributed architectures are more tolerant to individual failures, as the failure of one module or microservice does not affect the entire application, and can be resolved quicker [29]. Communication between modules or microservices is done over a network, where failures and delays are expected. Hence, distributed architectures are designed to be fault-tolerant, and can handle failures more gracefully than monolith applications.

In summary, the modular monolith architecture strikes a balance between the monolith and microservices architectures. It offers the simplicity of a monolith architecture, while providing the flexibility and scalability of a distributed architecture. In particular, the modular monolith architecture is well-suited for smaller developments teams where the domain complexity of the application is manageable [27]. On the other hand, when the requirements for scaling become more stringent, or the use of multiple, heterogeneous technologies is required, a microservices architecture might be a better choice [27].

5.3 Modularization

Modularization of monolith applications is not a new concept. In 1972, Parnas stated that modularization is a mechanism that can improve the flexibility and comprehensibility of software systems [14]. The author argued

that modularization separates the system into smaller and more manageable parts, which can be developed concurrently by different teams.

The process of modularization involves identifying the potential modules or microservices of the software system, defining the interfaces between them, and evaluating the impact of the modularization on the system.

Modularization techniques can be classified into three categories [30]:

- **Top-down (forward) engineering**: starting from the domain artifacts of the application (e.g. requirements, use cases), and deriving the modules from them
- **Bottom-up (reverse) engineering**: mining functionalities from the existing system (source code, documentation) and developing the modules based on this structure
- **Hybrid/reuse approach**: mapping domain artifacts onto existing functionalities in the system, and identifying the modules based on this mapping

Top-down approaches are more suitable for new projects, where the requirements are known in advance and little or no existing code is available. For migration of legacy applications to a modular architecture, bottom-up approaches are more suitable, as they can leverage the existing codebase and domain artifacts.

Abdellatif et al. [30] identified three main steps in the process of modularization of monolith applications:

1. **Collection**: extraction of useful information from the monolith application
2. **Decomposition**: partitioning the application into individual modules or microservices

3. **Analysis**: evaluating the impact of the modularization on the system

These steps can be done manually by the software architect, or automatically using tools that collect information about the application, and decompose it into a modular application. Manual collection of data is feasible when the application is small, but becomes unrealistic as the application grows. Automated tools can also help to extract implicit information from the application, such as the relationship between contributors and file changes, or the frequency of changes to a file. Automated decomposition tools can help to identify the modules of the application, although for the most part they cannot rely on the knowledge that the software architect has about the application, rendering them less effective than manual decomposition.

The process of identifying modules or microservices for a modular monolith application is similar to the process of identifying them for a microservices architecture [2]. However, the criteria for identifying modules are different, as the modules of a modular monolith application are not deployed as separate microservices.

In light of the above, we see a clear need for automated tools to aid the modularization of monolith applications, and a gap in the market for automated microservice candidate identification.

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 [31], [32], 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[5]
2. ACM Digital Library[6]

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*

[5]https://ieeexplore.ieee.org/
[6]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 3, 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[7] or brownfield[8] development of systems using microservices architecture • Publications published before 2014, as the definition of "microservices" as an architectural style is inconsistent before 2014 [32] • Publications addressing manual methods or technologies • Surveys, opinion pieces, or other non-technical publications

Table 3: 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.

[7]Development of new software systems lacking constraints imposed by prior work [33]
[8]Development of new software systems in the presence of legacy software systems [33]

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 4: 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.

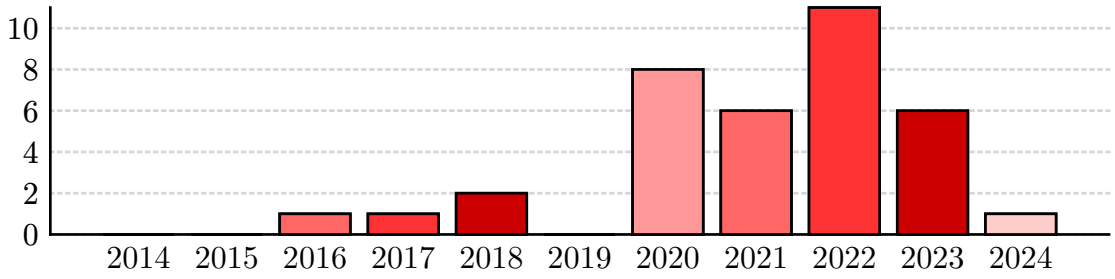


Figure 4: Distribution of publications by year

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

Kitchenham and Charters [5] 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. [P42]. 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 the work of Abdellatif et al. [30], who identified six types of microservice 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 5. The categories are based on Bajaj et al. [P42].

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

Table 5: 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).

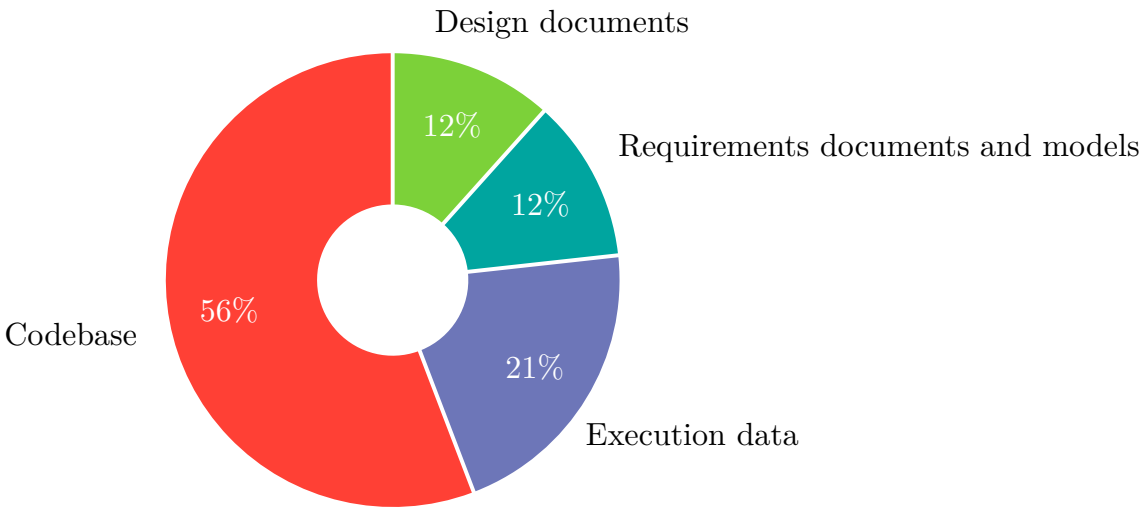


Figure 5: SDLC artifact categories

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 [34]. 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] modeled 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] tackled requirements engineering using problem frames [35]. 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] employed 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] proposed a data-driven method based on the analysis of the external API exposed by the application, specified in the OpenAPI[9] format. The method extracts the information

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

from the specification and converts it into vector representation for further processing.

Zhou and Xiong [P4] used 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] took 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] used 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), 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 software components. 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] used the source code of the software system to construct an Abstract Syntax Tree (AST), and mapped the dependencies between the business and data layer. Kamimura et al. [P7] used a more data-driven approach, and statically traced 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] mapped object-oriented classes in the source code to specific microservices, but required a list of microservices to be specified before the decomposition is performed.

Filippone et al. [P16] concentrated 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] made a distinction between presentation, business, and data layer.

