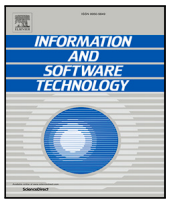
Information and Software Technology 183 (2025) 107732

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Reengineering software systems into microservices: State-of-the-art and future directions

Thakshila Imiya Mohottige [a,*], Artem Polyvyanyy [a], Colin Fidge [b], Rajkumar Buyya [a], Alistair Barros [b]

[a] The University of Melbourne, Australia
[b] Queensland University of Technology, Australia

ARTICLE INFO

Keywords:
Microservices
Software reengineering
Software architecture

ABSTRACT

Context: With the acknowledged benefits of microservices architectures, such as scalability, flexibility, improved maintenance, and deployment, legacy software systems are increasingly being reengineered into microservices. Recently, a plethora of methods, techniques, tools, and evaluation criteria for reengineering software systems into microservices have been proposed without being systematized.

Objectives: The objective of this work is to conduct an in-depth systematic literature review to identify and analyze methods, techniques, and tools for reengineering software systems into microservices and the ways for evaluating such reengineering initiatives and their results.

Methods: A systematic literature review of works on reengineering software systems into microservices was performed, yielding 117 primary studies. The review focused on addressing key research questions concerning the evolution of microservices reengineering, methodologies employed, tools available, and the challenges faced in the reengineering process. We used a taxonomy development method to systematize knowledge in these areas.

Results: The analysis revealed multiple reengineering approaches: static, dynamic, hybrid, and artifact-driven. Significant evaluation criteria identified include coupling, cohesion, and modularity. Key paradigms for microservices reengineering, such as domain-driven design and interface analysis, were identified and discussed. The study also highlights that incremental and iterative transitions are favored in practice.

Conclusion: This study provides a structured overview of the current state of research on reengineering software systems into microservices. It highlights challenges in existing reengineering methodologies. Future directions include validating behavioral equivalence of original and reengineered systems, automating microservices generation, and refining database layer partitioning. The findings emphasize the need for further work to enhance the reengineering process and evaluation of the transition between monolithic and microservices architectures.

Contents

1. Introduction 2
2. Systematic literature review process 3
   2.1. Research questions 4
   2.2. Search protocol and selection criteria 4
   2.3. Data extraction and synthesis 4
   2.4. Quality assessment 4
3. Results 5
   3.1. (RQ1) how did research on the reengineering of software systems into microservice-based systems develop over time? 5
   3.2. (RQ2) what approaches are used to reengineer software systems into microservice-based systems, and how are reengineered systems evaluated? 6
      3.2.1. (RQ2.1) what classes of approaches exist? 6
      3.2.2. (RQ2.2) what tools exist, and which level of automation do they support? 9

* Corresponding author.
E-mail addresses: thakshila.imiyamohottige@student.unimelb.edu.au (T.I. Mohottige), artem.polyvyanyy@unimelb.edu.au (A. Polyvyanyy), c.fidge@qut.edu.au (C. Fidge), rbuyya@unimelb.edu.au (R. Buyya), alistair.barros@qut.edu.au (A. Barros).

https://doi.org/10.1016/j.infsof.2025.107732
Received 24 September 2024; Received in revised form 27 December 2024; Accepted 18 March 2025
Available online 27 March 2025

3.2.3. (RQ2.3) which techniques/algorithms are used? ... 9
3.2.4. (RQ2.4) how is data used? ... 13
3.2.5. (RQ2.5) how are the reengineered systems evaluated? ... 14
3.3. (RQ3) what are the challenges and limitations of existing methods for reengineering software systems into microservice-based systems? ... 17
4. Discussion and future directions ... 17
4.1. Artifact-driven analysis ... 18
4.2. Static analysis ... 18
4.3. Dynamic analysis ... 18
4.4. Hybrid analysis ... 19
4.5. Database analysis ... 19
4.6. Emerging techniques ... 20
4.7. Evaluation ... 20
4.8. Paradigms ... 20
4.9. Gaps and future directions ... 21
5. Conclusion ... 21
CRediT authorship contribution statement ... 21
Declaration of competing interest ... 21
Acknowledgment ... 21
Appendix. Study list ... 21
Data availability ... 25
References ... 25

1. Introduction

Modernizing software systems is essential to obtain the benefits of the latest technical capabilities [9]. Monolithic, legacy mainframe-based software systems are an increasingly obsolete technology that suffers from scalability, maintainability, availability, and efficiency problems [10–13]. Therefore, there is an imperative need to modernize such systems to obtain better performance and improve the overall developer and user experience [14].

A wave of migration of monolithic software to object-oriented platforms was observed at the end of the previous millennium [15]. Later, service-oriented architectures (SOAs) emerged, and legacy software systems began moving towards service-oriented architectures [16]. In an SOA, software systems are modular, with distributed modules having clearly defined interfaces [17]. But these services are not independent services [18]. As opposed to the logically related operations in an SOA, the microservice architectural style emerged promising to distribute applications via fine-grained, loosely coupled, and highly cohesive autonomous components communicating via well-defined, lightweight protocols managing local, synchronized databases, achieving high scalability, availability, and efficiency [12,13].

The tightly coupled nature of legacy software systems reduces their scalability and maintainability. Often, making a change in one class affects several other classes. Hence, it increases complexity and development time [19]. Decomposing legacy systems into small independent units increases the maintainability [20]. Microservices were first discussed in 2011 [13]. In addition to addressing the aforementioned drawbacks of conventional software architectures, microservices enable independent development and deployment of services, flexibility in horizontal scaling in the cloud environment, and support for efficient development team management [21]. Due to their multiple advantages, companies like Google, Netflix, Amazon, Uber, and eBay upgraded to microservice-based systems.

Companies often have a substantial investment in their corporate business systems and cannot afford to redevelop them entirely. Instead, a legacy system can be converted into a microservice system by incrementally extracting microservices from it. This approach has several advantages. Firstly, it makes the best use of the company's existing investment in the original system, which is often considerable and spans several decades. Secondly, the complexity of a legacy system and the effort, time to market, and resource constraints (e.g., human resources) required to reimplement it from scratch can be prohibitive. Finally, only certain system parts may be suitable for migration, while others cannot benefit from or even will degrade when moved to the new architectural style. For example, functionality that is infrequently used, such as annual financial reporting, is probably best implemented in the head office's mainframe. Hence, the ability to extract specific services for reengineering and redeployment as microservices while leaving other functionalities unchanged is essential.

Several studies [1–8] have been conducted to review the works on microservices identification. Schmidt and Thiry [1] reviewed model-driven engineering and domain-driven analysis approaches to identify potential microservices. Schröer et al. [2] analyzed the techniques for identifying microservices during the requirement analysis and design phases with the evaluation techniques of identified microservices. Cojocaru et al. [3] discussed the quality assessment criteria for microservices automatically decomposed from monolithic applications. Quality-driven approaches in migration, quality attributes analysis, and quality-driven process implementation were reviewed by Capuano and Muccini [4]. Ponce et al. [5] conducted a rapid review study of migration techniques, the types of systems to which the proposed techniques are applied, methods for validating the migration techniques, and the challenges associated with such migrations. Fritzsch et al. [6] analyzed existing architectural refactoring approaches in the context of decomposing a monolithic application architecture into microservices and how they can be classified concerning the techniques and strategies used. The approaches to modernizing legacy software were discussed by Wolfart et al. [7]. They defined a road map for modernizing legacy systems with microservices that includes motivations, understanding and decomposing legacy systems, execution, validation, monitoring, and infrastructure aspects of the modernizing process. A taxonomy of service identification approaches that combine the inputs used for service identification, the process followed, the output of service identification, and the usability of service identification was developed by Abdellatif et al. [8].

Existing studies have been limited both in scope and in the number of reviewed works. The various aspects of redesigning monolithic software systems by extracting discrete functions from them that could be re-implemented as microservices, including service discovery approaches and techniques, tools that support reengineering, data used to inform migration processes, evaluation methods for the resulting microservice systems, and challenges and limitations of the existing reengineering approaches were not in the focus of previous studies. A comparison of existing studies is provided in Table 1. Thus, our research herein aims to provide a comprehensive review of previous studies, contribute to a better understanding of microservice discovery techniques regarding software architectural properties, and recommend

Table 1
A comparison of existing literature reviews on reengineering of software systems into microservices.

Literature review	LR [1]	LR [2]	LR [3]	LR [4]	LR [5]	LR [6]	LR [7]	LR [8]	This study	Research question(s) addressed in this study
Review period/year	2013–2019	2020	1998–2018	2016–2022	2019	2018	2020	2019	2023	
Number of reviewed papers	27	31	29	58	20	10	62	41	117	
Comparison of research questions										
What are the techniques/approaches/patterns for legacy software reengineering?	◐	●	○	○	●	●	●	◐	●	RQ2.1 & RQ2.3
What types of systems have the existing reengineering techniques been applied to?	○	○	○	○	●	●	○	○	●	RQ2.1
What tools are used for reengineering monolithic systems into microservices?	○	○	○	○	○	○	○	○	●	RQ2.2
What inputs/outputs are used by the existing reengineering techniques?	○	○	○	○	○	○	●	◐	●	RQ2.4
What driving forces/evaluation criteria are used for the identified microservices?	○	●	○	○	○	○	●	◐	●	RQ2.5
How reengineering processes/techniques are validated?	○	○	○	○	●	○	○	○	●	RQ2.5
What quality-driven/assessment criteria are used for reengineering?	○	○	●	●	○	○	○	○	●	RQ2.5
What quality attributes are analyzed, and how have they been implemented for reengineering?	○	○	●	●	○	○	○	◐	●	RQ2.5
What are the challenges of reengineering legacy software systems into microservices?	○	○	○	○	●	○	○	○	●	RQ3
What usability aspects, advantages, and disadvantages/ limitations are highlighted?	○	○	○	○	○	○	○	◐	●	RQ3
What are the roles and responsibilities involved in the identification of microservices?	●	○	○	○	○	○	○	○	○	N/A

● Addressed ◐ Partially addressed ○ Not addressed.

future research directions for migrating monolithic software systems to microservices architectures.

Our study below is based on 117 papers. It reveals that static (44%), dynamic (12%), hybrid (12%), and artifact-driven (32%) techniques are the major classes of approaches for microservices identification and extraction. Source code structure analysis that involves inheritance attributes and structural interactions analysis is a widely used static analysis technique. Dynamic analysis, however, is an under-explored area. It often relies on instrumented logs. Hybrid approaches combine the aspects of static and dynamic techniques. Artifact-driven techniques rely on domain-driven designs (DDD) and additional software artifacts. We have further observed two main techniques for microservices identification, namely system modeling and microservices extraction. Prominent studies [19,21–38] have been identified for each class of techniques. Input/output and tools used by the studies, the level of automation, and various evaluation techniques were thoroughly reviewed to address all the aspects of microservice extraction. Moreover, core design principles, such as domain-driven design, workflow analysis, feature analysis, semantic analysis, repository analysis, interface analysis, and runtime analysis, were identified. Finally, we discuss further insights into the limitations and future directions in the area of microservices-based software system reengineering.

