Automated Microservice Identification: an Approach to Decomposition into a Modular Monolith Architecture

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

UNIVERSITY OF TURKU
Department of Computing

Florian Dejonckheere: Automated Microservice Identification: an Approach to Decomposition into a Modular Monolith Architecture

Master of Science (Tech) Thesis, 29 p., 6 app. p.
Department of Computing
April 2024

The modular monolith architecture emerged in recent years as the harmonization of the monolithic and microservices architectures. The paradigm offers a compromise between modularity, flexibility, and scalability. Many monolithic applications are being migrated to modular monoliths or microservices entirely, to satisfy increasingly complex and volatile business requirements. This process is labour-intensive, slow, and may take months to years for larger codebases. Modularization of a codebase typically requires the developer to have an intimate knowledge of both the application code and domain.

In this thesis, we investigate the modular monolith software architecture, and how modules are typically determined as part of the modularization efforts. We propose an automated solution based on dependency analysis and machine learning algorithms to aid in the identification of module boundaries, and evaluate its effectiveness using a case study. We discuss the results and draw conclusions about the propsed solution.

Keywords: software architecture, monolith, microservices, modular monolith

Contents

List of Figures

List of Tables

List of Acronyms

AST	Abstract Syntax Tree
BPMN	Business Process Model and Notation
DSRM	Design Science Research Methodology
DSRP	Design Science Research Process
SDLC	Software Development Life Cycle
SLR	Systematic Literature Review

1. Introduction

1.1. Scope and goal

This research is centered around three research questions:

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

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

Research Question 3: How can static analysis of source code effectively identify optimal module boundaries in a modular monolith architecture?

To answer the first research question, we will first define the modular monolith architecture, and examine what sets it apart from monolithic and microservices architectures. Then, we will proceed to investigate the merits and drawbacks of the software architecture when applied to an existing codebase.

For the second research question, we will enumerate the existing technologies to aid modularization of monolithic codebases, and choose one automated technology for further examination. *(Automated technology)* will then be implemented for a given use case, and compared to manual modularization efforts in terms of accuracy, efficiency, development velocity. This comparison will help us to answer the third research question.

The goal of this research can be summarized as follows:

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

The proposed solution will add value to the field of software engineering, and will be able to be used as a base for future improvements regarding automated modularization of monolith codebases.

1.2. Motivation

1.3. Methodology

A literature review is conducted to answer the first and second research question. For the first research question, the study aims to find a definition of the modular monolith architecture, and to list the advantages and disadvantages of the architecture based on existing literature. For the second research question, the state of the art in automated modularization technologies is reviewed and summarized.

The third research question is answered by choosing the most appropriate automated technology, and implementing it for a given use case. The implementation is then evaluated based on quantitative and qualitative metrics, and compared to manual modularization efforts.

Finally, the findings are summarized, and an outlook on future work is given.

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]. Their model is based on prior research and is designed to guide researchers through the process of analysis, creation, and evaluation of artifacts.

The six steps of the process are:

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

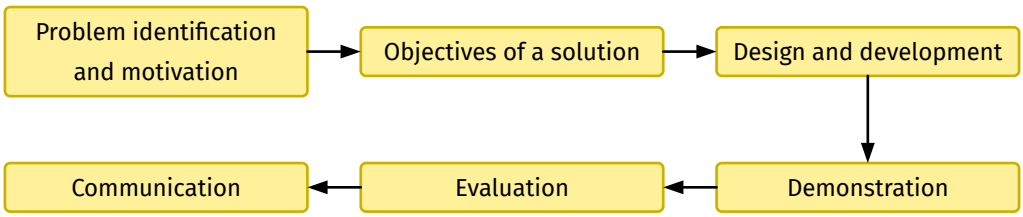

Figure 1: Design Science Research Process (DSRP)

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

In this thesis specifically, the DSRP is used to guide the design and development of the automated modularization technology, with a particular focus on the design and development, demonstration, and evaluation steps.

1.4. Outline

The thesis is divided into three parts.

The first part comprises the background and related work. In Chapter 1, the scope and goal of the research is defined, and the research questions are formulated. The stakeholders are identified, and the methodology is explained. Chapter 2 introduces the reader to the research background and necessary concepts. In Chapter 3, the existing literature is reviewed, and the state of the art is presented.

The second part of the thesis, starting with Chapter 4, is dedicated to the first research question. The modular monolith architecture is defined, and its merits and drawbacks are discussed.

The third part aims to solve the second and third research question. Chapter 5 gives an introduction into the automated modularization of monolith codebases, listing the existing technologies. It then continues to focus on one automated technology, *(automated technology)*, and explains its implementation. Chapter 7 applies *(automated technology)* on a given case study, and compares it to manual modularization efforts.

Finally, Chapter 8 summarizes the findings, and gives an outlook on future work.

2. Background

2.1. Monolith architecture

2.2. Modular programming

2.3. Microservice architecture

2.4. Modularization

3. Related work

4. Modular monolith architecture

4.1. Background

4.2. Challenges and opportunities

4.3. Modularization

5. 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 regarding automated modularization of monolith codebases. We also provided a brief overview of the most relevant approaches and tools.

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

The literature review was conducted using a three-step protocol as defined by Kitchenham & Charters [2]:

	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

5.1. 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 [3], [4], as well as systematic literature reviews related to microservice decomposition . However, in these studies the techniques described are mainly used as an aid for the software architect when identifying microservice candidates. Therefore, we believe that there is a need for a systematic literature review aimed at summarizing existing literature regarding fully automated techniques for modularization of monolith codebases.

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

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

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

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

- *Architecture*: the architectural styles being discussed in the publications.
  Keywords: microservice, monolith, modular monolith
- *Modularization*: the process of identifying and decomposing modules in a monolith architecture.
  Keywords: service identification, microservice decomposition, monolith modularization
- *Technology*: the technologies, algorithms, or methods for modularization.
  Keywords: automated tool, machine learning, static analysis, dynamic analysis, hybrid analysis

The resulting search query can be expressed as follows:

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

Listing 1: Search query

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

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

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

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

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

Table 2: Inclusion and exclusion criteria

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

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 approach or technique
- The publication includes a validation phase or case study demonstrating the effectiveness of the approach or technique

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 type of approach or technique they describe.