Most of the publications tracing dependencies between software components do this at the level of the component. 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] did not automatically extract information from the source code, but relied on a software architect to decompose the software system on the basis of business capability.

Romani et al. [P26] proposed 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] constructed 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 made a distinction between context-insensitive and context-sensitive dependency graphs. While the former captures the dependencies between software components 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 that used 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] collected the execution traces of the software system, in combination with performance logs.

Ma et al. [P15] used 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] collected 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] used 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 vertices 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 6.

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

Table 6: 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%).

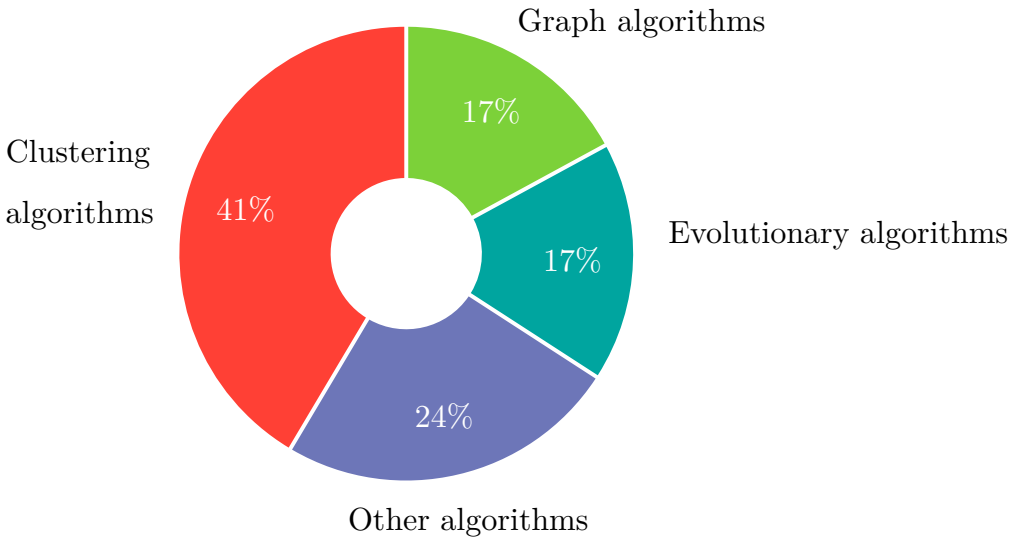


Figure 6: SDLC algorithm categories

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 that used 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] used Affinity Propagation [36] 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] used 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] extended the hierarchical agglomerative clustering (HAC) algorithm [37] 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] proposed two possible algorithms for microservice identification: a hierarchical clustering algorithm, and a clustering algorithm based on gravity centers.

Sellami et al. [P31] used the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm [38] 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 that used 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] used the next generation of NSGA, NSGA-III, in order to find a solution for the problem.

Amiri [P19] relied on a genetic algorithm using Turbo-MQ [39] 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] used Kruskal's algorithm [40] 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] applied the Louvain community detection algorithm [41] to obtain the granularity of the microservices, and high-cohesive communities of vertices. The Louvain algorithm 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] used the Leiden algorithm [42], an improvement of the Louvain algorithm that uses a refinement step to improve the quality of the communities.

Cao and Zhang [P34] used 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] used Context sensitive Label Propagation (CARGO), an algorithm built on the principles of the Label Propagation algorithm [43]. 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] used 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 [44], that aims at identifying software subsystems without the need for human intervention. Escobar et al. [P27] also used a custom clustering algorithm, detecting optimal microservices based on a meta-model of the class hierarchy.

Agarwal et al. [P18] proposed 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] used 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 monolith system into microservices.

6.3.3 Metrics

The quality metrics used in the publications are summarized in Table 7. 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	[P4][P5][P6][P8][P10][P11][P12][P13][P14] [P16][P17][P20][P22][P23][P25][P28][P29] [P30][P31][P32][P33][P34][P36][P40]
Coupling	[P4][P5][P10][P11][P12][P14][P16][P17] [P18][P20][P22][P23][P25][P28][P29][P30] [P31][P32][P33][P34][P36][P40]
Network overhead	[P4][P6][P16][P20]
Complexity	[P8][P14][P17][P32]
CPU and memory usage	[P5][P6][P33]
Modularity	[P12][P22][P23][P28][P29][P31][P36][P40]
Other metrics	[P1][P2][P7][P9][P10][P11][P15][P17][P21] [P22][P30][P31][P32][P33][P34][P35][P36] [P40]
No metrics	[P3][P24][P26][P27]

Table 7: 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%.

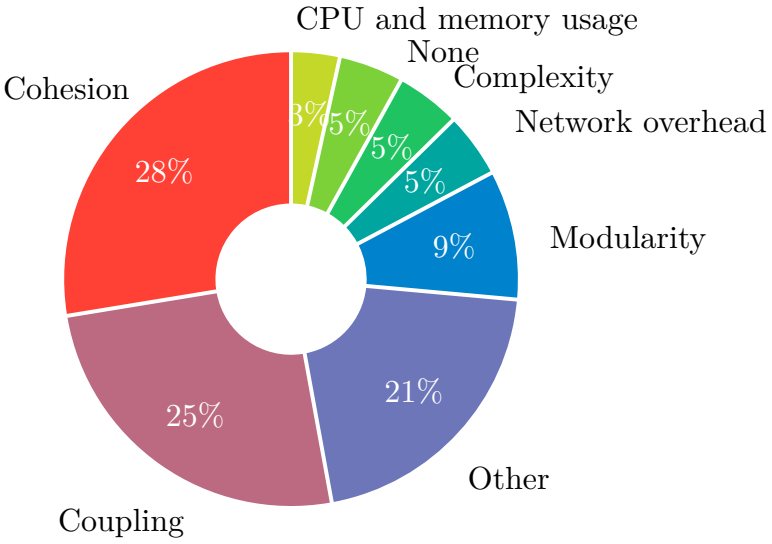


Figure 7: SDLC metric categories

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 [14]. As object-oriented programming became more popular, the concepts of cohesion and coupling were adapted to the new paradigm [45].

Throughout the years, many definitions of cohesion and coupling have been proposed both for procedural and object-oriented systems. For example, Briand et al. [46] defined 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] defined (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 microservice boundary. The total coupling of the solution is the sum of the coupling of all microservices. Similarly, Filippone et al. [P11] defined 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] introduced the concept of inter-service cohesion and inter-call percentage (ICP) as coupling metrics. Several other publications used the metrics introduced by Jin et al. [P23] 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 defined 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 microservice 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] used the inverse of cohesion as a metric, named lack of cohesion (LCOM). Lack of cohesion 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 microservice are related to each other in terms of functionality.

Network overhead

Microservices are distributed systems, and communication between microservices 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] calculated 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 included the protocol overhead in the calculation, which is the cost of the communication protocol used to send messages between microservices (for example, TCP headers, HTTP headers, etc.).

Quattrocchi et al. [P6] measured 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] defined 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] defined 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] defined complexity as the effort required to perform the decomposition, and expanded 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] used 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] defined operational costs as metric to minimize, which includes communication (network) and data management (CPU and memory) costs.

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

Modularity

Modularity is a measure of independence of microservices, and can be divided into many dimensions, such as structure, concept, history, and dynamism

[47]. Some definitions of modularity rely on the concepts of cohesion and coupling, and the balance between them.