The remainder of the paper proceeds as follows. Section 2 describes the research methodology followed in this work. Sections 3 and 4 present and discuss the results of our literature review. Finally, Section 5 states concluding remarks.

2. Systematic literature review process

In this work, we followed the guidelines for performing a systematic literature review in software engineering proposed by Kitchenham and Charters [39] and further refined by Kitchenham and Brereton [40]. Existing literature review studies [1–8] were identified by first performing an initial search for survey and literature review papers in the area of interest and then including all additional secondary studies identified when searching for the relevant primary studies. Table 1 compares the existing literature reviews. If a study has declared a specific review period or year, it is specified in the review period/year row. Otherwise, the study year has been provided to indicate that the review period cannot go beyond that year. If a study mentions the number of reviewed papers, it is indicated in the number of reviewed papers row. The research questions listed in the first column of Table 1 are the research questions addressed in the existing studies. We merged similar research questions and rephrased them to ensure the consistent use of terminology. The table summarizes which research questions are fully, partially, or not addressed in the existing literature reviews. Review LR [1] focuses on semi-automated approaches to reengineering and, thus, partially addresses the question of what techniques/approaches/patterns are used for legacy software reengineering. Reviews LR [2], LR [3], LR [4], and LR [6] listed in Table 1 have addressed only certain aspects of the problem. Reviews LR [5], LR [7], and LR [8] are extensive literature reviews on the topic. Note that LR [5] is not a systematic literature review. The scope of LR [7] is different from our research since it is focused on defining a road map for modernizing legacy systems. Finally, Review LR [8] focuses on service identification instead of microservice identification and reengineering.

Our analysis indicates that works on the identification and reengineering of microservices reached their peak between 2020 and 2022, as shown in Fig. 2. In particular, 69% of the studies were conducted during these years. Since the majority of existing literature review studies have been conducted in or before 2020, our study has a better coverage of the relevant works. As the existing literature reviews are limited in scope, objectives, and coverage, it is, therefore, essential to analyze and systematize existing works comprehensively, spanning different techniques, system modeling approaches, and evaluation strategies to understand the state-of-the-art, research gaps, and promising avenues for future work. Hence our work seeks to address this gap.

2.1. Research questions

Our literature review was conducted to examine existing methods, techniques, and tools for reengineering software systems into microservices, understand the limitations of the existing approaches, and identify fruitful avenues for future work. Consequently, we formulated the following research questions to guide our study.

- RQ1 How did research on the reengineering of software systems into microservice-based systems develop over time?
- RQ2 What approaches are used to reengineer software systems into microservice-based systems, and how are reengineered systems evaluated?
  - RQ2.1 What classes of approaches (e.g., static and dynamic) exist?
  - RQ2.2 What tools exist and which level of automation do they support?
  - RQ2.3 Which techniques/algorithms are used?
  - RQ2.4 How is data (e.g., software logs) used?
  - RQ2.5 How are the reengineered systems evaluated?
- RQ3 What are the challenges and limitations of existing methods for reengineering software systems into microservice-based systems?

Our research questions were defined to maximize the coverage of the questions addressed in the early studies (cf. the first column in Table 1) and to understand and refine them further. The last column in Table 1 maps the research questions addressed in our work onto the questions studied elsewhere. However, our study does not consider the last question listed in the table. Due to the typical roles involved in the software development lifecycle, we excluded this aspect from our study.

2.2. Search protocol and selection criteria

All the publications analyzed in this study were retrieved from five databases widely used to index publications in the areas of computer science and software engineering: Web of Science,[1] Scopus,[2] ScienceDirect,[3] ACM Digital Library,[4] and IEEE Xplorer Digital Library.[5] These databases provide good coverage of primary sources from high-quality academic journals and peer-reviewed conferences [41].

To maximize the chances of identifying papers that can contribute to answering the research questions of this study, we used these keywords: “microservice”, “reengineer”, “redesign”, “refactor”, “rearchitect”, “migrate”, “discover”, and “identify”. The keyword “microservice” was included as the study focuses on microservices systems. Keywords such as “reengineer”, “redesign”, “refactor”, “rearchitect”, and “migrate” were selected as this work focuses on reengineering software systems into microservice-based systems. Lastly, the keywords “discover” and “identify” were added to address the objective of identifying microservices. The search query used for the Web of Science database is listed below:

(TS = (microservice* AND (reengineer* OR re-engineer* OR redesign* OR re-design* OR discover* OR identify* OR refactor* OR rearchitect* OR re-architect* OR migrate*))) AND (WC = (Computer Science)) AND (DT = (Article OR Book Chapter OR Proceedings Paper)) AND (LA = (English)).

To guide the selection of primary studies to include in our review, we defined the inclusion and exclusion criteria listed in Table 2. These criteria were applied to assess the suitability of each study for inclusion.

Fig. 1 summarizes our search process for selecting primary studies, including the number of papers identified in each stage. The initial search for relevant papers over the five databases was conducted on the 23rd of January 2023.

To ensure the full coverage of works relevant to this study on the date the search was conducted, we did not impose restrictions on the publication dates of the retrieved references. In this initial search, 4 843 references were retrieved. As a paper can be indexed by several databases, we removed duplicate references to result in 2 441 distinct references. To determine their relevance to our study, all the references were evaluated against the inclusion and exclusion criteria from Table 2 using a checklist-based scoring procedure. Papers on legacy system refactoring, requirements for refactoring, refactoring techniques, and evaluation of reengineered systems were included for further analysis. Studies not related to our research questions, for example, papers on networks and deployment of microservice-based systems, non-peer-reviewed studies, studies not related to software systems, or not in English, were excluded from further processing. At the end of this stage, 220 papers were identified as potentially relevant for our literature review. The inclusion/exclusion decisions were taken based on paper titles and abstracts. Hence, papers with unclear exclusion decisions were kept for further full text analysis. The full text read of 220 papers revealed 107 relevant studies. During the review of the papers selected for full-text analysis, relevant references were noted. These references were analyzed in the snowballing stage, and relevant works were included in the study. Both forward and backward searching on references were performed. Ten additional papers were included in the snowballing stage. Consequently, the presented search process has resulted in the identification of 117 primary studies.

2.3. Data extraction and synthesis

To systematize the knowledge extracted during the in-depth analysis of the primary studies, we followed a method for taxonomy development by Nickerson et al. [42]. It is an iterative approach to identifying concepts and their characteristics and grouping them into dimensions. The method guides the evaluation of the developed taxonomies for usefulness, like the completeness and robustness of the developed taxonomy dimensions. After defining the classification criteria compatible with the research questions, the selected primary studies were analyzed in-depth, and relevant insights were extracted and recorded in a spreadsheet for subsequent analysis.

2.4. Quality assessment

To ensure the rigor and credibility of our study, the author team provided guidance and oversight of all stages of the literature review process. Multiple review iterations were conducted to enhance the quality of decisions and minimize errors. The team collaboratively selected the digital libraries, helped refine keywords to retrieve a sufficient number of relevant papers, and helped establish the selection criteria. Additionally, the entire team reached a consensus on the classification criteria before data extraction began and reviewed the results to ensure consistency and reliability.

[1] https://clarivate.com/webofsciencegroup/solutions/web-of-science.
[2] https://www.scopus.com/.
[3] https://www.sciencedirect.com/.
[4] http://portal.acm.org/.
[5] https://ieeexplore.ieee.org/.

Table 2
Inclusion and exclusion criteria.

Criterion type	Criterion definition
Inclusion	1. Study is on legacy software system reengineering 2. Study is on requirements for reengineering of legacy software systems 3. Study is on a technique for evaluating functional consistency of a reengineered software system 4. Study is on a technique for evaluating the performance of a microservice system 5. Study is on using software logs for legacy software system reengineering 6. Study is on an approach for evaluating microservices
Exclusion	1. Study is not related to software systems 2. Study is on microservice system deployment, self-adjusting models, Quality of Service, or scalability 3. Study is on networks or load testing, security, and fault tolerance of software systems 4. Study does not present sufficient technical details to contribute to at least one research question addressed in this literature review 5. Study did not undergo a peer-review process, for example, published in a non-reviewed journal or conference papers, theses, books and book chapters, and doctoral dissertations 6. Study is a literature review 7. Study is not in English

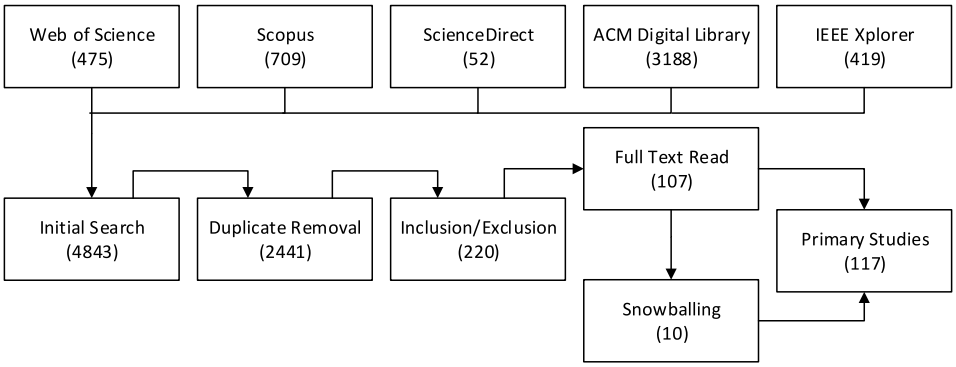


Fig. 1. Overview of the stages and results of our literature selection process.

Table 3
Paper classification details.

Type of study	Number of papers
Software system migration studies	83
Case studies and industry interviews	30
Greenfield development	4

3. Results

This section elaborates on the findings of the literature review based on the research questions. Appendix lists the primary studies selected for this literature review. Table 3 summarizes the classification of the selected 117 papers. The majority of the papers (71%) explain legacy system migration strategies, whereas most of the remainder of the papers (25%) focus on industry interviews and case studies. A small number of papers (4%) discuss greenfield development, where new system implementation in a microservice-based architecture is considered. The greenfield development was included in the analysis since it is applied in the context of artifact-based microservices extraction.

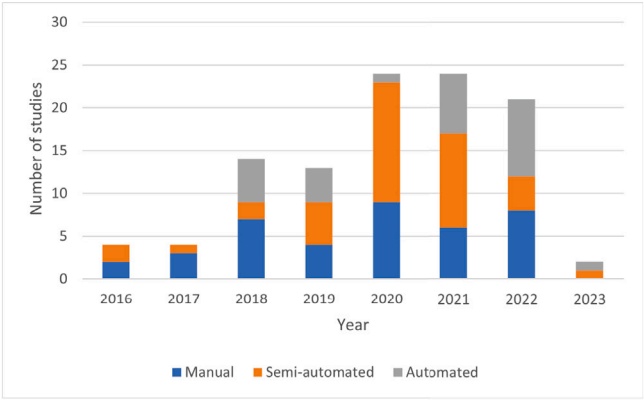
3.1. (RQ1) how did research on the reengineering of software systems into microservice-based systems develop over time?