[3]Development of new software systems lacking constraints imposed by prior work [5]
[4]Development of new software systems in the presence of legacy software systems [5]

5.2. 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		4
Total	507	46

Table 3: Summary of search results

After applying the inclusion/exclusion criteria, we selected 42 publications for inclusion in the systematic literature review. Of these publications, 35 are primary studies, and 6 are secondary studies. The secondary studies were used to categorize the selected primary studies (if any), and as a starting point for the snowballing process, which resulted in 4 additional publications being included in the systematic literature review. For a list of the selected publications, see Appendix A.

From the selected publications, we extracted relevant information, such as the type of approach or technique described, the algorithms used, and the metrics discussed. As Kitchenham & Charters [2] suggest for systematic literature reviews done by single researchers, the data extraction was 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.

5.3. Report

The publications selected for inclusion in the systematic literature review were qualitatively reviewed and categorized in three dimensions.

To begin with, we categorized the publications based on the Software Development Life Cycle (SDLC) artifact they use as input for the microservice candidate identification algorithm. Each artifact category has an associated collection type, either static, dynamic, or hybrid. [6]. Static collection describes a SDLC artifact that was collected without executing the software, while dynamic collection describes a SDLC artifact that was collected after or during execution of the software. Some publications describe algorithms or techniques that use a combination of SDLC artifacts, which we categorized as hybrid.

Thereafter we categorized the publications based on the algorithm used for microservice candidate identification. The algorithms were subdivided into several classes based on the technique.

Ultimately, the publications were also categorized by the metrics discussed.

5.3.1. SDLC artifact

The identified SDLC artifact categories used as input for the microservice candidate identification algorithm are described in Table 4. The categories are based on Bajaj, Bharti, Goel, & Gupta [6].

Artifact	Type	Publications
Requirements documents and models	Static	[7]–[11]
Design documents	Static	[12]–[16]
Code	Static	[17]–[34], [16], [35]–[38]
Execution data	Dynamic	[21], [27], [31], [32], [39]–[42], [37]

Table 4: SDLC artifact categories

5.3.1.1. 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 [43]. 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 [7] and Daoud, El Mezouari, Faci, Benslimane, Maamar, & El Fazziki [8] model a software system as a set of business process using the industry standard Business Process Model and Notation (BPMN), using the machine-readable XML representation as input for the algorithm. Yang, Wu, & Zhang [9] tackle requirements engineering using problem frames [44]. Problem frames are a requirements engineering method, which emphasizes the integration of real-world elements into the software system [9].

Some approaches use schematic requirements documents in XML format as input for the algorithm, as described by Saidi, Tissaoui, & Faiz [10]. The latter use domain-driven design techniques to extract functional dependencies from the software design as starting point in microservice identification. Li, Zhang, Yang, Wang, & Yin [11] employ an intermediate format containing a precise definition of business functionality, generated from validated requirements documents.

5.3.1.2. 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 & Martinek [12] propose a data-driven method based on the analysis of the software system's external API, specified in the OpenAPI[5] format. The method extracts the information from the specification and converts it into vector representation for further processing.

Zhou & Xiong [13] use readily available design documents as well, in the form of UML class diagrams, use cases, and object sequence diagrams as starting point for the microservice identification algorithm. The decomposition tool proposed by Hasan, Osman, Admodisastro, & Muhammad [16] 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, Cocco, Staffa, Margara, & Cugola [14] takes a different approach to the problem, using a data-driven approach combined with a domain-driven approach. Software architects describe the software system using a custom architecture description language, and the tool developed by the authors is able to identify microservice candidates. The tool can be prompted to generate different, more efficient decompositions when given additional domain-driven requirements. Wei, Yu, Pan, & Zhang [15] uses a similar approach, gathering a list of features from the software architect, and proposing a microservice decomposition based on pre-trained feature tables.

5.3.1.3. Code

A third category of SDLC artifacts is the executable code 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 [25] 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.

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

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, Cito, & Leitner [18] originally proposed the use of the revision history of the source code to identify couplings between classes. The authors suggest multiple strategies that can be used to extract information from the revision history. Others have built upon this approach, using the revision history to identify the authors of the source code, and use this information to drive the identification algorithm [31], [38]

Escobar, Cardenas, Amarillo, Castro, Garces, Parra, & Casallas [17] use the source code of the software system to construct an Abstract Syntax Tree (AST), and map the dependencies between the business and data layer. Kamimura, Yano, Hatano, & Matsuo [19] use a more data-driven approach, and statically trace data access calls in the source code.

Many publications [20], [25], [27]–[29], [36]–[38] construct a dependency graph from Java source code, and use the graph as input for a clustering algorithm. Bandara & Perera [22] map object-oriented classes in the source code to specific microservices, but require a list of microservices to be specified before the decomposition is performed.

Filippone, Autili, Rossi, & Tivoli [23] concentrate on the API controllers as entrypoints into the software system. A later paper by the same authors [24] 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, Seriai, Seriai, Shatnawi, & Derras [28] make a distinction between presentation, business, and data layer.

Most of the publications tracing dependencies between classes (or modules) do this at the level of the classes (or modules). As Mazlami, Cito, & Leitner [18] 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, Garcia, Colanzi, Assuncao, Pereira, Fonseca, Ribeiro, De Lima, & Lucena [21] use a more granular approach, identifying dependencies between methods in the source code. On the other hand, Kinoshita & Kanuka [26] do not automatically extract information from the source code, but rely on a software architect to decompose the software system on the basis of business capability.

Romani, Tibermacine, & Tibermacine [30] propose a data-centric microservice candidate identification method based on knowledge gathered from the database schema. The authors extract table and column methods from the database schema, and use the semantically enriched information as input for the identification algorithm. Hao, Zhao, & Li [32] construct access patterns from both the database schema (static) and the database calls during execution of the software system (dynamic).

A unique approach to constructing a call graph is proposed by Nitin, Asthana, Ray, & Krishna [35], who make a distinction between context-insensitive and context-sensitive dependency graphs. While the former captures the dependencies between classes using simple method calls, the latter also includes the context (i.e., the arguments) of the method call in the dependency graph.