Jin et al. [P23] used modularity as a metric to evaluate potential decompositions. The authors use Modularity Quality [16] and extend the concept with structural and conceptual dependencies to assess the modularity of microservice candidates.

Carvalho et al. [P20] introduced 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] introduced 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 fewer than 1 indicates a reduction in the number of authors. Mazlami et al. [P9] made 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] proposed 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] used 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] talked about data autonomy determined by the internal and external data access of a microservice candidate.

Kamimura et al. [P7] introduced a metric called dedication score, which measures the relationships between microservices as a function of access frequency. Along with a modularity metric, the dedication score is used in their custom SArF dependency-based clustering algorithm [44].

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] used 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] used 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] used 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] used the silhouette coefficient originally proposed by Rousseeuw [48] 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 judgement of the software architect to choose a qualitative decomposition.

7 Proposed solution

In this chapter, we propose **Modular Optimization to Service-oriented Architecture Identification Kit (MOSAIK)**, our solution for identification of microservice candidates in a monolith 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 monolith 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 software components in the monolith application, and the edges represent the dependencies between them. The input of the algorithm is a representation M of the monolith 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.

$$M_{F_i} = \{f_1, f_2, ..., f_j\}$$
$$M_{C_i} = \{c_1, c_2, ..., c_k\} \quad (2)$$
$$M_{H_i} = \{h_1, h_2, ..., h_l\}$$

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.

$$S_i = \{s_1, s_2, ..., s_m\} \quad (3)$$

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.

$$f : M_{C_i} \to S_i \quad (4)$$

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 monolith 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 Lopes and Silva [21].

- **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 8. 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.

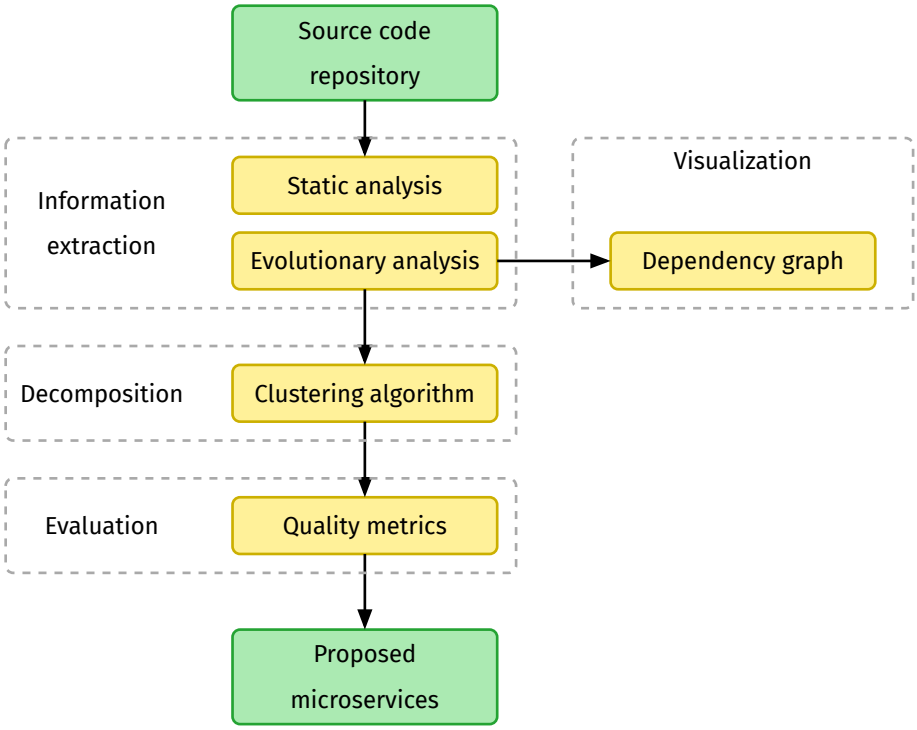


Figure 8: MOSAIK architecture overview

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 in the Ruby programming language[10]. The source code of the implementation is available on Github[11].

[10]https://ruby-lang.org
[11]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 [15]. Non-functional requirements are additional requirements imposed at design-time that do not directly impact the functionality of the application [15].

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

1. **Quality**: the solution provides a high-quality decomposition of the monolith application, based on a set of quality metrics
2. **Automation**: the solution automates the decomposition process as much as possible
3. **Visual**: the solution can output the proposed decomposition in a visual manner, to aid understanding of the process and the results

The non-functional requirements identified for our solution are:

1. **Performance**: the solution performs the analysis, decomposition, and evaluation reasonably fast

7.4 Extraction

Software development is typically done in multiple steps, either using the waterfall model, or using an iterative approach [49]. 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.

Mazlami et al. [P9] proposed 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 monolith 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 monolith application using the `parser` library[12]. 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 monolith application, MOSAIK extracts the references between classes.

Using this information, a call graph is constructed that represents the structural coupling of the monolith application. For each class in the monolith 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.

[12]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 Carvalho et al. [P20] noted, 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 monolith 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 monolith application. The algorithm iterates over all classes in the monolith 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 [50], 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 monolith 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 [28]. 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 monolith 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 monolith 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 monolith 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 entry corresponding to the file 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 monolith 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 9.

$G = (V, E)$

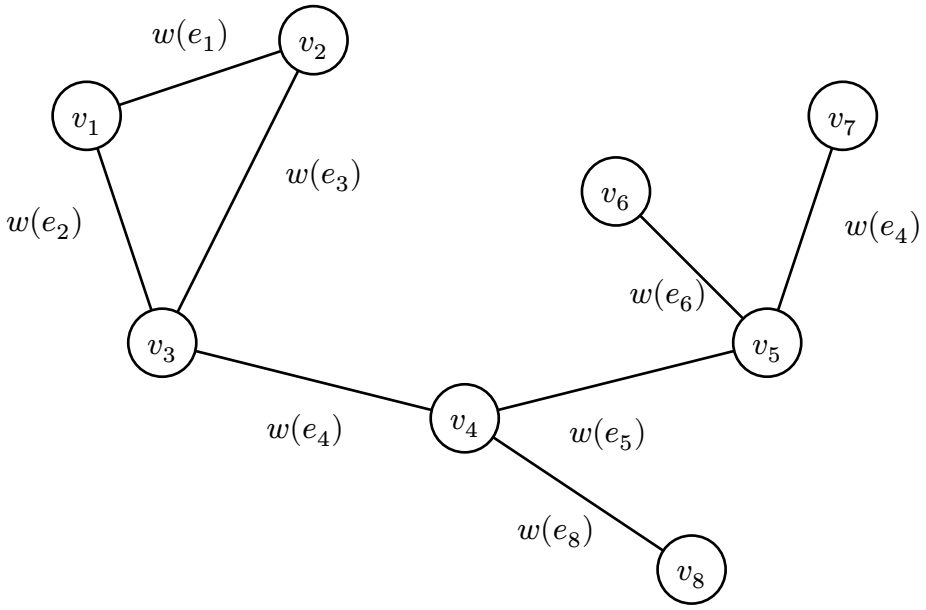


Figure 9: Dependency graph

7.5 Decomposition

Monolith decomposition is the process of identifying microservice candidates in a monolith application. The goal of this process is to split the monolith into smaller, more manageable software components which can be deployed independently [51]. 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, Li et al. [P22] proposed 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 monolith 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 does not require specifying the number of clusters up-front, but it is computationally expensive as well [36].