The first study on software systems reengineering into microservices was published in 2016. Manual, semi-automated, and automated techniques for migrating systems are discussed in the literature. Manual techniques are completely human-oriented, whereas appropriate modeling, extraction, and visualization tools assist people during semi-automated system reengineering projects. In contrast, automatic techniques produce possible microservice recommendations from various inputs, e.g., source code, software logs, and software design artifacts. These recommendations can then form the basis for system reengineering.

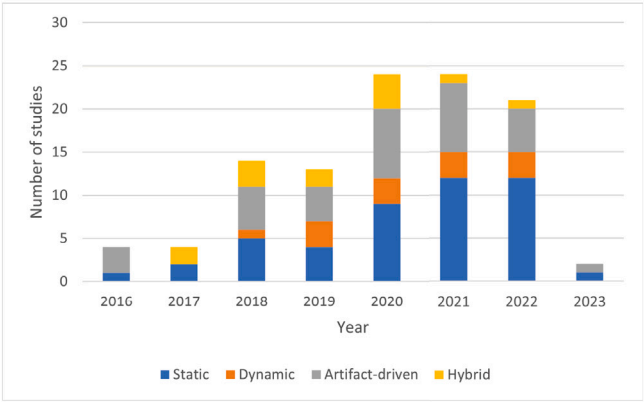
Fig. 2(a) depicts the progression of automation levels in the techniques examined across the surveyed studies over time. A significant proportion of the studies (38%) concentrated on semi-automated identification methods. Manual approaches are similarly prevalent, comprising 37% of the total studies. In recent years, there has been a notable shift towards automated approaches, which now account for the remaining 25% of the studies.

The identified approaches for decomposing software systems into microservices are classified as static, dynamic, artifact-driven, and hybrid analyses. In static analysis, program source code, database schemata, and source code repository histories are used to provide insights into the system under study. By contrast, dynamic analysis considers execution time details like software system and server event logs, and runtime monitoring. The artifact-driven approaches are based on system artifacts like UML and data flow diagrams, architectural documents, use cases and user stories, ubiquitous language, and domain models. Domain-driven design (DDD) and task-driven (functional-driven) design patterns are a subset of artificial-driven approaches. Finally, the hybrid approach can combine static, dynamic, and artifact-driven approaches.

Fig. 2(b) illustrates the numbers of different microservice identification approaches published over time. Most of the existing studies are based on static system analysis (44% of studies). The artifact-driven analysis is the second most used technique for software systems



(a) Level of automation



(b) Analysis type

Fig. 2. Number of studies over time.

reengineering (32%). The studies of dynamic and hybrid approaches are less frequent, with each approach comprising only 12% of the total studies.

3.2. (RQ2) what approaches are used to reengineer software systems into microservice-based systems, and how are reengineered systems evaluated?

In this section, we discuss the identified approaches for reengineering software systems into microservices systems.

3.2.1. (RQ2.1) what classes of approaches exist?

The approaches used to analyze monolithic applications for their reengineering into microservices systems can be broadly classified into three main categories: static, dynamic, and artifact-driven analysis. An additional hybrid approach is identified, consolidating the main approaches.

The artifact-driven approaches use software artifacts like requirements, design diagrams, UML diagrams, data flow diagrams, business processes, use cases, user stories, domain models, and other design artifacts to identify bounded contexts for microservices. Each such bounded context implements a small, highly cohesive, loosely coupled behavior [43]. These contexts are then accepted as microservice candidates.

The static analysis approaches are based on analysis of source code, database schema, and histories of source code repositories. These approaches use dependencies between classes, like inheritance, extended class relationships, similarities between classes and database tables, and dependent commits in code repositories.

In contrast, the dynamic analysis approaches use runtime information to identify microservices. For example, they use runtime monitoring, execution time data correlations, and system-generated logs.

Lastly, the hybrid analysis techniques combine principles from the approaches discussed above. Often, a hybrid approach results from extending one "pure" approach with some feature of an approach of a different type. For example, a static analysis technique can borrow ideas of software log analysis to complement its microservice identification decisions.

Fig. 3 shows categories and subcategories of the three main approaches. The leaf nodes in the figure correspond to relevant study IDs, which are detailed in Appendix. A comprehensive analysis of the categories and subcategories follows.

Artifact-driven analysis

An artifact-driven analysis uses various system representations to examine requirements, features, use cases, classes, and components of the system. The main categories of artifact-driven approaches, defined by the types of analyzed artifacts, are detailed below:

- *Domain models/languages*: Domain models and languages play a crucial role in software engineering by representing relationships between classes or entities. For example, UML diagrams provide abstract visualizations of the software system. The term domain language, also known as ubiquitous language, refers to the consistent terminology used to describe business operations and is essential for capturing terms from legacy systems [44]. Use cases describe user interactions with the system, while user stories outline specific system features. Architecture Description Language (ADL) and Unified Modeling Language (UML) are commonly used to define and visualize the system's architecture. These artifacts help identify service boundaries and are typically analyzed manually to determine the scopes and candidates for microservices.
- *Business processes*: A business process comprises activities coordinated within an organizational and technical environment to achieve a specific business goal [45]. In software systems, the dependencies between business processes—such as data, structural, semantic, and control dependencies—can be analyzed to gain insights into their interactions. These dependencies are represented as matrices, which serve as input to identify microservices.
- *Data flow diagrams*: A data flow diagram (DFD) graphically represents the flow of data within a system, detailing how business functions or operations process inputs into outputs [46]. It consists of processes (activities or functions that transform data), data stores (repositories where data is stored), data flows (paths showing how data moves between components), and external entities (sources or destinations of data outside the system). DFDs play a key role in microservice identification by mapping dependencies between processes and data stores. These dependencies are analyzed through the construction of dependency matrices, which help identify highly correlated processes and components. Alternatively, custom algorithms are used to examine the relationships between processes and data stores, aiding in the identification of microservices.
- *System features and functions*: System requirements, features, and functionalities are used to identify microservices [47]. The system functionalities are analyzed or divided into sub-tasks that cannot be divided further to identify the dependencies. Based on these dependencies, connected groups of functionalities are identified as candidate microservices.
- *Domain semantics*: Semantic analysis involves a detailed examination of various software artifacts to derive meaningful insights [48,49]. In the context of microservice identification, these techniques analyze the extracted vocabularies of system terms—such as domain-specific keywords, entity names, and operation

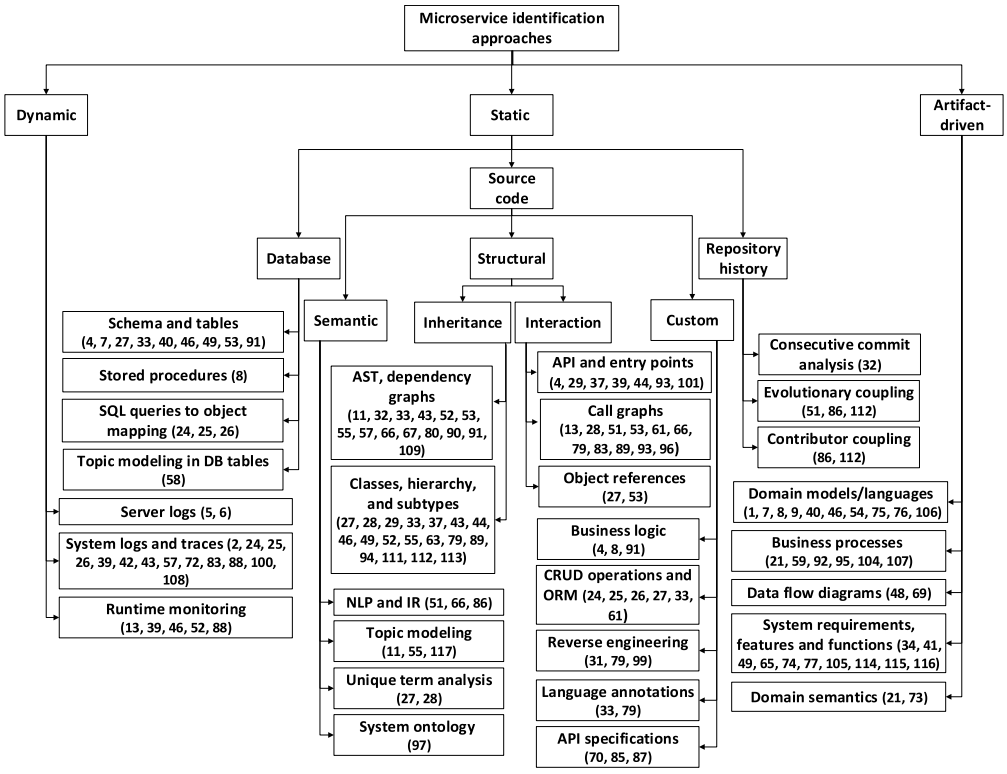


Fig. 3. Classification of approaches.

descriptions. Similarity calculations are performed on these vocabularies to identify related system entities and operations that share commonalities, enabling the grouping of these elements into potential microservices.

Static analysis

Static analysis is one of the most commonly discussed approaches for microservice identification in the literature. Static analysis techniques rely on analyzing artifacts derived from source code, databases, and version control systems. Next, we provide details on these techniques.

Source code analysis involves examining various components of a system, including classes that represent entities, core functions implementing business logic, communication APIs, and user interface (UI) components. The analysis leverages the structure and semantics of the source code, as well as custom approaches, to identify and extract potential microservices. These methods aim to group related functionalities into cohesive and independent services by studying these information elements in the source code:

- *Structural inheritance*: Structural inheritance analysis examines source code packages, classes, method-level dependencies, and class inheritance hierarchies to uncover relationships within the system. This analysis often involves constructing an abstract syntax tree (AST) of the source code, which is then used to generate system dependency graphs. In these graphs, classes and methods are represented as vertices, while their dependencies form the edges. Dependency graphs and ASTs are typically generated using static analysis tools, with further details provided in Table 5. Additionally, the class hierarchy is analyzed by examining extended (inherited) and implemented classes to identify structural relationships that may guide microservice identification.
- *Structural interaction*: Structural interaction analysis focuses on the interconnections between classes and methods in source code to identify microservice boundaries. This process begins by analyzing APIs and other entry points, such as UI calls, to determine a set of execution paths. These paths, along with their subpaths and interconnected segments, are examined to understand data usage and dependencies. Call graphs, which map method invocations within the source code, are also utilized to identify interconnected components. These graphs can be either context-sensitive, where different calls are annotated with unique identifiers to distinguish paths through the same code sections, or context-insensitive, which lack such distinctions [23]. Additionally, object reference relationships, including information flows that trigger the creation of object instances, are analyzed to uncover related classes and methods. These interconnected components form the foundation for identifying potential microservices.
- *Semantics*: These approaches examine the similarity between the words (terminology) in the source code and derive the co-related classes as possible microservice candidates. This type of approach is also known as domain-related service decomposition. The core assumption for the approaches from this category is that related features use similar terminology at the implementation level. Specifically, semantic approaches employ these techniques in their analysis:
  - *Natural language processing* (NLP) and information retrieval (IR) techniques are commonly used to extract semantic details. These techniques filter source code to exclude programming language keywords and space characters. Then, word tokenization, stop word removal, stemming, word enrichment using the synonyms from existing word dictionaries, and tf–idf calculations are performed. Brito et al. [25] use ASTs instead of source code to exclude the library dependencies to identify the terms of the system.
  - *Topic modeling* is another approach used for semantic analysis. Stop word removal and stemming are applied to remove insignificant terms and reduce multiple variations of identical terms from the source code. After identifying the unique bag of words, topic modeling classifiers like Latent Dirichlet Allocation (LDA) and Seeded Latent Dirichlet Allocation

(SLDA) are applied to group the lexical terms into clusters. These clusters are either directly identified as microservices or further processed using graph-based modeling.
  - *Unique term analysis* identifies distinct keywords in the source code and constructs a word frequency matrix for each class. This matrix is then used to calculate cosine similarity, which quantifies the semantic relatedness between two classes based on the overlap of their term distributions. By identifying classes with high relatedness, this analysis helps uncover potential groupings or dependencies that can inform microservice identification.