5.3.1.4. Execution

As a last category, information about the behaviour of the system can also be collected during the runtime of the software system. Execution data includes log files, execution traces, and performance metrics. This category is often combined with static analysis on source code, as the execution data can provide additional information to the identification algorithm. In dynamic languages such as Java, dynamic analysis can trace access patterns that static analysis cannot (e.g., due to late binding and polymorphism). Additionally, execution data can be collected when the source code of the software system is not available.

Examples of approaches using execution traces are Jin, Liu, Cai, Kazman, Mo, & Zheng [39] and Eyitemi & Reiff-Marganiec [42]. Using software probes inserted into the bytecode of respectively Java and .NET applications, the authors are able to monitor execution paths. Zhang, Liu, Dai, Chen, & Cao [40] collect the execution traces of the software system, in combination with performance logs.

Ma, Lu, & Li [41] use a data-centric approach based on the analysis of database access requests.

5.3.1.5. Hybrid approach

Some publications suggest a hybrid approach using both static and dynamic analysis. For instance, Wu & Zhang [27], Carvalho, Garcia, Colanzi, Assuncao, Pereira, Fonseca, Ribeiro, De Lima, & Lucena [21] and Cao & Zhang [37] collect information statically from the source code (entity classes and databases), as well as dynamically from the execution of the software system (execution traces). The approach proposed by Lourenço & Silva [31] uses either static of the source code or dynamic analysis of the system execution to gather access patterns.

Hao, Zhao, & Li [32] use both static and dynamic analysis, albeit aimed at the database schema and database calls, respectively.

5.3.2. Algorithms

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

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

Type	**Example algorithms**	**Publications**
Clustering algorithms	K-Means, DBSCAN, Hierarchical Agglomerative Clustering, Affinity Propagation	[8], [10], [12], [20], [22], [27]–[32], [41], [34]–[37]
Evolutionary algorithms	NSGA-II, NSGA-III	[7], [11], [13], [21], [26], [39], [40]
Graph algorithms	Kruskal, Louvain method, Leiden algorithm, Label Propagation	[9], [18], [24], [33], [16], [35], [37]
Other algorithms	Linear optimization, custom algorithms	[14], [17], [19], [20], [23], [25], [42], [15], [16], [38]

Table 5: Microservice candidate identification algorithm

5.3.2.1. Clustering algorithms

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

Examples of publications using K-Means clustering to identify microservice candidates are Saidi, Tissaoui, & Faiz [10], Wu & Zhang [27], Romani, Tibermacine, & Tibermacine [30], and Hao, Zhao, & Li [32].

Al-Debagy & Martinek [12] use Affinity Propagation [45] 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 [20], [29], [31], [41], [28], [22]. Lourenço & Silva [31] uses similarity between domain entities accesses and development history of source code files as a guiding measure for the clustering algorithm, while Zaragoza, Seriai, Seriai, Shatnawi, & Derras [28] uses structural and data cohesion of microservices. Daoud, El Mezouari, Faci, Benslimane, Maamar, & El Fazziki [8] extend the hierarchical agglomerative clustering (HAC) algorithm [46] with a collaborative approach, where the clustering is performed by multiple homogenous clustering nodes, each responsible for a subset of the data.

Selmadji, Seriai, Bouziane, Oumarou Mahamane, Zaragoza, & Dony [20] propose two possible algorithms for microservice identification: a hierarchical clustering algorithm, and a clustering algorithm based on gravity centers.

Sellami, Saied, & Ouni [34] use the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm [47] to identify microservices.

5.3.2.2. Evolutionary algorithms

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

Examples of publications using Non-Dominated Sorting Algorithm II (NGSA-II) as multi-objective optimization algorithm to identify microservice candidates are Zhou & Xiong [13], Kinoshita & Kanuka [26], Zhang, Liu, Dai, Chen, & Cao [40], Jin, Liu, Cai, Kazman, Mo, & Zheng [39], and Li, Zhang, Yang, Wang, & Yin [11]. Carvalho, Garcia, Colanzi, Assuncao, Pereira, Fonseca, Ribeiro, De Lima, & Lucena [21] use the next generation of NSGA, NSGA-III, in order to find a solution for the problem.

Amiri [7] rely on a genetic algorithm using Turbo-MQ [48] as fitness function.

5.3.2.3. Graph algorithms

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

For example, Mazlami, Cito, & Leitner [18] and Yang, Wu, & Zhang [9] use Kruskal's algorithm [49] 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, Qaisar Mehmood, Autili, Rossi, & Tivoli [24] apply the Louvain community detection algorithm [50] to obtain the granularity of the microservices, and high-cohesive communities of nodes. The Louvain method is a greedy optimization algorithm that aims to extract non-overlapping communities from a graph, using the modularity value as optimization target. Hasan, Osman, Admodisastro, & Muhammad [16] use the Leiden algorithm [51], an improvement of the Louvain method that uses a refinement step to improve the quality of the communities.

Cao & Zhang [37] use both the Leiden algorithm and the hierarchical clustering algorithm to identify microservice candidates. First, the Leiden algorithm is used to detect cohesive communities in static and dynamic analysis data, and then the hierarchical clustering algorithm is used to merge the communities into microservice candidates based on a call relation matrix.

Nitin, Asthana, Ray, & Krishna [35] use Context sensitive Label Propagation (CARGO), an algorithm built on the principles of the Label Propagation algorithm [52]. 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.

5.3.2.4. 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, Cocco, Staffa, Margara, & Cugola [14] 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, Autili, Rossi, & Tivoli [23] use a linear optimization algorithm to solve a combinatorial optimization problem.

The approach taken by Kamimura, Yano, Hatano, & Matsuo [19] is to use a custom clustering algorithm named SArF [53], that aims at identifying software subsystems without the need for human intervention. Escobar, Cardenas, Amarillo, Castro, Garces, Parra, & Casallas [17] also use a custom clustering algorithm, detecting optimal microservices based on a meta-model of the class hierarchy.

Agarwal, Sinha, Sridhara, Das, Desai, Tamilselvam, Singhee, & Nakamuro [25] propose an algorithm based on seed expansion. The seed classes are detected by using formal concept analysis. Then, using a seed expansion algorithm, clusters are created around the seeds by pulling in related code artefacts based on implementation structure of the software system [25].

Eyitemi & Reiff-Marganiec [42] use a rule-based approach to microservice candidate identification. The 6 proposed rules are based on the principles of high cohesion and low coupling, and using a step-based protocol can be used to manually decompose a monolithic system into microservices.