We found that the Louvain [41] and Leiden [42] 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 Rahiminejad et al. [52] 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 [41] performed best in terms of quality and speed.

The Louvain/Leiden algorithm

The Louvain algorithm, introduced by Blondel et al. [41], 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.

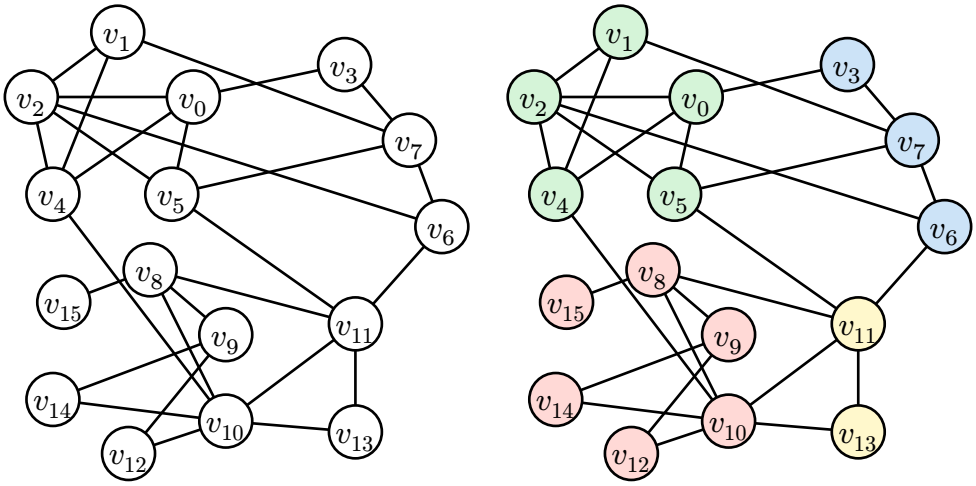
Modularity is a measure of the strength of division of a network. Networks with high modularity have dense connections between the internal vertices of a community, and sparse connections between vertices of different communities. The domain of the metric is between -0.5 (non-modular clustering) and 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 [53]:

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

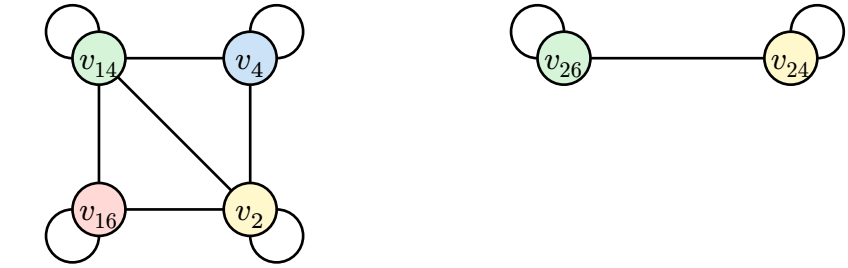
Where:

- 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 10: Visualization of the Louvain algorithm

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 10.

A major disadvantage of the Louvain algorithm is that it can only detect non-overlapping communities [41]. 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 [42], which is not desirable in the context of microservices [54].

In 2019, Traag et al. [42] 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 [55]:

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

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 [42].

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 performant than the Louvain algorithm, it is more complex to implement due to the refinement phase. Hence, the proposed solution uses the Louvain algorithm as the default clustering algorithm, but allows easy integration of additional algorithms such as the Leiden algorithm.

7.6 Visualization

MOSAIK can generate visualizations of the steps in the microservice decomposition process. The information extraction step outputs a dependency graph, where the vertices represent the classes of the monolith application, and the edges represent the dependencies between the classes.

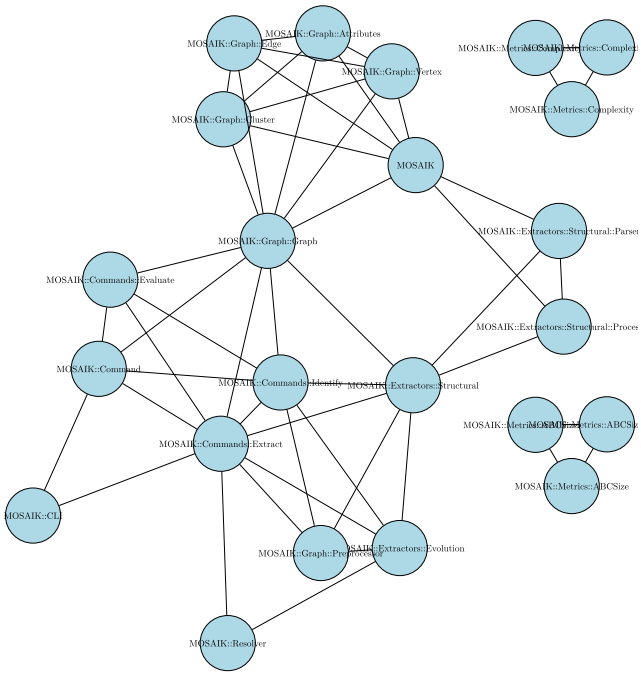


Figure 11: Visualization of information extraction step

As each coupling strategy can create one or more edges, the graph can contain significant visual cluttering and can be difficult to interpret. The graph visualization implementation accepts several parameters to control the layout and appearance of the graph, such as aggregating the edges between two vertices, filtering out edges with a low weight, and hiding vertices with no outgoing or incoming edges. The application offers multiple renderers to visualize the graph, using different layout algorithms and styles. Figure 11

illustrates the visualization of the information extraction step applied to the source code of the application itself.

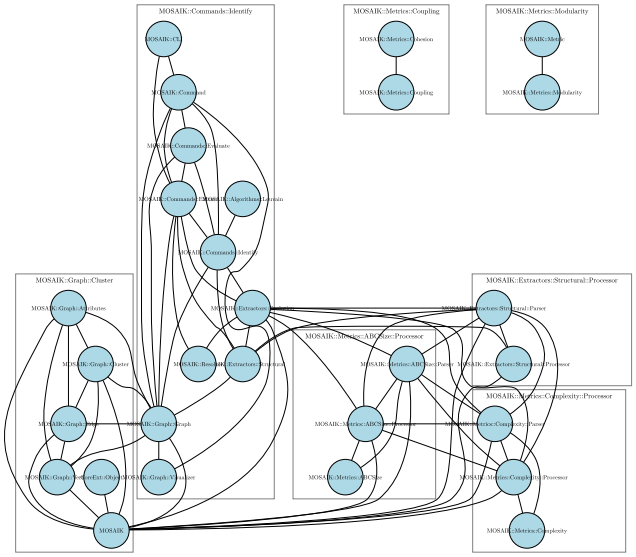


Figure 12: Visualization of decomposition step

The decomposition step does not modify the graph structure, but adds subgraphs to the visualization to indicate the microservice candidates.

The graphs are rendered using the open-source Graphviz software[13], which provides a set of tools for generating and manipulating graph visualizations using the DOT language[14].

[13]https://graphviz.org/
[14]https://graphviz.org/doc/info/lang.html

7.7 Evaluation

Several quality metrics were considered to evaluate the quality of the microservice decomposition. Carvalho et al. [56] 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 [56].

Candela et al. [47] 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.