- *Custom analysis*: These methods leverage additional elements of the source code to identify microservices, as outlined below:
  - *Business logic* in the source code is analyzed to identify core business functions, which can then be grouped into microservices based on their roles and dependencies.
  - *Persistence layer* of the application is examined to identify entities associated with data sources, along with the Create, Read, Update, and Delete (CRUD) operations performed on them. This analysis is often conducted in conjunction with data source analysis to understand the relationships between data and services.
  - *Reverse engineering* is another custom analysis technique where reverse engineering tools are used to extract the underlying system architecture. This extracted architecture is then analyzed to apply dependency analysis, helping identify related partitions within the system.
  - *Programming language annotations* are used to identify key components in the source code. For example, Java annotations like @EJB, @Controller, and @Entity help pinpoint key classes and components, which can then be grouped into microservices based on their functionality and dependencies.
  - *API specification* techniques involve analyzing API documentation, such as those following OpenAPI standards,[6] to examine semantic similarities. This information is then used to infer potential microservices based on the relationships and dependencies between APIs.

Database analysis involves examining tables, relationships, and entity mappings used by Object-Relational Mapping (ORM) frameworks to understand how data is structured within the system. In the context of microservices, the "database per microservice" pattern is often recommended to ensure each service has its own dedicated data store, which promotes data autonomy and scalability [23,50]. When identifying microservices, it is crucial to analyze the persistent entities, such as database tables, that are associated with each service, as these entities play a key role in defining the boundaries and responsibilities of microservices. Specifically, these elements are studied:

- *Schema and tables*: The primary approach to database analysis involves examining tables, their attributes, and the relationships between them, including key constraints and triggers.
- *Stored procedures*: In legacy systems, business logic is often implemented in the database layer, typically as stored procedures, due to performance concerns and network overhead. This practice results in the mapping of stored procedures to business functions, which is another valuable technique for data source analysis.
- *Queries and business objects*: Validating SQL queries and their associated business objects is crucial for microservice identification. This involves analyzing the information derived from SQL queries, as well as the relevant entities and attributes accessed through these queries, to identify potential microservice boundaries.
- *Topics*: Topic modeling applied to database tables is another technique for data source analysis. In this method, each table is treated as a document, with its properties serving as the document's attributes. The lexical similarity between these documents is then calculated, allowing for the grouping of related tables into highly cohesive partitions, which can be identified as potential microservices.

Version control systems maintain a history of source file changes through collections of code commits, along with associated author information. Evolutionary coupling, which involves analyzing commit histories to identify correlated classes within the change logs, helps identify relationships between components based on their modification patterns. Consecutive commit analysis, a subcategory of evolutionary coupling, examines changes across multiple classes in consecutive commits to group them accordingly.

Additionally, evolutionary coupling graphs aggregate commits over different time periods. In these graphs, vertices represent classes, and edges are drawn between classes that are modified together within a single commit. This approach is known as logical coupling [21]. Lastly, the contributor coupling graph maps developers to the changes they have made in the source code. Since effective team organization is a key factor in successful microservice migration [21], this analysis helps extract system changes from the perspective of contributors.

Dynamic analysis

The final category of identified approaches is dynamic analysis. In a dynamic analysis approach, the software system is treated as a black box, where the produced outputs are analyzed based on the provided inputs to identify recurring patterns and execution traces. Three subcategories fall under dynamic analysis, as discussed below:

- *Server logs*: Server access log analysis plays a crucial role in the reengineering of web applications, where web server access log files are examined to identify frequently invoked URIs. Server logs, such as those from Apache Tomcat,[7] and WildFly[8] provide detailed information on access URIs, request and response times, and response sizes. These logs are analyzed by examining the frequency of URIs and response sizes and times, which helps group requests into potential candidate microservices.
- *System logs*: Most existing studies that conduct dynamic analysis rely on system log analysis. Instrumenting the source code using aspect-oriented programming (AOP) is a log collection technique in which an agent is integrated into the source code to capture logs based on the operations performed by the system. These logs are subsequently provided as inputs to a process mining tool, like Disco,[9] or analyzed further to identify frequent execution traces, processes, and dependencies. The validity of this approach depends on the extent of coverage of actions performed on the instrumented system. To enhance the coverage of operations, use cases, functional tests, unit tests, and user simulations have been employed.
- *Runtime monitoring*: Runtime monitoring has been defined as another class of dynamic analysis approaches. In such an approach, the system is observed during execution time, and collected information is used for system reengineering. Kieker,[10] Elastic APM,[11] and dynatrace[12] are the tools used for this purpose.

[6] https://spec.openapis.org/oas/v3.1.0.

[7] https://tomcat.apache.org/.
[8] https://www.wildfly.org/.
[9] https://fluxicon.com/disco/.
[10] https://kieker-monitoring.net/.
[11] https://www.elastic.co/.
[12] https://www.dynatrace.com/.

Table 4
Tools and levels of automation; automated (A) and partially automated (PA).

Study ID	Level of automation	Available artifacts
1	PA	https://github.com/ServiceCutter/ServiceCutter
11	A	https://github.com/miguelfbrito/microservice-identification
24	PA	https://github.com/AnuruddhaDeAlwis/NSGAII
25	PA	https://github.com/AnuruddhaDeAlwis/Subtype
26	PA	https://github.com/AnuruddhaDeAlwis/NSGAIIFOROptimization
29	PA	https://github.com/utkd/cogcn
37	PA	https://github.com/Rofiqul-Islam/logparser
39	PA	https://github.com/wj86/FoSCI
42	A	https://www.ibm.com/cloud/mono2micro
51	PA	https://github.com/loehnertz/Steinmetz https://github.com/loehnertz/semantic-coupling
52	PA	https://github.com/tiagoCMatias/monoBreaker
55	A	https://essere.disco.unimib.it/wiki/arcan
61	A	https://github.com/socialsoftware/mono2micro
70	A	https://github.com/HduDBSI/MsDecomposer
77	PA	https://github.com/RLLDLBF/FeatureTable
79	PA	https://gitlab.com/LeveragingInternalArchitecture/IdentificationApproach
86	PA	https://github.com/gmazlami/microserviceExtraction-backend https://github.com/gmazlami/microserviceExtraction-frontend
89	A	https://drive.google.com/drive/folders/1TQaS8etLr-32d0RXwC1Le-IOMVaDBcSS

Furthermore, a hybrid approach can integrate several artifact-driven, static, and dynamic analysis techniques. However, in such an approach, one technique is often dominant. For instance, static analysis may be performed first, and the extracted data can then be enhanced with dynamic analysis details for further investigation [43,51,52]. Alternatively, artifact-driven analysis may serve as the dominant technique, with static analysis providing additional insights [53].

3.2.2. (RQ2.2) what tools exist, and which level of automation do they support?

In the existing studies, two types of tools have been identified: tools developed during the studies of microservice reengineering (in line with the concept in the study) and existing tools to support different stages of the reengineering process, e.g., call graph generation and log analysis. The existing migration frameworks, their levels of automation, and freely available source code/tools are listed in Table 4. Frameworks that provide microservice recommendations based on primary inputs, like source code, log files, and system artifacts, are considered automated. The studies with tools involved in different stages of the migration process, like data extraction and system modeling, are categorized as partially automated.

Multiple categories of tools are available based on the approaches used to examine the monolithic system. There are tools for the static analysis of software systems, database administration, runtime monitoring, visualization, architectural validation, and load simulations. These tools, technologies used, and respective study IDs are listed in Table 5. Moreover, a comparison between existing tools utilized to extract microservices has been made in a separate study by Lapuz et al. [54]. Ren et al. [55] used their tool EasyAPM to record the operation data and parameter information through the instrument on the JDBC and data access class libraries. Other supportive tools used for testing, clustering, and other specific purposes are listed in Table 6. The purpose column indicates the use of these tools in different steps in the system modeling and microservices extraction process.

3.2.3. (RQ2.3) which techniques/algorithms are used?

The identified techniques can be broadly categorized into two types: system modeling techniques and microservice extraction techniques. The system modeling techniques are used to interpret or model software systems, creating their abstract representations, while microservice extraction techniques are applied to identify the microservices within the interpreted systems, thereby defining boundaries of potential microservices. Identified system modeling and extraction techniques are summarized in Figs. 4 and 5, respectively. Leaf nodes in the figures refer to the relevant study IDs.

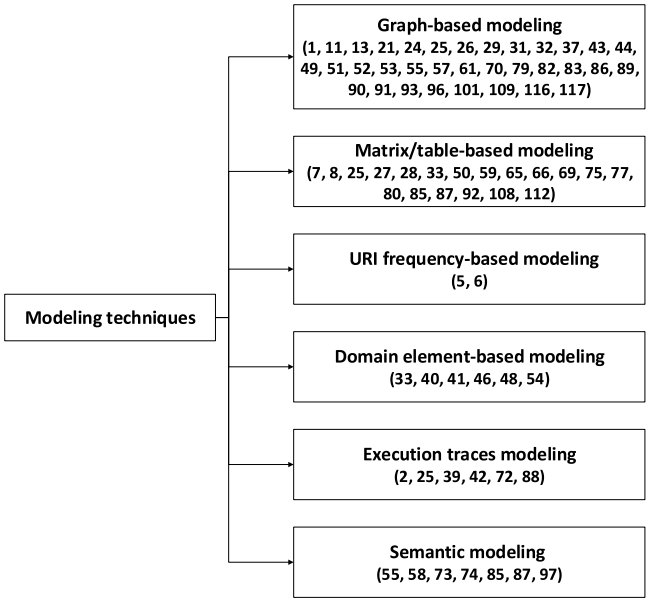


Fig. 4. Classification of legacy system modeling techniques.