5.3.3. Metrics

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

Metric	Publications
Cohesion	[8], [11]–[14], [20]–[24], [27]–[29], [31], [39], [40], [33], [15], [34], [16], [35]–[37]
Coupling	[8], [11], [13], [20]–[25], [27], [29], [31], [39], [40], [33], [15], [34], [16], [35]–[37]
Network overhead	[13], [14], [21], [23]
Complexity	[12], [29], [31], [16]
CPU and memory usage	[14], [40], [35]
Modularity	[11], [22], [39], [27], [33], [34], [36]
Other metrics	[9]–[11], [18]–[20], [24], [31], [32], [41], [15], [34], [16], [35]–[38]
No metrics	[17], [26], [30], [42]

Table 6: Quality metrics

5.3.3.1. 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 [54]. As object-oriented programming became more popular, the concepts of cohesion and coupling were adapted to the new paradigm [55].

Throughout the years, many definitions of cohesion and coupling have been proposed both for procedural and object-oriented systems. For example, Briand et al. [56] define cohesion as the tightness with which related program features are grouped together, and coupling as the amount of relationships between the elements belonging to different modules of a system.

The publications in this review use different definitions for cohesion and coupling, and different methods of calculating them. For example, Selmadji, Seriai, Bouziane, Oumarou Mahamane, Zaragoza, & Dony [20] define (internal) cohesion as the number of direct connections between the methods of the classes belonging to a microservice over the number of possible connections

between the methods of the classes. The authors then define internal coupling as the number of direct method calls between two classes over the total number of method calls in the application.

Others [21], [23], [13], [40], [25], [22] use a similar definition of cohesion, but they define (individual) coupling as the number of method calls from a microservice class to another class outside of the service boundary. The total coupling of the solution is the sum of the coupling of all microservices. Similarly, Filippone, Qaisar Mehmood, Autili, Rossi, & Tivoli [24] define average cohesion and average coupling as ratio of the total cohesion and coupling respectively, to the number of microservices in the decomposition.

Jin, Liu, Cai, Kazman, Mo, & Zheng [39] introduce the concept of inter-service cohesion and inter-call percentage (ICP) as coupling metrics. Several other publications use the metrics introduced by W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng in their research [27], [33]–[35].

Another approach to cohesion and coupling is that of Santos & Silva [29] and Lourenço & Silva [31], who define cohesion as the percentage of entities accessed by a functionality. If all entities belonging to a microservice candidate are accessed each time a microservice candidate is accessed, the service is strongly cohesive. Coupling is defined as the percentage of the entities exposed by a microservice candidate that are accessed by other microservice candidates.

Al-Debagy & Martinek [12] use the inverse of cohesion as a metric, named lack of cohesion (LCOM). It is calculated by the number of times a microservice uses a method from another microservice, divided by the number of operations multiplied by the number of unique parameters. This metric quantifies how the operations in a service are related to each other in terms of functionality.

5.3.3.2. Network overhead

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

Filippone, Autili, Rossi, & Tivoli [23] and others [21], [13] calculate the value based using a heuristic function that uses the size of primitive types of method call arguments to predict the total network overhead of a microservice decomposition. Carvalho, Garcia, Colanzi, Assuncao, Pereira, Fonseca, Ribeiro, De Lima, & Lucena [21] also includes the protocol overhead in the calculation, which is the cost of the communication protocol used to send messages between services (for example, TCP headers, HTTP headers, etc.).

Quattrocchi, Cocco, Staffa, Margara, & Cugola [14] measure network overhead as part of their operational cost metric. The metric also includes data management costs (CPU and memory).

5.3.3.3. Complexity

The complexity of a microservice candidate is another metric that can impact the quality of the microservice decomposition. Al-Debagy & Martinek [12] defines complexity based on Number of Operations, a metric that uses Weighted Methods per Class (WMC), summing the number of methods in a class.

Santos & Silva [29] define the complexity metric in terms of the functionality redesign effort, rather than the complexity of the microservice candidates. The metric is associated with the cognitive load of the software architect when considering a migration from monolith to microservice.

In another publication by the same co-author, Lourenço & Silva [31] define complexity as the effort required to perform the decomposition, and expand the concept to uniform complexity, which is calculated by dividing the complexity of a decomposition by the maximum possible complexity.

5.3.3.4. 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, Liu, Dai, Chen, & Cao [40] use this metric to evaluate the quality of the microservice decomposition, by predicting the average CPU and memory usage of the microservices. The prediction is made based on performance logs collected by executing the monolith application.

Quattrocchi, Cocco, Staffa, Margara, & Cugola [14] define operational costs as metric to minimize, which includes communication (network) and data management (CPU and memory) costs.

Nitin, Asthana, Ray, & Krishna [35] don't utilize the CPU and memory usage directly as a metric, but instead assume the latency and throughput as indicators of performance.

5.3.3.5. Modularity

Modularity is a measure of independence of services, and can be divided into many dimensions, such as structure, concept, history, and dynamism [57].

Jin, Liu, Cai, Kazman, Mo, & Zheng [39] use modularity as a metric to evaluate potential decompositions. The authors use Modularity Quality [58] and extend the concept with structural and conceptual dependencies to assess the modularity of microservice candidates.

Carvalho, Garcia, Colanzi, Assuncao, Pereira, Fonseca, Ribeiro, De Lima, & Lucena [21] introduce a metric named feature modularization, which maps a list of features supplied by the software architect onto classes and methods, determining the set of predominant features per microservice.

5.3.3.6. Other metrics

Lourenço & Silva [31] introduce the concept of Team Size Reduction (TSR), which indicates if the average team size is shorter after the decomposition, by comparing the average number of authors

per microservice to the total number of authors. A Team Size Reduction value of 1 indicates that the microservices architecture has the same number of authors as the monolith, while a value less than 1 indicates a reduction in the number of authors. Mazlami, Cito, & Leitner [18] make use of the TSR metric, as well as the Average Domain Redundancy (ADR) metric, which represents the amount of domain-specific duplication or redundancy between the microservices. The ADR metric uses a scale from 0 to 1, where 0 indicates no redundancy and 1 indicates that all microservices are redundant.