Coupling

Coupling is a measure of the degree of interdependence between modules in a software system [57]. 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 [57]. 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 defined 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 [58] 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[15]. 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 that the microservice candidates are too small, and that the decomposition is too fine-grained.

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

Complexity

Cyclomatic complexity is a metric that quantifies the number of linearly independent control paths through the source code of a program [59]. The measure is computed by constructing a control-flow graph of the program, and counting the number of possible paths through the graph. Each vertex in the graph represents a group of non-branching instructions, and each edge represents a possible transfer of control between the groups. If the program does not contain any branching instructions, the complexity is 1 (there is only one path).

Like ABC size, cyclomatic complexity in the context of microservices can be defined as the averaged sum of the cyclomatic complexities of all methods in a microservice candidate $M_c \in S$.

$$complexity(M_c) = \frac{\sum_{v_i \in M_c} brch(v_i)}{|v_i|} \tag{26}$$

Where $brch(v_i)$ returns the number of branches in method v_i.

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

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

A solution with a lower total complexity is considered to be better, as it indicates microservice candidates that are easier to understand and maintain.

8 Case study

In this chapter, we present a case study to evaluate the effectiveness of the proposed solution using a real-world use case. We focus on answering 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 called NephroFlow™ Link. The application is developed and distributed by Nipro Digital Technologies Europe NV[16], a subsidiary of Nipro[17]. Nipro is a leading global manufacturer of medical devices, specialized in renal care products. NephroFlow™

[16]https://www.niprodigital.com
[17]https://www.nipro-group.com

Link, part of the NephroFlow™ Product Suite[18], is an application that allows the NephroFlow™ Healthcare Platform to communicate with the dialysis machines installed at hospitals and dialysis centers, and vice versa. NephroFlow™ Link is responsible for collecting data from the dialysis machines, processing it, and sending it to the NephroFlow™ Platform for storage and visualization.

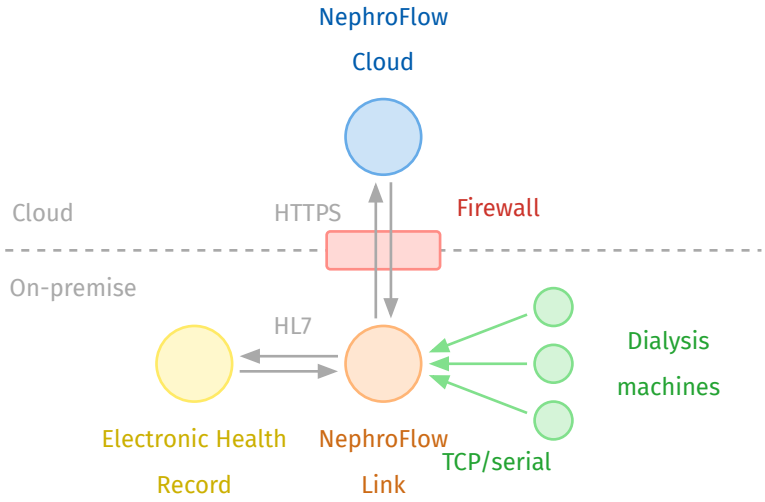


Figure 13: NephroFlow™ Link architecture overview

Dialysis machines measure and publish 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 has deployed NephroFlow™ Link in dozens of dialysis centers and hospitals across Europe, Central America, India, and Japan, collectively ensuring connection to hundreds of dialysis machines. To safeguard the privacy and information security of the patient, NephroFlow™ Link is

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

deployed on a virtual server or hardware appliance within the physical and virtual premises of the hospital or dialysis center, and the data is sent to a secure cloud environment.

The application is written in the Ruby programming language[19] as a single-threaded process, deployed as a single unit. The choice of Ruby as programming language was influenced by the Ruby on Rails framework[20], which is used for the NephroFlow™ Platform. The application itself is not stateful, and stores only transitional data (e.g. for rate-limiting purposes) using the Redis key-value datastore[21]. The source code of the application is hosted in a private Github repository[22] and is not publicly available.

NephroFlow™ Link currently supports integration with 13 dialysis machine models, and the number of supported devices grows year by year. The codebase of the application is rapidly becoming increasingly complex, which decreases the development velocity of new features and makes the application harder to maintain. When it is deployed at bigger sites with up to 400 dialysis machines, the throughput and latency suffer and performance issues arise. Additionally, patient safety and information security would benefit from a more modular architecture, as it would allow for a more fine-grained access control to the data.

For these reasons, the application would benefit from an architectural overhaul. While a microservices-based architecture would allow the application to scale efficiently, it also introduces a significant amount of complexity, and requires an upfront investment in development time. Since the number of developers working on NephroFlow™ Link is limited, migrating to a microservices architecture would be challenging. Hence, we

[19]https://www.ruby-lang.org
[20]https://rubyonrails.org
[21]https://www.redis.com
[22]https://github.com

believe that migrating NephroFlow™ Link towards a modular monolith architecture would prove beneficial.

8.2 Experimental setup

Lourenço and Silva [P17] analyzed multiple source code repositories and concluded that repositories with a large number of committers perform better when considering the contributor coupling in various scenarios. 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 improve the quality of the decomposition. Four test scenarios 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 8 for a list of the test configurations. Strategies using a single coupling are not considered, as they extract limited information by themselves, and are not expected to perform well in the context of NephroFlow™ Link.

ID	ω_s	ω_c	ω_d	**Scenario**
[S1]	1	1	0	*structural-logical*
[S2]	1	0	1	*structural-contributor*
[S3]	0	1	1	*logical-contributor*
[S4]	1	1	1	*structural-logical-contributor*

Table 8: Test configurations

The source code repository of the application contains 204 Ruby source code files, with a total of 9288 Source Lines of Code (SLOC), as measured by the `cloc` tool[23]. Only the application code is considered, excluding tests and configuration files.

The repository contains a `main` branch with the latest code, and several branches for released versions. For the purpose of this study, we only consider the `main` branch, from the release of NephroFlow™ Link version 5.0 on October 27, 2023 up until the pre-release of NephroFlow™ Link version 5.2 on April 25, 2024, which is the most recent commit in the repository 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 are automatically generated by Github to update the dependencies of the application[24].

An overview of the source code repository is presented in Table 9.

SLOC	**Classes**	**Methods**	**Commits**	**Contributors**
9288	207	840	277	10

Table 9: Source code statistics

[23]https://github.com/AlDanial/cloc
[24]https://github.com/features/security

We identified 10 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 80% of the commits, while the other eight contributors account for the remaining 20% of the commits.

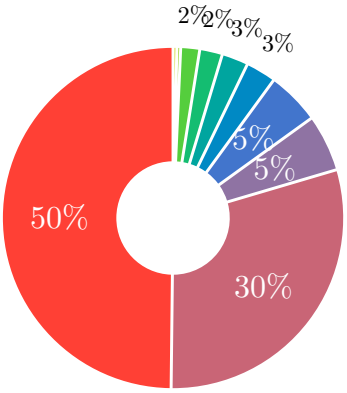


Figure 14: Commits by contributor

8.3 Evaluation and results

In this section, we present the results of the decomposition of NephroFlow™ Link using MOSAIK. The results are based on the four test scenarios described in the previous section.

Figures 15 to 18 show the coupling, cohesion, ABC size, and complexity metrics, respectively, for each scenario using a box plot. The plots indicate the distribution of the metrics for each scenario, calculated from the individual metrics of each microservice in the decomposition.