System modeling techniques

Graph-based modeling is the prominent technique for modeling legacy systems, refer to Fig. 4. The vertices in such graphs can be components, system entities, classes, methods, business processes, entry points, execution traces, database tables, and system functionalities. Edges can be either weighted or non-weighted. Undirected weighted edges are frequently used in the system graphs. The existence of an edge and its weight are based on the strength of the relationship between two vertices. Structural relationship graphs are constructed based on the number of dependencies, method calls, and coupling scores. Dependencies and method calls are directly derived from ASTs, call graphs, and dependency graphs. Moreover, structural relationships can be prioritized by assigning weights based on their types, e.g., generalization, aggregation, implementation, association, instantiation, and method invocation [56].

Table 5
Tools for monolithic system analysis.

Purpose	Tool/Library	Details	Technology	Study IDs
Static analysis (source code)	Java call graph (open source)	Reads jar files to collect the method calling sequences. Dynamic Analysis is possible but is used only in static context. (https://github.com/gousiosg/java-callgraph)	Java	13, 111, 112
	Java parser/Symbol Resolver (open source)	Constructs abstract syntax tree for structural dependency extraction. (https://javaparser.org/)	Java	11, 33
	Mondrian (open source)	Performs static source code analysis (https://github.com/Trismegiste/Mondrian)	PHP	27, 28
	WALA (open source)	Analyzes project class hierarchies and generates call graphs. (https://github.com/wala/WALA)	Java, JavaScript	53, 57
	Soot (open source)	Models source code to analyze, instrument, optimize, and visualize applications. (https://soot-oss.github.io/soot/)	Java, Android	29, 53
	Doop & Datalog (open source)	Conducts static analysis of source code using Datalog engine. (https://plast-lab.github.io/doop-pldi15-tutorial/)	Java, Android	53
	JackEE (open source)	Provides static analysis of Java Web applications with enterprise framework support. Additional parameter is used for Doop framework to run JackEE. (https://github.com/plast-lab/doop)	JEE applications	53
	Spoon (open source)	Parses source code into abstract syntax tree for further analysis. (https://spoon.gforge.inria.fr/)	Java	61
	Structure 101 (commercial)	Validates software architectures by visualizing their structures from source code. (https://www.sonarsource.com/structure101/)	Java, .Net, C/C++	2, 46, 71, 111
	Sonargraph Architect (commercial)	Offers architecture checks, duplicate code detection, virtual refactorings, cyclic dependency resolution, and comparison with previous versions. Suppors Git repository mining. (https://www.hello2morrow.com/products/)	C#, C/C++, Java, Python 3	77
Semantic analysis (source code)	ANTLR (open source)	Parses the source code to generate grammar for language recognition. (https://www.antlr.org/)	Java, C#, Python, Go, C++, Swift, JavaScript, TypeScript	97
Static analysis (database)	SchemaSpy (free software)	Generates Web-based visual representations by analyzing database metadata. (https://schemaspy.sourceforge.net/)	Java-based tool	2, 71, 72
	DBeaver (free and commercial versions)	Provides tools for database administration and schema analysis. (https://dbeaver.io/)	MySQL, Maria DB, PostgreSQL, SQLite	46
	JSqlParser (open source)	Parses SQL statements and translates them into hierarchies of Java classes. (https://github.com/JSQLParser/JSqlParser)	Java, SQL	61
Dynamic analysis	Kieker (open source)	Monitors and analyzes runtime behavior of software systems. (https://kieker-monitoring.net/)	Java, .Net, C, VB	13, 39, 88, 108
	Elastic APM (commercial)	Supports real-time monitoring, performance analysis of incoming requests/responses, database queries, cache invocations, and external calls. (https://www.elastic.co/solutions/apm)	Java-based Web, Data access frameworks, application servers, messaging frameworks, AWS	2, 72
	Disco (free and commercial versions)	Analyzes event logs to identify call graphs and enables automated process discovery. (https://fluxicon.com/disco/)	Log files of software systems	2, 24, 25, 26, 72
	ExplorViz (open source)	Provides runtime monitoring and visualization of software systems (https://explorviz.dev/)	Applied to Java-based systems	46
	django-silk (open source)	Profiles and inspects the django framework, analyzing HTTP requests and database queries. (https://github.com/jazzband/django-silk)	Python django framework-based tools	52

Static dependency graphs are generated by static analysis tools. Subsequently, the coupling scores are often calculated manually based on the pre-defined parameters. Depending upon the four categories of cohesiveness, compatibility, constraints, and communication, 16 coupling criteria have been defined by Gysel et al. [31]. A priority and score can be defined for each criterion that contributes to the final edge weight of the graph. Semantic similarity-based graphs are based on tf–idf (term frequency–inverse document frequency) or topic modeling. Once the tf–idf is calculated, a vector with the frequency of each word distribution in the class is obtained. The cosine similarity

Table 6
Additional tools used for system analysis.

Purpose	Tool	Details	Technology	Study IDs
Testing	Jmeter (open source)	Provides load simulation (https://jmeter.apache.org/)	Java	6, 7, 108
	Gatling (commercial)	Provides stress testing (https://gatling.io/)	Java, Kotlin, Scala	16
Reverse engineering	MoDisco (open source)	Provides model-driven reverse engineering of the source code (https://wiki.eclipse.org/MoDisco/)	Java, JEE, XML	31, 99
Topic modeling	GuidedLDA (open source)	Provides topic modeling using latent Dirichlet allocation (https://guidedlda.readthedocs.io/en/latest/)	Python	55
Clustering	SciPy (open source)	Provides hierarchical clustering and generates dendograms (https://www.scipy.org/)	Python	61, 111
Optimization algorithm	Jmetal (open source)	Supports multi-objective optimization algorithms NSGA II and NSGA III (https://jmetal.sourceforge.net/)	Java	82, 96
Document enrichment	WordWeb, WordNet (public dictionary)	Lexical databases to identify synonyms for topic modeling (https://wordnet.princeton.edu/)	Word dictionary	58
Lines of code count	CLOC (open source)	Blank, comment, and physical lines counting (https://github.com/AlDanial/cloc)	Java, C, Python	12

between two vectors is calculated, capturing the degree of similarity between two data points. A high degree of similarity defines the closely related classes.

Probabilistic Latent Semantic Analysis (PLSA), Latent Dirichlet Allocation (LDA), Latent Semantic Analysis (LSA), and Non-negative Matrix Factorization (NMF) are four classes of algorithms used for topic modeling. Latent Dirichlet Allocation (LDA) and Seeded Latent Dirichlet Allocation (SLDA) are commonly used to identify the topic distribution within the source code. LDA is a probabilistic topic model. It is an unsupervised model, whereas SLDA is a semi-supervised variant of LDA. SLDA accepts the list of keywords as input that stimulates the expected topics. LDA uses high coherence and fewer overlaps between the concepts to derive clusters of concepts [57]. Once the clusters are identified, cosine similarities between the clusters are calculated to define the edge weights in the graph representation.

In dynamic analysis-based graphs, edges represent runtime frequencies of method invocations and execution traces, while evolutionary coupling graphs define edges based on correlations between classes in commits and contributors involved in their development. Examples of graph models include classes or components as vertices with topic modeling strengths as edge weights, domain entities as vertices connected by coupling scores as edge weights, call graphs where vertices represent classes or methods and edges represent execution calls and their frequencies, system entities and entry points as vertices with method calls as edges, system classes or entities as vertices connected by evolutionary coupling edges based on revision history, classes as vertices with edges denoting contributors involved in their development, runtime graphs with classes or methods as vertices and edges representing invocation relationships and frequencies, and architectural graphs generated by reverse engineering tools.

Matrix/table-based modeling represents a software system as a mapping of its attributes and components captured in a matrix with the number of occurrences as entries to classify the co-related attributes further. Once matrices are constructed, similarity measures are used to identify related components that can define microservices. Either classes, methods, database tables/entities, use cases, micro-tasks (tasks that cannot be decomposed further), or business processes are used in the computations of the frequencies of executions, sub-type/reference relationships, coupling, and cohesion values to determine the relationship between elements. Semantic similarity analysis uses classes against unique word matrices. Then, cosine similarity determines the semantic similarity between the classes. Example matrix/table-based modeling techniques include use-case-to-use-case similarity and use-case-to-database-entity similarity matrices, subgraph similarity matrices, class-to-database-object matrices, class-subtype (subtype relationships between classes) matrices, class-reference-type matrices, micro-tasks-to-data-object matrices, business process dependency matrices, structural similarity matrices (structural relationships between classes in a matrix format), conceptual similarity matrices (semantic similarity between classes in a matrix format), read/write operations between primitive types (further non-decomposable functions) and data storage, user story coupling and cohesion matrices, BPMN structural and data dependency matrices, feature tables, and use case to business process mapping tables.

URI-based modeling is an approach to modeling web applications. Web applications operate on a request/response base, where features are requested via URI calls, and responses are redirected to the relevant clients. Application servers like Tomcat and WildFly record logs with the request/response details. These details are used to infer models of the applications and identify the frequent URI calls that can be isolated as separate services for better performance. Mean request/response time (MRRT) and response size are used as indicators of network overhead and resource utilization.

Domain element-based modeling is another approach used to represent software systems. This approach uses data flow diagrams, UML diagrams, system capability models, and context maps to represent the software systems. This is a manual approach with detailed system diagrams with fine-grain information and capabilities that are analyzed to identify bounded contexts.