Carvalho, Garcia, Colanzi, Assuncao, Pereira, Fonseca, Ribeiro, De Lima, & Lucena [21] propose a metric called reuse, which measures the reusability of a microservice. Reuse is calculated as the number of times a microservice is called by the user, relying on dynamic analysis to collect this information.

The usage metric of an object-oriented software system, defined as the sum of the inheritance factor (is-a) and the composition factor (has-a) is used by Bandara & Perera [22] as a part of the fitness function for the clustering algorithm.

Saidi, Tissaoui, & Faiz [10] use the intra-domain and inter-domain data dependency metrics to delineate microservice boundaries, based on the read and write access pattern of the operations. In a similar fashion, Selmadji, Seriai, Bouziane, Oumarou Mahamane, Zaragoza, & Dony [20] talk about data autonomy determined by the internal and external data access of a microservice candidate.

Kamimura, Yano, Hatano, & Matsuo [19] introduce a metric called dedication score, which measures the relationships between services as a function of access frequency. Along with a modularity metric, the dedication score is used in their custom SArF dependency-based clustering algorithm [53].

The correlation metric is used by Yang, Wu, & Zhang [9] 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, Lu, & Li [41] use the Adjusted Rand Index (ARI) as clustering evaluation criterion. The metric measures the similarity between two clusters in a decomposition, and ranges from -1 to 1, with 0 being the optimal value.

Hao, Zhao, & Li [32] use the Matching Degree metric as quality indicator. The metric is calculated by dividing the number of intersections of database tables in a given microservice and a given cluster by the total number of tables used in the microservice.

Hasan, Osman, Admodisastro, & Muhammad [16] and Kalia, Xiao, Krishna, Sinha, Vukovic, & Banerjee [36] use the Size metric to evaluate the quality of the microservice decomposition. The metric measures how evenly the size of the proposed microservices is. The size metric was originallly proposed by Wu et al. [59].

Santos & Paula [38] use the silhouette coefficient originally proposed by Rousseeuw [60] as evaluation metric. The silhouette coefficient assesses clustering consistency by comparing the average dissimilarity within the cluster.

5.3.3.7. No metrics

Finally, some of the publications 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 & Reiff-Marganiec [42], Romani, Tibermacine, & Tibermacine [30], Amiri [7], and Escobar, Cardenas, Amarillo, Castro, Garces, Parra, & Casallas [17].

The evaluation method by Kinoshita & Kanuka [26] also does not rely on quantifying the quality of the microservice decomposition using metrics, but rather relies on the software architect's judgement to choose a qualitative decomposition.

5.3.4. Results

5.3.5. Conclusion

6. Proposed solution

In this chapter, we analyze and design our solution for identification of microservice candidates. We start by identifying the functional and non-functional requirements for the solution.

Then, we propose a microservice decomposition approach using the three-step process by Abdellatif et al. [61].

- **Collect**: the necessary data is collected from the application and its environment.
- **Decomposition**: using the collected data, a decomposition of the application into microservices is proposed.
- **Analysis**: the proposed decomposition is analyzed to evaluate the effectiveness of the chosen approach.

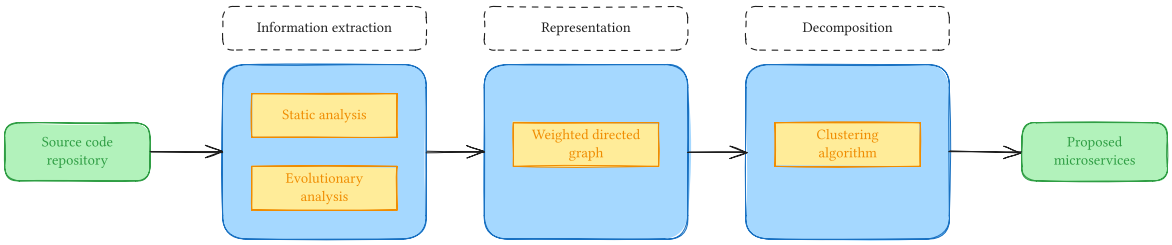

Figure 2: Overview of the architecture of the proposed solution

6.1. Requirements

6.2. Collection

6.3. Decomposition

6.4. Analysis

7. Case study

7.1. Background

7.2. Experimental setup

7.3. Evaluation and results

7.4. Discussion

7.5. Threats to validity

8. Conclusion

8.1. Future work

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] B. Kitchenham and S. Charters, “Guidelines for performing Systematic Literature Reviews in Software Engineering.” 2007.

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

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

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

[6] D. Bajaj, U. Bharti, A. Goel, and S. C. Gupta, “A Prescriptive Model for Migration to Microservices Based on SDLC Artifacts,” *Journal of Web Engineering*, Jun. 2021, doi: 10.13052/jwe1540-9589.20312.

[7] M. J. Amiri, “Object-Aware Identification of Microservices,” in *2018 IEEE International Conference on Services Computing (SCC)*, San Francisco, CA, USA: IEEE, Jul. 2018, pp. 253–256. doi: 10.1109/SCC.2018.00042.

[8] M. Daoud, A. El Mezouari, N. Faci, D. Benslimane, Z. Maamar, and A. El Fazziki, “Towards an Automatic Identification of Microservices from Business Processes,” in *2020 IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, Bayonne, France: IEEE, Sep. 2020, pp. 42–47. doi: 10.1109/WETICE49692.2020.00017.

[9] Z. Yang, S. Wu, and C. Zhang, “A Microservices Identification Approach Based on Problem Frames,” in *2022 IEEE 2nd International Conference on Software Engineering and Artificial Intelligence (SEAI)*, Xiamen, China: IEEE, Jun. 2022, pp. 155–159. doi: 10.1109/SEAI55746.2022.9832106.

[10] M. Saidi, A. Tissaoui, and S. Faiz, “A DDD Approach Towards Automatic Migration To Microservices,” in *2023 IEEE International Conference on Advanced Systems and Emergent Technologies (IC_ASET)*, Hammamet, Tunisia: IEEE, Apr. 2023, pp. 1–6. doi: 10.1109/IC_ASET58101.2023.10150522.

[11] Y. Li, Y. Zhang, Y. Yang, W. Wang, and Y. Yin, “RM2MS: A Tool for Automatic Identification of Microservices from Requirements Models,” in *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, Västerås, Sweden: IEEE, Oct. 2023, pp. 50–54. doi: 10.1109/MODELS-C59198.2023.00018.