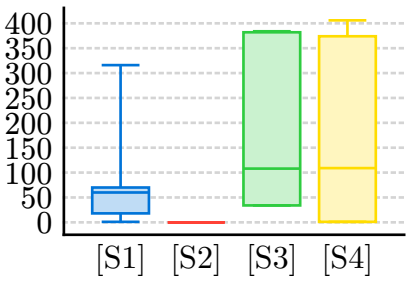


Figure 15: Coupling statistics

The coupling metric measures how loosely coupled the microservices are in relation to each other, with a lower value indicating a better modularization. Scenario [S3] and Scenario [S4] have a similar mean coupling value at 227.0 and 222.5 respectively, with few outliers.

Scenario [S1] has a much lower mean coupling value at 83.2, indicating that the microservices are more loosely coupled, and the decomposition is more modular. On the other hand, Scenario [S2] has a mean coupling value of 0.0, which means that the microservices are not coupled at all. This can happen when the decomposition is too fine-grained, and the microservices are too small to be useful. Looking at the size of the microservices in Figure 20, we see that the microservices are very small, with the exception of one microservice that is significantly larger than the others, which explains the low coupling value.

The cohesion metric measures how well the microservices are internally cohesive and group related functionality together. A higher value indicates a better modularization.

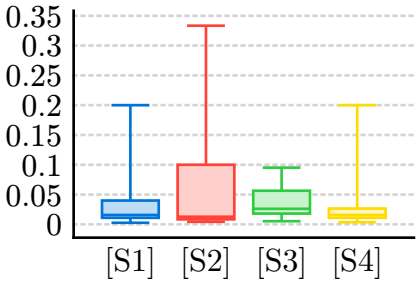


Figure 16: Cohesion statistics

All scenarios have a similar mean cohesion value, ranging from 0.04 to 0.07 for Scenario [S2]. The latter is likely caused by the significantly larger microservice, which raises the mean cohesion value.

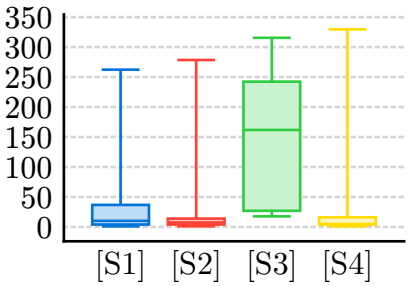


Figure 17: ABC size statistics

ABC size measures the size of a microservice given the assignments, branches, and conditions in the code. A lower value indicates a smaller microservice, which is generally preferred.

In Figure 17, we see that the extrema of the ABC size metric are quite high for all scenarios, with values ranging from 262.4 to 329.6. This indicates that some microservices are significantly larger than others. As the number of microservices in the decomposition of Scenario [S3] is quite small, it has

the least variation in the ABC size metric, with a mean value of 152.8 and a median value of 161.8. The other scenarios have mean ABC size values ranging from 23.9 to 66.9, indicating that the microservices in these decompositions are on average smaller in size, but with more variation.

Figure 18 shows the cyclomatic complexity of the microservices in the decomposition. A lower value indicates a simpler microservice, which is generally preferred.

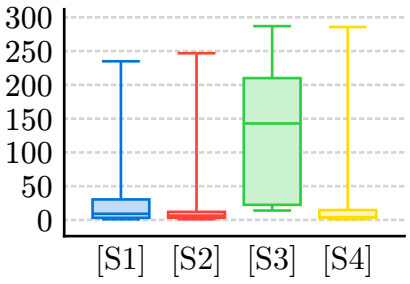


Figure 18: Complexity statistics

The statistical values for the complexity metric are similar to the ABC size metric, with the mean complexity values ranging from 21.1 to 59.2, with Scenario [S3] being an outlier with a mean complexity value of 135.2, and a median value of 142.8. ABC size and complexity are closely related, as they both measure the perceived complexity of the code, but they do so in different ways.

Table 10 lists the mean values of the metrics for each scenario. The mean value of each metric indicates the average value of the metric for all microservices in the decomposition, and can be used as a reference to compare the scenarios.

ID	Coup	Coh	Size	Cplx
[S1]	83.25	0.04	39	34
[S2]	0	0.07	24	21
[S3]	227	0.04	153	135
[S4]	222.5	0.04	67	59

Table 10: Mean of metric per scenario

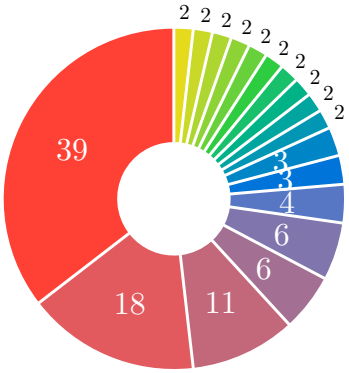


Figure 19: Microservice size distribution (Scenario [S1])

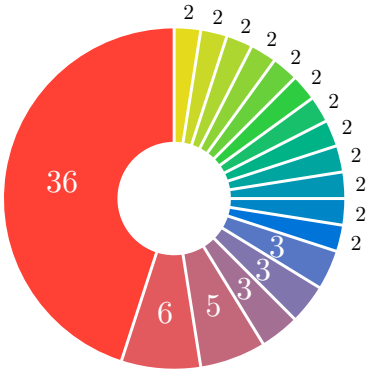


Figure 20: Microservice size distribution (Scenario [S2])

Scenario [S1] and Scenario [S2] each have 18 microservices, with a mean size of 6.1 and 4.4 classes per microservice respectively. Scenario [S4] has slightly fewer at 14 microservices, with a larger mean size of 9.2 classes per microservice. These decompositions consists of several larger microservices, and a number of smaller microservices that contain only a few classes.

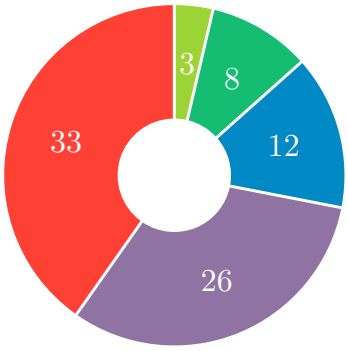


Figure 21: Microservice size distribution (Scenario [S3])

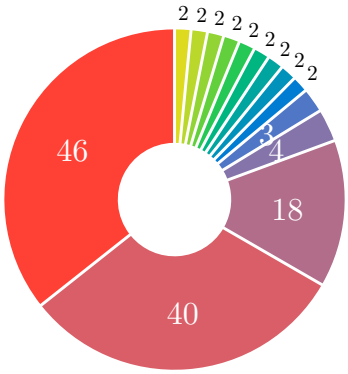


Figure 22: Microservice size distribution (Scenario [S4])

ID	$\#M_c$	$\overline{\#M_c}$
[S1]	18	6.1
[S2]	18	4.4
[S3]	5	16.4
[S4]	14	9.2

Table 11: Number of microservices per scenario

Table 11 lists the number of microservices identified by MOSAIK for each scenario. Aside from Scenario [S3], the number of microservices identified by MOSAIK is consistent across the scenarios. Scenario [S3], which consists of a test setup using only evolutionary coupling, identified a significantly lower number, five microservices. The lack of structural coupling information has a significant impact on the size of the microservices.