Execution trace modeling uses software logs to identify the actual methods/classes invoked during the runtime of the software systems. The collection of active execution traces defines the overall behavior of the system. In addition, inactive paths can be identified in the runtime traces analysis [35]. Multiple techniques have been used in the literature to investigate these execution traces. One approach is providing the software logs into the runtime trace analysis tool, e.g., Disco process mining tool [19]. Tool-generated execution call graphs can be used to analyze and extract the co-related classes/methods manually [19] or programmatically identify the subtypes and common subgraphs [52]. Execution traces can be further modeled and reduced to identify functional atoms, which are coherent and minimal functional units [29], identify direct/indirect call patterns in execution

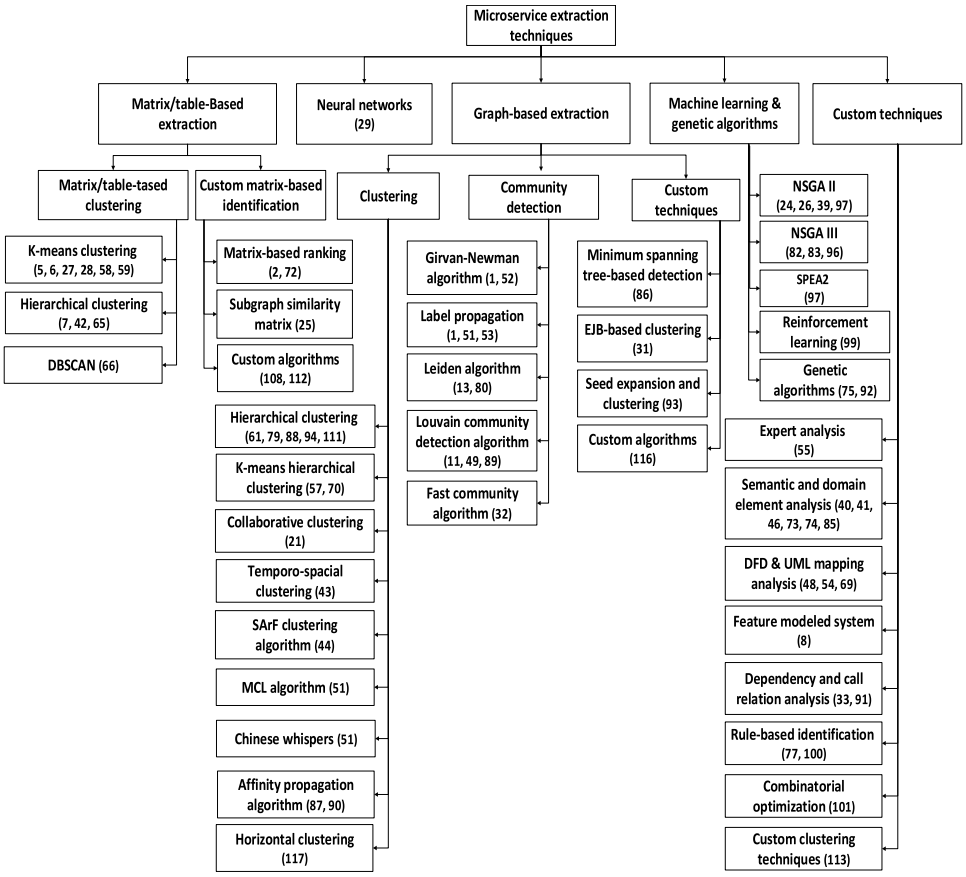


Fig. 5. Classification of microservice extraction techniques.

traces [35], and analyze class and method level execution traces based on system functionalities [30].

Semantic-based modeling is used to model the system based on linguistic information. Identifying system topics based on the application domain [58], generating a vocabulary tree to illustrate the system terminology [49], and examining the system subject and operations to group the terms used in the API specifications [28,59] have been done in semantic-based modeling.

Microservices extraction techniques

Fig. 5 illustrates the microservice extraction techniques identified in the study. Following the modeling of the system using the aforementioned techniques, the extraction process is conducted to identify potential microservice candidates. Clustering is used as the predominant extraction technique.

Graph-based extraction is the leading technique due to the widespread use of graph-based modeling. Hierarchical clustering is used when the number of clusters is not given as an input. In contrast, K-means clustering is used when prior knowledge of the number of desired clusters (microservices) is available. The advantage of parameterizing the number of clusters is the ability to analyze the service decomposition with any possible number of services. It can be used for a better understanding of the system and coupling between the parts of the system [31]. Two variations of hierarchical clustering that are used are agglomerative clustering and divisive clustering. Agglomerative clustering starts with data points and iteratively generates the clusters, whereas divisive clustering starts with the complete dataset and splits it into clusters. Furthermore, temporo-spatial clustering and collaborative clustering are also used, implemented as adaptations of hierarchical clustering and specifically hierarchical agglomerative clustering,

Community detection studied in large-scale networks has been applied for microservice extraction from graph models. For instance, Girvan–Newman deterministic [60] and Epidemic Label Propagation (ELP) non-deterministic [61] algorithms were applied to discover microservices. ELP algorithm takes in the number of clusters as an input parameter. Louvain and fast community detection algorithms are based on maximizing modularity within a given network. Louvain algorithm is an unsupervised algorithm. It is based on modularity maximization and does not require the number of communities or the size of the communities as input [25]. Among the algorithms evaluated for microservice detection, including MCL, Walktrap, Louvain, label propagation, Infomap, and Chinese Whispers, it was observed that the Louvain algorithm demonstrated the highest performance in supporting the identification of microservices [33].

Hierarchical agglomerative clustering is used to analyze matrix/table-based models of systems. DBSCAN is a density-based clustering algorithm that aims to group elements that are densely packed in the search space and identify noisy elements that do not fit into any clusters using two concepts, which are neighborhood distance and the minimum number of elements in a neighborhood [26].

The Non-dominated Sorting Genetic Algorithm II (NSGA II) and Non-dominated Sorting Genetic Algorithm III (NSGA III) are multi-objective optimization algorithms. A multi-objective optimization algorithm aims to provide optimal solutions while achieving global optima when multiple conflicting objectives, e.g., coupling, cohesion, and modularity, are to be considered [51]. Two studies have compared the performance of NSGA II and NSGA III and identified that NSGA III does not consistently outperform NSGA II in microservice discovery [62,63].

Several custom extraction techniques have been identified in the literature. Manual and expert analysis are basic extraction techniques, with artifact-driven approaches being the most widely used manual microservice identification approaches.

Table 7
Inputs and outputs of artifact-driven approaches.

Study IDs	Input and intermediate representation	Output
1	JSON-based representation of SSA to identify nano entities to generate a graph with coupling as the edge weight.	Entities grouped into clusters to represent microservices (service cuts).
7	Component-attribute dataset. Use case to use case (U to U) and use case to database entity (U to DB) relationship matrix to generate the similarity matrix between use cases.	Candidate microservices with use case grouping.
8	Microservice Discovery Table (MDT) with requirement, features, and stored procedure mapping.	MDT augmented with information on microservices, entities, and rules.
21	Business processes and dependencies (control, semantic, data, and organizational) and dependency score matrix.	Groups of cohesive activities.
40	Ubiquitous language, business operations, data operations, domain models, database schema, and design documents.	Bounded contexts obtained after DDD pattern analysis, business operation, and data dependency analysis.
41	System responsibilities obtained and ubiquitous language.	Identified candidate microservice boundaries.
48	Data flow diagram (DFD) of the system.	Set of decomposable DFDs and grouping of DFDs as microservices.
50	System functionalities—mapping between business requirements and system services. Task dependency matrix for clustering.	Task decomposition as clusters to represent microservices.
54	Class model derived from UML diagrams—boundary (interface), control (business logic), and entity (mapped to database table).	Entities separated as microservices.
59	Set of business processes (BPs) to generate the dependency matrix.	Set of clusters derived from dependency matrix.
69	Data flow diagrams (DFD) as the input. Relationship matrix between primitive functions and data storage for extraction.	Primitive functions grouped into microservices.
73	BPEL of the system converted to Subject–Verb–Object table to obtain system vocabulary trees.	System operations grouped as microservices derived from vocabulary trees.
74	Use case, requirements, and functionalities. From use cases, generate operation/relation table.	Manually identified microservices from the visualization of the operation/relation table.
75, 115	Product backlog's user stories.	Decomposed microservices, backlog diagram, and quality matrices.
76	Architecture Domain Language (ADL) to identify bounded context from ADL.	Converted and deployable system with database and repository per microservice.
77	Feature cards and feature table.	Feature partitions identified as microservices based on mapping rules.
92	Business processes converted to structural and data dependencies relationship matrix.	Clustered processes as microservices.
116	System requirements to derive graph-based representation of problem domain and correlation as vertices and edges.	Clustered problem domains as microservices.

3.2.4. (RQ2.4) how is data used?

Next, we discuss the inputs and outputs of existing approaches for reengineering software systems into microservices.

The artifact-driven approaches use artifacts like UML diagrams, Data Flow Diagrams (DFD), use cases, user stories, and architectural documents as inputs. Most existing artifact-driven reengineering approaches are manual. However, several existing studies convert system artifacts to computer-readable formats or intermediate representations and use them to identify candidate microservices. After such (semi-)automatic identification, recommended microservice candidates are delivered as output. Such outputs can have visual representation or be given as clusters of elements. Table 7 summarizes formats of inputs and outputs used by the artifact-driven approaches. This table only covers studies with clearly identified input and output details.

Study 1 is a semi-automated approach that takes System Specification Artifacts (SSA), such as UML and ER diagrams, use cases, security zones, and entities, in JSON-based machine-readable format as input. Study 7 uses use-case-to-use-case and use-case-to-database-entity relationship matrices to generate a similarity matrix, which serves as a basis for microservice identification. As input, ⟨component — attribute⟩ data matrix is used, where components can be the use cases of the system, and attributes are its properties. Study 8 uses business requirements, features, and stored procedure/business logic mapping for features as input. Then, a microservice discovery table (MDT) is created with system requirements, corresponding features of interest in the source code, and the stored procedures that implement the business logic. This table is then used as the ground for microservice discovery. The control, semantic, data, and organizational dependencies between business processes represented in a matrix format with a dependency score matrix are used in Study 21 as input to the microservice extraction. Studies 40, 41, 48, 54, 75, and 76 use domain artifacts, such as UML, DFD, ADL, BPEL, and use case diagrams as input. Studies 40 and 41 produce bounded contexts identified as microservice candidates as outputs. Studies 58 and 68 follow the same pattern and produce DFD and entity groping, respectively, as output. In contrast, the results of Study 75 and Study 115 are a set of matrices and microservices. The matrices indicate the quality measures of extracted microservices in terms of complexity, coupling, cohesion, interface count, and estimated development time. Study 76, as output, provides a converted and deployable system with a repository and database per microservice. In Study 50, business requirements and functionalities are divided into task levels, and a dependency matrix is created for microservice identification. Studies 59, 69, and 92 use matrix-based representations derived from business processes, while DFDs are used to extract the microservices. A table-based representation of domain artifacts is input to Studies 73, 74, and 77. Business Process Execution Language (BPEL) models are used in Study 73 to derive the subject–verb–object relationship table. This table is used as a vocabulary to identify system operations. These system operations are used as output microservices. In Study 74, use cases are used to construct operation/relations tables

for requirements and functionalities. This table visually represents the identified microservices and is the output of the approach. Study 77 is grounded in system features. It has additional input of feature cards, assigning weight to features. Microservice candidates are produced, resulting from feature table analysis using predefined rules.

The inputs of static analysis approaches are source code, database artifacts, and code repository histories. Most studies represent the source code as graph- or matrix-based abstractions, which are then used to discover microservices. Specifically, graph or matrix-based clustering, genetic, and community detection algorithms are used. As output, these approaches often provide clusters that define candidate microservices. Table 8 summarizes the details of inputs and outputs used by static analysis approaches. Again, only the studies with detailed descriptions of the inputs and outputs are included in the table.

Inputs and outputs of the dynamic analysis approaches are detailed in Table 9. These approaches often perform statistical studies over the system's performance data before identifying its constituent parts or microservices. System logs are usually collected using instrumented source code. The latter is also used to conduct use cases and functional testing of the original and reengineered systems.

3.2.5. (RQ2.5) how are the reengineered systems evaluated?