[12] O. Al-Debagy and P. Martinek, “Extracting Microservices’ Candidates from Monolithic Applications: Interface Analysis and Evaluation Metrics Approach,” in *2020 IEEE 15th*

International Conference of System of Systems Engineering (SoSE), Budapest, Hungary: IEEE, Jun. 2020, pp. 289–294. doi: 10.1109/SoSE50414.2020.9130466.

[13] X. Zhou and J. Xiong, "Automated Microservice Identification from Design Model," 2022.

[14] G. Quattrocchi, D. Cocco, S. Staffa, A. Margara, and G. Cugola, "Cromlech: Semi-Automated Monolith Decomposition Into Microservices," *IEEE Transactions on Services Computing*, pp. 1–16, 2024, doi: 10.1109/TSC.2024.3354457.

[15] Y. Wei, Y. Yu, M. Pan, and T. Zhang, "A Feature Table Approach to Decomposing Monolithic Applications into Microservices," in *12th Asia-Pacific Symposium on Internetware*, Singapore Singapore: ACM, Nov. 2020, pp. 21–30. doi: 10.1145/3457913.3457939.

[16] M. H. Hasan, M. H. Osman, N. I. Admodisastro, and M. S. Muhammad, "AI-based Quality-driven Decomposition Tool for Monolith to Microservice Migration," in *Proceedings of the 2023 4th Asia Service Sciences and Software Engineering Conference*, Aizu-Wakamatsu City Japan: ACM, Oct. 2023, pp. 181–191. doi: 10.1145/3634814.3634839.

[17] D. Escobar *et al.*, "Towards the Understanding and Evolution of Monolithic Applications as Microservices," in *2016 XLII Latin American Computing Conference (CLEI)*, Valparaíso, Chile: IEEE, Oct. 2016, pp. 1–11. doi: 10.1109/CLEI.2016.7833410.

[18] G. Mazlami, J. Cito, and P. Leitner, "Extraction of Microservices from Monolithic Software Architectures," in *2017 IEEE International Conference on Web Services (ICWS)*, Honolulu, HI, USA: IEEE, Jun. 2017, pp. 524–531. doi: 10.1109/ICWS.2017.61.

[19] M. Kamimura, K. Yano, T. Hatano, and A. Matsuo, "Extracting Candidates of Microservices from Monolithic Application Code," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, Nara, Japan: IEEE, Dec. 2018, pp. 571–580. doi: 10.1109/APSEC.2018.00072.

[20] A. Selmadji, A.-D. Seriai, H. L. Bouziane, R. Oumarou Mahamane, P. Zaragoza, and C. Dony, "From Monolithic Architecture Style to Microservice One Based on a Semi-Automatic Approach," in *2020 IEEE International Conference on Software Architecture (ICSA)*, Salvador, Brazil: IEEE, Mar. 2020, pp. 157–168. doi: 10.1109/ICSA47634.2020.00023.

[21] L. Carvalho *et al.*, "On the Performance and Adoption of Search-Based Microservice Identification with toMicroservices," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Adelaide, Australia: IEEE, Sep. 2020, pp. 569–580. doi: 10.1109/ICSME46990.2020.00060.

[22] C. Bandara and I. Perera, "Transforming Monolithic Systems to Microservices - An Analysis Toolkit for Legacy Code Evaluation," in *2020 20th International Conference on Advances in ICT for Emerging Regions (ICTer)*, Colombo, Sri Lanka: IEEE, Nov. 2020, pp. 95–100. doi: 10.1109/ICTer51097.2020.9325443.

[23] G. Filippone, M. Autili, F. Rossi, and M. Tivoli, "Migration of Monoliths through the Synthesis of Microservices Using Combinatorial Optimization," in *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Wuhan, China: IEEE, Oct. 2021, pp. 144–147. doi: 10.1109/ISSREW53611.2021.00056.

[24] G. Filippone, N. Qaisar Mehmood, M. Autili, F. Rossi, and M. Tivoli, "From Monolithic to Microservice Architecture: An Automated Approach Based on Graph Clustering and Combinatorial Optimization," in *2023 IEEE 20th International Conference on Software Architecture (ICSA)*, L'Aquila, Italy: IEEE, Mar. 2023, pp. 47–57. doi: 10.1109/ICSA56044.2023.00013.

[25] S. Agarwal *et al.*, "Monolith to Microservice Candidates Using Business Functionality Inference," in *2021 IEEE International Conference on Web Services (ICWS)*, Chicago, IL, USA: IEEE, Sep. 2021, pp. 758–763. doi: 10.1109/ICWS53863.2021.00104.

[26] T. Kinoshita and H. Kanuka, "Automated Microservice Decomposition Method as Multi-Objective Optimization," in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, Honolulu, HI, USA: IEEE, Mar. 2022, pp. 112–115. doi: 10.1109/ICSA-C54293.2022.00028.

[27] S. Wu and C. Zhang, "Identification of Microservices through Processed Dynamic Traces and Static Calls," in *2022 3rd International Conference on Computer Science and Management Technology (ICCSMT)*, Shanghai, China: IEEE, Nov. 2022, pp. 304–308. doi: 10.1109/ICCSMT58129.2022.00071.

[28] P. Zaragoza, A.-D. Seriai, A. Seriai, A. Shatnawi, and M. Derras, "Leveraging the Layered Architecture for Microservice Recovery," in *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, Honolulu, HI, USA: IEEE, Mar. 2022, pp. 135–145. doi: 10.1109/ICSA53651.2022.00021.

[29] S. Santos and A. R. Silva, "Microservices Identification in Monolith Systems: Functionality Redesign Complexity and Evaluation of Similarity Measures," *Journal of Web Engineering*, Aug. 2022, doi: 10.13052/jwe1540-9589.2158.

[30] Y. Romani, O. Tibermacine, and C. Tibermacine, "Towards Migrating Legacy Software Systems to Microservice-based Architectures: A Data-Centric Process for Microservice Identification," in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, Honolulu, HI, USA: IEEE, Mar. 2022, pp. 15–19. doi: 10.1109/ICSA-C54293.2022.00010.