Investigating further into the microservices identified in Scenario [S3], we observe that the granularity of the microservices is significantly lower than in other scenarios, with some microservices containing seemingly unrelated functionality. Some of the microservices include a mix of data processing code, utility functions, and data access code. The other scenarios, which include structural coupling information, identified microservices that have more related functionality grouped together. Due to the restricted nature of the source code, precise information about the functionality for each microservice cannot be included in this report.

Figure 23 depicts the runtime of the analysis for each scenario. Each execution is divided into three phases: extraction, decomposition, and evaluation. The extraction step is equal for all scenarios, as it is based on the same input data. This step takes around 8 seconds to complete. The second step, decomposition, is the most time-consuming step, as it involves the iterative process of identifying the microservices. Finally, the evaluation step executes in a similar time for all scenarios, ranging from 3.4 to 4.9 seconds.

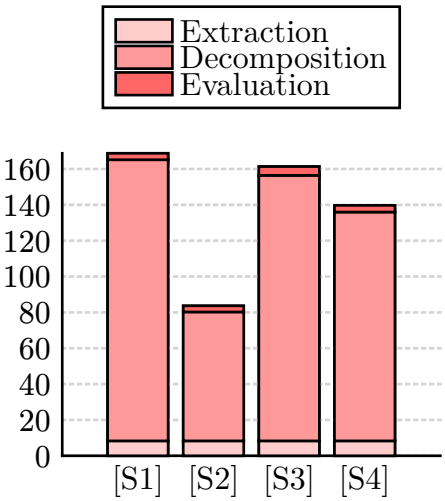


Figure 23: Total analysis runtime

The decomposition in scenarios [S1], [S3], and [S4] had a similar runtime (157, 148, and 128 seconds respectively), while the scenario [S2] executed in less than half the time: 72 seconds. Scenario [S2] is the fastest scenario, as it does not consider the logical coupling, which contains a lot of information that needs to be processed.

8.4 Discussion

The results of the case study show that the decomposition of NephroFlow™ Link into a modular monolith architecture using MOSAIK is feasible. The various test scenarios provide insights into how modularization behaves, with varying levels of success in terms of coupling, cohesion, and complexity. Scenario [S3] indicates that structural coupling is an integral part of the extracted information, and decomposition performs poorly when not considering it. Scenario [S2] shows that the granularity of the decomposition can be too fine-grained, resulting in microservices that are too small to be useful. Scenario [S4], the scenario that considers all three types of coupling, performs well in terms of coupling, cohesion, and complexity, though the microservices end up with a tighter coupling than the scenario that only considers structural and logical coupling. Given the results, we can conclude that the quality requirement is met, as the proposed solution is able to identify module boundaries with sufficient quality.

MOSAIK is able to automatically generate decompositions of the application's source code without intervention of the software architect, fulfilling the automation requirement. The tool is able to generate visualizations of the decompositions, which can be used to gain insight into the structure of the application, fulfilling the visual requirement.

Finally, the runtime of the analysis is acceptable for the source code of NephroFlow™ Link, with the decomposition of the application taking less than three minutes to complete. However, the runtime of the analysis may increase significantly for larger applications, as the time complexity of the algorithm is $O(n \log(n))$ [60]. Hence, we conclude that the performance

requirement is only partially met, as the tool may not be performant enough for very large applications.

The results indicate that the transformation of NephroFlow™ Link into a modular monolith architecture is feasible and can provide numerous benefits, such as improved development velocity, as well as increasing the overall performance of the application.

In order to validate the ability of MOSAIK to perform qualitative decompositions on other applications, more case studies should be conducted. In the industry there are many applications written in Ruby that are open-source, and could be used for this purpose. Furthermore, as only the parsing of the source code is language-specific, the extraction part of the tool could be rewritten to support other dynamic languages (e.g. Python) as well.

9 Conclusion

This thesis discussed the problem of (semi-)automated modularization of monolith applications, with a focus on the automated identification of microservice candidates. In the first part of the thesis, we investigated the modular monolith architecture, and discussed the merits and demerits of the software architecture. Then, we presented a comprehensive overview of the state of the art on (semi-)automated technologies for modularization of monolith applications. We identified and described the most frequently used approaches in the literature, and compared them against each other.

In the third part, we presented our solution for the automated identification of microservice candidates in monolith codebases, based on the findings of our systematic literature review. We comprehensively described the architecture of our tool, and discussed the technical implementation details.

Finally, we evaluated the effectiveness of our solution using a case study on a real-world monolith application. We conclude that our solution is able to identify microservice candidates with an acceptable level of quality, and that it can be used as a basis for further research and development in this area.

9.1 Future considerations

The research field of automated modularization of monolith applications is still developing, and there are many opportunities for improvements. In recent years, there has been a pickup in the number of publications on this topic, and we expect to see more research papers published in the near future.

There are several areas where we see potential for improvement in our proposed solution. As many experts have already pointed out, the information extracted from the codebase can be improved by using additional techniques. For example, the structural coupling can be improved by dynamically collecting information during the runtime of the application, as static analysis is very limited in highly dynamic languages such as Ruby and Python. Furthermore, enhancing the extracted information with additional object-oriented relationships (e.g. inheritance and composition) can improve the accuracy of the microservice identification algorithm. Alternatively, supplementary heuristics, such as semantic coupling, can be utilized as well.

Finally, the MOSAIK tool can be improved technically. For instance, rewriting critical parts of the identification algorithm in another, more performant language could yield large gains in runtime performance. Moreover, the tool can be extended to support more programming languages, as the current implementation only supports Ruby.

References

[1] 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.

[2] M. Tsechelidis, N. Nikolaidis, T. Maikantis, and A. Ampatzoglou, "Modular Monoliths the Way to Standardization," in *Proceedings of the 3rd Eclipse Security, AI, Architecture and Modelling Conference on Cloud to Edge Continuum*, Ludwigsburg Germany: ACM, Oct. 2023, pp. 49–52. doi: 10.1145/3624486.3624506.

[3] A. Küçükoğlu, Mar. 29, 2022. Accessed: Apr. 22, 2024. [Online]. Available: https://www.ahmetkucukoglu.com/en/what-is-modular-monolith

[4] 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.

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

[6] E. Evans, "Domain-Driven Design Reference: Definitions and Pattern Summaries." 2015.

[7] R. Smith *et al.*, *Professional Active Server Pages 2.0*, 2nd ed. GBR: Wrox Press Ltd., 1998.

[8] A. Parikh, P. Kumar, P. Gandhi, and J. Sisodia, "Monolithic to Microservices Architecture - A Framework for Design and Implementation," in *2022 International Conference on Computer, Power and Communications (ICCPC)*, Chennai, India: IEEE, Dec. 2022, pp. 90–96. doi: 10.1109/ICCPC55978.2022.10072238.

[9] E. W. Dijkstra, *Selected writings on computing: a personal perspective*. Berlin, Heidelberg: Springer-Verlag, 1982.

[10] *The Open Group Architecture Framework (TOGAF)*. 2024. Accessed: Mar. 08, 2024. [Online]. Available: https://www.opengroup.org/togaf

[11] "Microservices: a definition of this new architectural term." 2014. Accessed: Mar. 08, 2024. [Online]. Available: https://martinfowler.com/articles/microservices.html

[12] T. Cerny, M. J. Donahoo, and J. Pechanec, "Disambiguation and Comparison of SOA, Microservices and Self-Contained Systems," in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, Krakow Poland: ACM, Sep. 2017, pp. 228–235. doi: 10.1145/3129676.3129682.