Once the microservice extraction process is completed, the migrated system can be evaluated from various perspectives. From a functional viewpoint, the migrated system must retain all the essential features and functions provided by the legacy system. Additionally, the performance of the system should meet acceptable standards post-migration. The system should also maintain key quality attributes, such as modularity, loose coupling, high cohesion, and appropriate granularity. The literature highlights several techniques for evaluating reengineered systems, including manual expert evaluations, prototype implementations, industrial case studies, cross-system comparisons, and property assessments.

The basic approach for validating the refactored system is via expert opinions, which can be carried out directly by experts evaluating the refactored system or indirectly by comparing the resulting system with expert-extracted solutions. Prototyping is another approach in which the proposed reengineering technique is applied over one or multiple open-source systems. In contrast, in industrial case studies, a migration approach is evaluated based on industry applications. Cross-system evaluation is a highly used technique in which the proposed solution gets cross-compared with the available state-of-the-art techniques to check if the new solution is superior. Property measuring is another widely used technique. Properties like modularity, quality of decomposition, and runtime performance of the original and reengineered systems are calculated and compared. Moreover, a few studies have considered hyperparameter optimization [26,59], where reengineering technique configurations are evaluated for performance tuning.

The properties used to measure the quality of the reengineered systems can broadly be categorized into six categories: runtime performance, modularity, coupling, cohesion, independence of functionality and evolvability, and quality of decomposition. These categories and the studies in each category are summarized in Fig. 6.

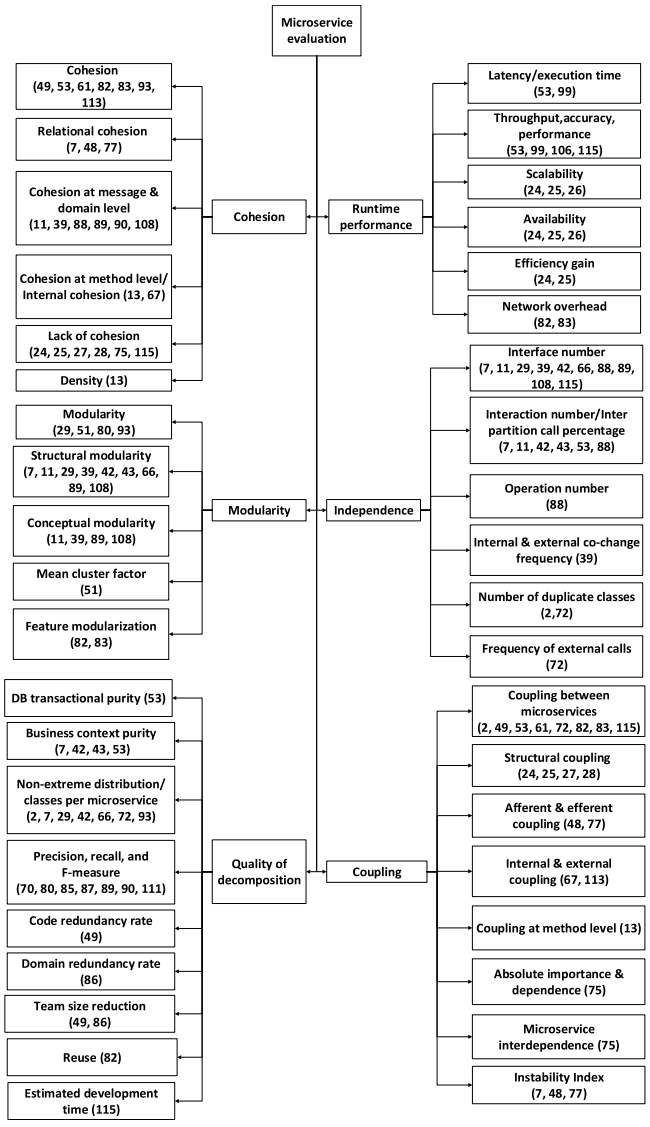


Fig. 6. Classification of evaluation techniques.

Runtime performance

Runtime performance analyzes the properties of the reengineered system during the execution phase and compares them with the corresponding properties of the monolithic application. In this context, the efficiency gain is the proportion of the total time taken by the legacy system to process all the requests compared to the total time taken by the corresponding microservices system to process the same requests.

Modularity

Modularity measures the quality of the clusters and how well components of a system can be distinguished, decomposed, and recombined. Structural modularity measures the soundness of the clusters from the structural viewpoint, while conceptual modularity measures the conceptual soundness of the clusters. The mean cluster factor analyzes the interconnectivity and intraconnectivity of the clusters or microservices. Feature modularity is a measure of feature distribution across the system derived from the notion of the single responsibility per microservice. The predominant feature number is the number of occurrences of the most common feature divided by the sum of all feature occurrences. Feature modularization is the sum of the predominant feature number in every microservice divided by the number of distinct features. In most of the studies, modularity calculation has been conducted based on the method by Newman and Girvan [60].

Coupling

Coupling measures the level of interaction between services. Structural coupling refers to the structural relationships between services. Afferent coupling quantifies the responsibility of a service by measuring the number of classes in other services that depend on the classes within the service. Efferent coupling indicates the extent to which the classes in a service depend on the classes in other services. The instability index is calculated as the ratio of efferent coupling to the sum of afferent and efferent coupling, reflecting a service's resilience to changes in other services. Internal coupling measures the degree of direct or indirect dependencies between classes within a microservice, while external

Table 8
Inputs and outputs of static approaches.

Study IDs	Input and intermediate representation	Output
11	Source code, AST, and topic-based strength between components as a graph.	Components partitions as microservice candidates.
27	Source code and SQL queries are represented as classes to business object relationship matrix, cosine similarity matrix with semantic similarity between classes and subtypes, and reference relationship matrices generated by analyzing the class relationship graph generated by the Mondrian tool.	Classes partitioned into clusters as microservices.
28	Classes to business object relationship matrix, cosine similarity matrix with the semantic similarity between methods, method call relationship matrix.	Methods partitioned into clusters as microservices.
29	Source code represented as a graph with classes as nodes and edges as calls between the classes. Classes and entry point matrix, classes vs. number of common entry points matrix, and class inheritance matrix.	Cluster assignment matrix.
31	Source code to MoDisco tool to get the system model as AST.	Visual representation of EJB clusters and microservices.
32	Source code and repository represented as a graph. Classes/interfaces as vertices, static and evolutionary coupling as edges.	Set of clusters as microservice candidates.
33	Source code, database, set of proposed microservices in JSON format. Source code as abstract syntax tree for structural data extraction.	Database and source code refactored as microservices.
44	Source code as a set of programs and data (data access write, read operations) represented as a graph.	Visualization as the list of programs and data using city metaphor.
51	Source code and repository history represented as static, semantic, and evolutionary coupling graphs.	Classes of clusters as microservices.
53	System dependency graph of source code and database.	Graph communities as recommended microservices.
55	Source code is the input to the Arcan tool that creates a system dependency graph.	Semantics of the migrating project with Java classes as microservices.
58	Database tables and table attributes for topic detection.	Clusters of tables as microservices.
61	System functionalities and persistent domain entities.	Clusters of domain entities as microservices.
66	Structural similarity and semantic similarity matrix.	Classes grouped into microservices and outlier classes.
67	Classes in the source code.	Classes grouped into microservices based on dependencies.
70	Open API specification based API details to generate API similarity graph.	API clusters as microservices.
79	Source code for reverse engineering to obtain layered architecture metamodel for class clustering based on structural and data similarity.	Clusters of classes as microservices.
80	Source code represented as class dependency graph.	Visualization of graph clusters as microservices.
85	Open API specification based API details and reference vocabulary details to calculate semantic similarity.	API mappings as microservices.
86	Source code to derive logical, semantic, and contributor coupling graphs.	Clusters of classes as microservices.
87	Open API specification based API details to extract operation names for semantic similarity.	Clustered operation names as recommended microservices.
89	Source code to analyze static and semantic relationships using machine learning techniques to generate graph-based representation of the system.	Clusters of classes as microservices.
90	Source code to extract the methods and code embedding model using neural network model (code2vec) and cluster based on semantic similarity.	Clusters of classes as microservices.
91	Source code and database to generate dependency graph of classes, facades, and database tables as vertices and call relationships as edges.	Identified microservice candidates from dependency graph.
93	Call graph of the source code with entry points and database access points.	Clusters created around the detected seed classes.
96	Source code with indicators that should not be parsed, a list of features and related execution of the legacy system and the number of microservices to be identified.	The candidates as individual graphs and the associated legacy system code.
97	Source code semantic descriptors in Extended Backus–Naur Form (EBNF).	Identified microservices in EBNF format.
99	Source code model after extracting by MoDisco tool with service cuts (from Study 1) to train the AI-based application.	Mapping between microservices and methods in the source code.
101	Source code as a graph with methods/entities as vertices and the number of invocations of methods/entities as edge weight.	Clustered methods/entities as microservices.
109	Graph-based representation of the source code constructed by using the AST and call graph of the source code.	Candidate microservices identified by combining highly coupled classes in the graph.
111	Call graph generated from the source code.	Set of clusters as candidate microservices.
112	Source code classes/methods identified by Java call graph and repository history to generate similarity matrix of related classes/methods.	Set of clusters as candidate microservices.
117	Database and source code classes mapping to calculate semantic similarity.	Set of graph-based clusters as microservices.

Table 9
Inputs and outputs of dynamic approaches.

Study IDs	Input and intermediate representation	Output
2	Log files collected after AOP-based instrumentation feed into the Disco tool to obtain graphical representation of processes.	Multiple decomposition options with matrix-based ranking for solution selection.
5	Web server access logs to analyze URI invokes.	URI frequency and mean request response time (MRRT)-based clusters.
6	Web server access logs to analyze URI invokes.	Response size- and time-based clusters.
13	Monitoring logs generated using Kieker with full business operations coverage.	Method invocation logs with time and frequency as inputs for a node attribute network.
24	System logs with major functionality and use case coverage.	Call graph generated using the Disco tool, combined with static analysis results of business objects and operations for clustering.
25	System logs with major functionality and use case coverage.	Call graph generated using the Disco tool, combined with business objects to identify single-entry-single-exit (SESE) regions to derive frequently executed patterns (FEPs).
26	Execution logs collected by simulating user behavior using Selenium scripts.	Call graphs related to executions.
39	Collected execution traces using the Kieker tool with a predefined functional test suite.	Grouped functional atoms after applying NSGA-II on identified functional atoms from execution traces.
42	Use case-based runtime logs to identify direct and indirect call relationships to generate a similarity matrix between classes.	Clustered set of class partitions based on similarity.
43	Use case-based logs collected from instrumented source code to generate a calling context tree.	Classes as clusters derived after combining dynamic data with static information.
46	Live monitoring and visualization using the ExploreViz application.	Identified bounded contexts by static analysis and ExploreViz visual results.
52	Operational data (entry points, classes, queries) collected using the Silk tool.	Classes as clusters after combining results with system static data.
57	System logs to analyze statistics and invoke relationships to generate the call graph with dynamic tracing frequencies.	Clustered microservice partitions.
72	Collected traces after AOP-based instrumentation to feed into the Disco tool.	Microservices identified after visually inspecting the tool-generated call graphs.
83	Data on the frequency of method invokes collected by instrumenting the source code.	Identified microservice boundaries after combining with static details of the source code.
88	Log files generated after instrumenting the source code and executing test cases.	Identified microservices after execution traces analysis.
100	Traces collected from the software system.	Set of class/package interactions as microservices.
108	Execution traces to derive an object call relationship matrix.	Clusters of classes as microservices.

coupling assesses the dependencies between a class in a candidate microservice and external classes. The absolute importance of a service (AIS) is defined as the number of clients that invoke at least one operation of the microservice interface. Similarly, the absolute dependence of a service (ADS) refers to the number of other microservices that invoke at least one operation of the service. Finally, interdependence represents the total number of dependent service pairs.