[31] J. Lourenço and A. R. Silva, "Monolith Development History for Microservices Identification: A Comparative Analysis," in *2023 IEEE International Conference on Web Services (ICWS)*, Chicago, IL, USA: IEEE, Jul. 2023, pp. 50–56. doi: 10.1109/ICWS60048.2023.00019.

[32] J. Hao, J. Zhao, and Y. Li, "Research on Decompostion Method of Relational Database Oriented to Microservice Refactoring," 2023.

[33] M. Brito, J. Cunha, and J. Saraiva, "Identification of Microservices from Monolithic Applications through Topic Modelling," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, Virtual Event Republic of Korea: ACM, Mar. 2021, pp. 1409–1418. doi: 10.1145/3412841.3442016.

[34] K. Sellami, M. A. Saied, and A. Ouni, "A Hierarchical DBSCAN Method for Extracting Microservices from Monolithic Applications," in *The International Conference on Evaluation and Assessment in Software Engineering 2022*, Gothenburg Sweden: ACM, Jun. 2022, pp. 201–210. doi: 10.1145/3530019.3530040.

[35] V. Nitin, S. Asthana, B. Ray, and R. Krishna, "CARGO: AI-Guided Dependency Analysis for Migrating Monolithic Applications to Microservices Architecture," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, Rochester MI USA: ACM, Oct. 2022, pp. 1–12. doi: 10.1145/3551349.3556960.

[36] A. K. Kalia, J. Xiao, R. Krishna, S. Sinha, M. Vukovic, and D. Banerjee, "Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Athens Greece: ACM, Aug. 2021, pp. 1214–1224. doi: 10.1145/3468264.3473915.

[37] L. Cao and C. Zhang, "Implementation of Domain-oriented Microservices Decomposition Based on Node-attributed Network," in *2022 11th International Conference on Software and Computer Applications*, Melaka Malaysia: ACM, Feb. 2022, pp. 136–142. doi: 10.1145/3524304.3524325.

[38] A. Santos and H. Paula, "Microservice Decomposition and Evaluation Using Dependency Graph and Silhouette Coefficient," in *15th Brazilian Symposium on Software Components, Architectures, and Reuse*, Joinville Brazil: ACM, Sep. 2021, pp. 51–60. doi: 10.1145/3483899.3483908.

[39] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng, "Service Candidate Identification from Monolithic Systems Based on Execution Traces," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 987–1007, May 2021, doi: 10.1109/TSE.2019.2910531.

[40] Y. Zhang, B. Liu, L. Dai, K. Chen, and X. Cao, "Automated Microservice Identification in Legacy Systems with Functional and Non-Functional Metrics," in *2020 IEEE International Conference on Software Architecture (ICSA)*, Salvador, Brazil: IEEE, Mar. 2020, pp. 135–145. doi: 10.1109/ICSA47634.2020.00021.

[41] S.-P. Ma, T.-W. Lu, and C.-C. Li, “Migrating Monoliths to Microservices Based on the Analysis of Database Access Requests,” in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, Newark, CA, USA: IEEE, Aug. 2022, pp. 11–18. doi: 10.1109/SOSE55356.2022.00008.

[42] F.-D. Eyitemi and S. Reiff-Marganiec, “System Decomposition to Optimize Functionality Distribution in Microservices with Rule Based Approach,” in *2020 IEEE International Conference on Service Oriented Systems Engineering (SOSE)*, Oxford, United Kingdom: IEEE, Aug. 2020, pp. 65–71. doi: 10.1109/SOSE49046.2020.00015.

[43] “IEEE Guide for Software Requirements Specifications.” pp. 1–26, 1984.

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

[45] 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.

[46] 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.

[47] 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.

[48] 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.

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

[50] 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.

[51] 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.

[52] X. Zhu and Z. Ghahramani, “Learning from Labeled and Unlabeled Data with Label Propagation,” p. , 2003.

[53] 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.

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

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

[56] 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.

[57] 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.

[58] 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.

[59] J. Wu, A. Hassan, and R. Holt, "Comparison of Clustering Algorithms in the Context of Software Evolution," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, Budapest, Hungary: IEEE, 2005, pp. 525–535. doi: 10.1109/ICSM.2005.31.

[60] 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.

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

[62] Y. Abgaz *et al.*, "Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review," *IEEE Transactions on Software Engineering*, vol. 49, no. 8, pp. 4213–4242, Aug. 2023, doi: 10.1109/TSE.2023.3287297.

[63] I. Oumoussa and R. Saidi, "Evolution of Microservices Identification in Monolith Decomposition: A Systematic Review," vol. 12, 2024.

[64] R. A. Schmidt and M. Thiry, "Microservices Identification Strategies : A Review Focused on Model-Driven Engineering and Domain Driven Design Approaches," in *2020 15th Iberian Conference on Information Systems and Technologies (CISTI)*, Sevilla, Spain: IEEE, Jun. 2020, pp. 1–6. doi: 10.23919/CISTI49556.2020.9141150.

[65] J. Kazanavicius and D. Mazeika, “Migrating Legacy Software to Microservices Architecture,” in *2019 Open Conference of Electrical, Electronic and Information Sciences (eStream)*, Vilnius, Lithuania: IEEE, Apr. 2019, pp. 1–5. doi: 10.1109/eStream.2019.8732170.

[66] A. Mparmpoutis and G. Kakarontzas, “Using Database Schemas of Legacy Applications for Microservices Identification: A Mapping Study,” in *Proceedings of the 6th International Conference on Algorithms, Computing and Systems*, Larissa Greece: ACM, Sep. 2022, pp. 1–7. doi: 10.1145/3564982.3564995.

[67] L. Baresi, M. Garriga, and A. De Renzis, “Microservices Identification Through Interface Analysis,” *Service-Oriented and Cloud Computing*, vol. 10465. Springer International Publishing, Cham, pp. 19–33, 2017. doi: 10.1007/978-3-319-67262-5_2.

[68] S. Eski and F. Buzluca, “An Automatic Extraction Approach: Transition to Microservices Architecture from Monolithic Application,” in *Proceedings of the 19th International Conference on Agile Software Development: Companion*, Porto Portugal: ACM, May 2018, pp. 1–6. doi: 10.1145/3234152.3234195.

[69] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, “From Monolith to Microservices: A Classification of Refactoring Approaches,” *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, vol. 11350. Springer International Publishing, Cham, pp. 128–141, 2019. doi: 10.1007/978-3-030-06019-0_10.