[13] "Microservices vs. Service-Oriented Architecture." O'Reilly Media, 2015.

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

[15] A. Abran, P. Bourque, R. Dupuis, and J. W. Moore, *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, 2001.

[16] 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.

[17] A. Tahir and S. G. MacDonell, “A Systematic Mapping Study on Dynamic Metrics and Software Quality,” in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Trento, Italy: IEEE, Sep. 2012, pp. 326–335. doi: 10.1109/ICSM.2012.6405289.

[18] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, “Service Cutter: A Systematic Approach to Service Decomposition,” *Service-Oriented and Cloud Computing*, vol. 9846. Springer International Publishing, Cham, pp. 185–200, 2016. doi: 10.1007/978-3-319-44482-6_12.

[19] D. Pereira da Rocha, “Monólise: Uma técnica para decomposição de aplicações monolíticas em microsserviços.” 2018.

[20] S. Petrovic and G. Sawhney, “Introducing Service Weaver: A Framework for Writing Distributed Applications.” Mar. 01, 2023. Accessed: Mar. 01, 2023. [Online]. Available: https://opensource.googleblog.com/2023/03/introducing-service-weaver-framework-for-writing-distributed-applications.html

[21] 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.

[22] R. Su and X. Li, “Modular Monolith: Is This the Trend in Software Architecture?.” Accessed: Mar. 01, 2024. [Online]. Available: http://arxiv.org/abs/2401.11867

[23] B. Foote and J. Yoder, “Big Ball of Mud.” Aug. 26, 1997.

[24] S. Brown, “Software architecture for developers,” *Coding the Architecture*, 2013.

[25] K. Grzybek, “Modular Monolith: Architectural Drivers.” Dec. 26, 2019. Accessed: Apr. 22, 2024. [Online]. Available: https://www.kamilgrzybek.com/blog/posts/modular-monolith-architectural-drivers

[26] M. Fowler, “Monolith first.” Jun. 03, 2015. Accessed: Apr. 22, 2024. [Online]. Available: https://martinfowler.com/bliki/MonolithFirst.html

[27] A. Kodja, “Modular Monoliths vs. Microservices.” Aug. 06, 2023. Accessed: Apr. 22, 2024. [Online]. Available: https://adriankodja.com/modular-monoliths-vs-microservices

[28] M. E. Conway, “How do committees invent?,” 1968.

[29] K. Grzybek, “Modular Monolith: A Primer.” Dec. 02, 2019. Accessed: Apr. 22, 2024. [Online]. Available: https://www.kamilgrzybek.com/blog/posts/modular-monolith-primer

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

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

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

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

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

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

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

[44] 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.

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

[46] 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.

[47] 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.

[48] 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.

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

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

[51] 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

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] “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.

[58] J. Fitzpatrick, "Applying the ABC Metric to C, C++, and Java," 1997.

[59] T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, no. 4, pp. 308–320, Dec. 1976, doi: 10.1109/TSE.1976.233837.

[60] A. Lancichinetti and S. Fortunato, "Community Detection Algorithms: A Comparative Analysis," *Physical Review E*, vol. 80, no. 5, p. 56117–56118, Nov. 2009, doi: 10.1103/PhysRevE.80.056117.

[61] 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.

A Systematic Literature Review publications

Primary studies

ID	Publication
[**P1**]	Saidi, Tissaoui and Faiz, *A DDD Approach Towards Automatic Migration To Microservices*, 2023
[**P2**]	Yang, Wu and Zhang, *A Microservices Identification Approach Based on Problem Frames*, 2022
[**P3**]	Kinoshita and Kanuka, *Automated Microservice Decomposition Method as Multi-Objective Optimization*, 2022
[**P4**]	Zhou and Xiong, *Automated Microservice Identification from Design Model*, 2022
[**P5**]	Zhang et al., *Automated Microservice Identification in Legacy Systems with Functional and Non-Functional Metrics*, 2020
[**P6**]	Quattrocchi et al., *Cromlech: Semi-Automated Monolith Decomposition Into Microservices*, 2024
[**P7**]	Kamimura et al., *Extracting Candidates of Microservices from Monolithic Application Code*, 2018
[**P8**]	Al-Debagy and Martinek, *Extracting Microservices' Candidates from Monolithic Applications: Interface Analysis and Evaluation Metrics Approach*, 2020
[**P9**]	Mazlami, Cito and Leitner, *Extraction of Microservices from Monolithic Software Architectures*, 2017
[**P10**]	Selmadji et al., *From Monolithic Architecture Style to Microservice One Based on a Semi-Automatic Approach*, 2020
[**P11**]	Filippone et al., *From Monolithic to Microservice Architecture: An Automated Approach Based on Graph Clustering and Combinatorial Optimization*, 2023

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

[**P31**]	Sellami, Saied and Ouni, *A Hierarchical DBSCAN Method for Extracting Microservices from Monolithic Applications*, 2022
[**P32**]	Hasan et al., *AI-based Quality-driven Decomposition Tool for Monolith to Microservice Migration*, 2023
[**P33**]	Nitin et al., *CARGO: AI-Guided Dependency Analysis for Migrating Monolithic Applications to Microservices Architecture*, 2022
[**P34**]	Cao and Zhang, *Implementation of Domain-oriented Microservices Decomposition Based on Node-attributed Network*, 2022
[**P35**]	Santos and Paula, *Microservice Decomposition and Evaluation Using Dependency Graph and Silhouette Coefficient*, 2021
[**P36**]	Kalia et al., *Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices*, 2021
[**P37**]	Eski and Buzluca, *An Automatic Extraction Approach: Transition to Microservices Architecture from Monolithic Application*, 2018
[**P38**]	Wu, Hassan and Holt, *Comparison of Clustering Algorithms in the Context of Software Evolution*, 2005
[**P39**]	Baresi, Garriga and De Renzis, *Microservices Identification Through Interface Analysis*, 2017
[**P40**]	Kalia et al., *Mono2Micro: An AI-based Toolchain for Evolving Monolithic Enterprise Applications to a Microservice Architecture*, 2020
[**P41**]	Saidani et al., *Towards Automated Microservices Extraction Using Multi-objective Evolutionary Search*, 2019

Table 12: Selected publications (primary studies)

Secondary studies

ID	Publication
[**P42**]	Bajaj et al., *A Prescriptive Model for Migration to Microservices Based on SDLC Artifacts*, 2021
[**P43**]	Abgaz et al., *Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review*, 2023
[**P44**]	Oumoussa and Saidi, *Evolution of Microservices Identification in Monolith Decomposition: A Systematic Review*, 2024
[**P45**]	Schmidt and Thiry, *Microservices Identification Strategies : A Review Focused on Model-Driven Engineering and Domain Driven Design Approaches*, 2020
[**P46**]	Kazanavicius and Mazeika, *Migrating Legacy Software to Microservices Architecture*, 2019
[**P47**]	Mparmpoutis and Kakarontzas, *Using Database Schemas of Legacy Applications for Microservices Identification: A Mapping Study*, 2022
[**P48**]	Fritzsch et al., *From Monolith to Microservices: A Classification of Refactoring Approaches*, 2019

Table 13: Selected publications (secondary studies)