A Systematic Literature Review (SLR) publications

Primary studies

	Publication
1	*A DDD Approach Towards Automatic Migration To Microservices*, Saidi, Tissaoui, & Faiz [10]
2	*A Microservices Identification Approach Based on Problem Frames*, Yang, Wu, & Zhang [9]
3	*Automated Microservice Decomposition Method as Multi-Objective Optimization*, Kinoshita & Kanuka [26]
4	*Automated Microservice Identification from Design Model*, Zhou & Xiong [13]
5	*Automated Microservice Identification in Legacy Systems with Functional and Non-Functional Metrics*, Zhang, Liu, Dai, Chen, & Cao [40]
6	*Cromlech: Semi-Automated Monolith Decomposition Into Microservices*, Quattrocchi, Cocco, Staffa, Margara, & Cugola [14]
7	*Extracting Candidates of Microservices from Monolithic Application Code*, Kamimura, Yano, Hatano, & Matsuo [19]
8	*Extracting Microservices' Candidates from Monolithic Applications: Interface Analysis and Evaluation Metrics Approach*, Al-Debagy & Martinek [12]
9	*Extraction of Microservices from Monolithic Software Architectures*, Mazlami, Cito, & Leitner [18]
10	*From Monolithic Architecture Style to Microservice One Based on a Semi-Automatic Approach*, Selmadji, Seriai, Bouziane, Oumarou Mahamane, Zaragoza, & Dony [20]
11	*From Monolithic to Microservice Architecture: An Automated Approach Based on Graph Clustering and Combinatorial Optimization*, Filippone, Qaisar Mehmood, Autili, Rossi, & Tivoli [24]
12	*Identification of Microservices through Processed Dynamic Traces and Static Calls*, Wu & Zhang [27]
13	*Leveraging the Layered Architecture for Microservice Recovery*, Zaragoza, Seriai, Seriai, Shatnawi, & Derras [28]

14	*Microservices Identification in Monolith Systems: Functionality Redesign Complexity and Evaluation of Similarity Measures*, Santos & Silva [29]
15	*Migrating Monoliths to Microservices Based on the Analysis of Database Access Requests*, Ma, Lu, & Li [41]
16	*Migration of Monoliths through the Synthesis of Microservices Using Combinatorial Optimization*, Filippone, Autili, Rossi, & Tivoli [23]
17	*Monolith Development History for Microservices Identification: A Comparative Analysis*, Lourenço & Silva [31]
18	*Monolith to Microservice Candidates Using Business Functionality Inference*, Agarwal, Sinha, Sridhara, Das, Desai, Tamilselvam, Singhee, & Nakamuro [25]
19	*Object-Aware Identification of Microservices*, Amiri [7]
20	*On the Performance and Adoption of Search-Based Microservice Identification with toMicroservices*, Carvalho, Garcia, Colanzi, Assuncao, Pereira, Fonseca, Ribeiro, De Lima, & Lucena [21]
21	*Research on Decompostion Method of Relational Database Oriented to Microservice Refactoring*, Hao, Zhao, & Li [32]
22	*RM2MS: A Tool for Automatic Identification of Microservices from Requirements Models*, Li, Zhang, Yang, Wang, & Yin [11]
23	*Service Candidate Identification from Monolithic Systems Based on Execution Traces*, Jin, Liu, Cai, Kazman, Mo, & Zheng [39]
24	*System Decomposition to Optimize Functionality Distribution in Microservices with Rule Based Approach*, Eyitemi & Reiff-Marganiec [42]
25	*Towards an Automatic Identification of Microservices from Business Processes*, Daoud, El Mezouari, Faci, Benslimane, Maamar, & El Fazziki [8]
26	*Towards Migrating Legacy Software Systems to Microservice-based Architectures: A Data-Centric Process for Microservice Identification*, Romani, Tibermacine, & Tibermacine [30]
27	*Towards the Understanding and Evolution of Monolithic Applications as Microservices*, Escobar, Cardenas, Amarillo, Castro, Garces, Parra, & Casallas [17]

28	*Transforming Monolithic Systems to Microservices - An Analysis Toolkit for Legacy Code Evaluation*, Bandara & Perera [22]
29	*A Feature Table Approach to Decomposing Monolithic Applications into Microservices*, Wei, Yu, Pan, & Zhang [15]
30	*A Hierarchical DBSCAN Method for Extracting Microservices from Monolithic Applications*, Sellami, Saied, & Ouni [34]
31	*AI-based Quality-driven Decomposition Tool for Monolith to Microservice Migration*, Hasan, Osman, Admodisastro, & Muhammad [16]
32	*CARGO: AI-Guided Dependency Analysis for Migrating Monolithic Applications to Microservices Architecture*, Nitin, Asthana, Ray, & Krishna [35]
33	*Implementation of Domain-oriented Microservices Decomposition Based on Node-attributed Network*, Cao & Zhang [37]
34	*Microservice Decomposition and Evaluation Using Dependency Graph and Silhouette Coefficient*, Santos & Paula [38]
35	*Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices*, Kalia, Xiao, Krishna, Sinha, Vukovic, & Banerjee [36]

Table 7: Selected publications (primary studies)

Secondary studies

	Publication
1	*A Prescriptive Model for Migration to Microservices Based on SDLC Artifacts*, Bajaj, Bharti, Goel, & Gupta [6]
2	*Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review*, Abgaz et al. [62]
3	*Evolution of Microservices Identification in Monolith Decomposition: A Systematic Review*, Oumoussa & Saidi [63]
4	*Microservices Identification Strategies : A Review Focused on Model-Driven Engineering and Domain Driven Design Approaches*, Schmidt & Thiry [64]
5	*Migrating Legacy Software to Microservices Architecture*, Kazanavicius & Mazeika [65]
6	*Using Database Schemas of Legacy Applications for Microservices Identification: A Mapping Study*, Mparmpoutis & Kakarontzas [66]

Table 8: Selected publications (secondary studies)

B Examples

Examples First page

```ruby
class Example
  def initialize
    @text = "Hello world"
  end

  def say_hello
    puts @text
  end
end

example = Example.new

example.say_hello
# => Hello world
```

Listing 2: Ruby code example

Examples Second page