Cohesion

Cohesion measures the degree of interconnectedness of a service. It represents the number of static calls within a server over all the static calls. Relation cohesion is the number of internal relationships, including inheritance, method invocations, access to class attributes, and access via references. Cohesion at the message level (CHM) defines the cohesiveness of interface messages, while cohesion at the domain level (CHD) is the cohesiveness of services measured using the similarity of functions. Lack of cohesion is the number of pairs of services that do not have interdependence. Density is the degree of internal co-relation of each microservice.

Independence of functionality and evolvability

A microservice should be independent and support flexible changes in the system that do not affect other services. Therefore, functional independence is an essential characteristic of microservices. Interface number is the average number of interfaces published by a microservice. The percentage or number of calls between two microservices is measured as the interaction number or interpartition call percentage. Operation number (OPN) is the number of operations provided by the microservice. Internal and external co-change frequency is the frequency of entity changes inside and outside the microservices calculated based on the revision history. The frequency of external calls is measured as the fraction of the number of calls over the number of classes in a microservice. In addition, the fraction between external change frequency (across services) and internal change frequency (within services) is known as the REI ratio. Ideally, changes inside a service should be higher than those across the services. Therefore, the value is expected to be less than one. Smaller values indicate the services tend to evolve independently [29].

Quality of decomposition

The measures from this category assess the quality of the functional distribution across the microservices. This distribution can be, for instance, in terms of use cases, operations, or classes. Business context purity indicates business use case distribution across the services. It is defined as the mean entropy of business use cases per partition. DB Transaction purity measures the distributed transactions. This measure

Table 10
Evaluated applications.

Application	Study IDs	Technology
JPetStore	7, 11, 13, 39, 42, 53, 57, 66, 80, 88, 90, 108, 115, 116	Java
Acme Air	6, 7, 29, 53, 66, 80, 93	Java
Cargo Tracking System	1, 7, 48, 49, 77, 80, 115	Java
Daytrader	29, 43, 53, 55, 57, 66, 93	Java
Springblog	39, 80, 88, 90, 113	Java
Jforum	39, 88, 89, 90	Java
Apache Roller	39, 88, 90	Java
Spring boot pet clinic	44, 66, 89, 93	Java
E-commerce system	49, 58	Java
Microservices event sourcing	66, 70	Java
Kanban board	66, 70	Java
TFWA (Teachers Feedback Web Application)	5, 7	Java
Train Ticket Microservice Benchmark	12, 88	Java
Plants by WebSphere	29, 53	Java
SugarCRM	24, 25, 26, 27	PHP
ChurchCRM	24, 25, 26, 27	PHP

prioritizes decomposition with dedicated databases per microservice. Per each DB table, calculate the partitions that access the table to get the entropy. Smaller entropy values indicate high transactional purity [23]. The degree of even distribution of the classes among the microservices has been measured in non-extreme distribution. Code redundancy rate is the code volume difference between the original and migrated systems over the original code. Domain redundancy rate measures the duplication of responsibilities. The team size of each service is defined as the number of functions provided by the partition. Reuse is measured by the relationship between identified services and the legacy system users, e.g., API calls and UI interactions. Analysis needs to be conducted in the migrated system to calculate this property.

Other measures

MoJoSim and MoJoFM are used to evaluate a microservice-based architecture against a reference architecture, e.g., against an expert-identified architecture. It is calculated by measuring the minimum number of operations, e.g., moves or joins, required to transform the identified microservice architecture to the reference architecture [32, 64]. API division accuracy [65] is a measure to calculate the efficiency of API identification. It calculates the accuracy by relating the correctly identified API against all APIs. The cluster-to-cluster coverage (c2ccvg) measures the degree of overlap of the implementation-level entities between two clusters [64].

Certain studies [26,59] perform hyperparameter optimization to explore multiple alternative decompositions to identify optimal ones with respect to the properties discussed above. Furthermore, the Silhouette coefficient (SC) is used to evaluate the performance of the clustering algorithms [59,66].

Existing applications have been used to implement and evaluate the proposed reengineering solutions. Most of the reengineered systems were Java-based, with limited PHP systems identified. Applications reengineered in at least two works are listed in Table 10.

Extensive evaluations have been conducted in several studies, where the proposed solution was assessed against existing migration frameworks, benchmarked against established applications, or subjected to comprehensive prototyping and property calculations. The evaluation criteria employed by key studies are presented in Table 11.

3.3. (RQ3) what are the challenges and limitations of existing methods for reengineering software systems into microservice-based systems?

This section discusses challenges associated with microservices migration. Deciding to embark on a legacy system migration project poses several organizational challenges, including:

- *Defining strategic goals*: Business owners and analysts must set clear strategic goals and decide whether to pursue microservices migration. This requires identifying and clarifying the business and technical drivers behind the migration [67,68].
- *Organizational restructuring*: Microservices migration often necessitates changes in organizational structure [67,69]. Large teams need to be split into smaller, specialized teams capable of managing microservices. Hierarchical organizations may require significant restructuring to support this transition effectively.
- *Resource and cost management*: Preparing resources and managing migration costs are critical challenges. This includes costs for human resources, hardware, and tools, as well as expenses related to design, development, and infrastructure setup. Organizations must also identify and train key developers to handle the reengineered systems [67,70].

Moreover, various technical challenges associated with microservices migration have been identified, as summarized below:

- *Lack of expert knowledge and tools*: Migration to microservices often requires specialized expertise in DevOps and cloud technologies. Organizations must establish continuous integration (CI) and continuous delivery (CD) pipelines and adopt DevOps practices during the migration process [67,69].
- *Design decisions*: Making design decisions and modifying the legacy system is challenging due to the complexity of existing software and the lack of comprehensive design documentation [68].
- *Deployment and operational challenges*: Migrating to microservices introduces a complicated deployment process, increased operational overhead, difficulties in debugging and testing, and higher resource utilization [70].
- *Database decomposition*: Splitting the centralized database layer into distributed components can lead to data inconsistencies between services [71].
- *Managing statefulness*: In microservice architectures, managing state is more complex than in monolithic systems due to their distributed nature. Stateful systems produce outputs dependent on previous interactions, posing significant challenges in ensuring consistent state management [71].

Several limitations were identified in the current migration frameworks. The primary limitation is the lack of a standardized mechanism for ensuring optimal migration and assessing the quality of decomposition. Furthermore, depending on the approach employed, identified limitations across various service migration systems are outlined in Table 12; the last column mentions study IDs in which the corresponding limitations were stated.

4. Discussion and future directions

This section discusses the insights we inferred from this literature review. Specifically, we discuss the insights into the artifact-driven,

Table 11
Cross-system evaluation frameworks.

ID	Name	Evaluation type	Details
1	Service cutter	Prototype and case study	Evaluated the approach with cargo tracking system and trading system.
6		Compared with legacy system	ACME air web application compared in monolith and microservices versions.
7	Green micro	Cross comparison	Cross compared with FoSCI, CoGCN, Mono2Micro, MEM, Service Cutter, API, DFD, and Business process analysis.
11	Topic modeling	Case study	Evaluated using 200 Java Spring applications selected from GitHub for property calculations.
13		Cross comparison	Evaluated against Fosci, DFD approach and distributed source code representation.
25, 26, 27		Compared with legacy system	Compared Sugar CRM and Church CRM legacy and microservice versions.
28		Compared with legacy system	Compared Dolibarr open-source enterprise management system legacy and microservice versions.
29	Co_GCN	Prototype	Evaluated using Daytrader, Plants by websphere, Acme-Air, and Diet App
39	FOSCI	Cross comparison	Compared with LIMBO, WCA, and MEM approaches.
42	Mono2Micro	Cross comparison	Compared with FOSCI, CO_GCN, and Munch approaches.
48	DFD	Cross comparison	Cross compared with Service cutter and API analysis approach.
49	Knowledge graphs	Prototype	Evaluated the approach with E-commerce application and cargo tracking system.
51	Steinmetz	Case study	Evaluated properties using 14 applications.
53	CARGO	Cross comparison	Evaluated against Mono2Micro, CoCGN, MEM, and FOSCI.
61		Case study	Applied the approach to 121 monolith applications for comparison.
66	Hierarchical DBSCAN	Benchmark and cross comparison	Evaluated existing microservices projects — Spring PetClinic, Microservices Event Sourcing, and Kanban Board Cross compared with Bunch, CoGCN, FOSCI, MEM, and Mono2Micro frameworks.
70	API graph	Benchmark and cross comparison	Evaluated existing microservices projects Kanban, Money Transfer, Piggy Metrics, Microservices Event Sourcing, and Sock Shop Cross compared with Service Cutter.
77	Feature table	Cross comparison	Evaluated against DFD, Service Cutter, API analysis frameworks.
85	Interface analysis	Prototype	Precision and recall properties evaluated using Cargo tracking system.
86	MEM	Case study	Evaluated 21 projects for logical, semantic, and contributor coupling.
87		Benchmark	Evaluated existing microservices projects Kanban Board, and Money Transfer app. Amazon Web Services and PayPal evaluated using OpenAPI specifications.
88	FOME	Cross comparison	Evaluated LIMBO, WCA, and MEM frameworks.
89		Case study and cross comparison	Evaluated against existing Service Cutter and topic modeling frameworks. Five applications including PetClinic JForum 3, and Compiere applications evaluated for accuracy.
90		Case study and cross comparison	Evaluated against FOME and multi-objective evolutionary search frameworks. Property evaluated in JPetStore, SpringBlog, Jforum, Roller applications.
108	Log2MS	Case study and cross comparison	Evaluated against FOSCI and Mono2Micro frameworks. Property evaluated in four applications including JPetStore.
115	Backlog	Case study and cross comparison	Evaluated against Domain-driven design, Interface analysis, and Service Cutter frameworks. JPetStore, Cargo Tracking System, and Foristom Conferences(real life system) used for evaluation.

static, dynamic, hybrid, and database analysis approaches, emerging approaches, ways to evaluate the reengineered systems, and reengineering paradigms. Finally, the section proposes directions for future research based on our findings.

4.1. Artifact-driven analysis

The artifact-driven approaches constitute 32% of the reviewed studies. Service Cutter [31] is one of the pilot artifact-driven studies in microservice identification. Hence, it has been used as a baseline in multiple studies. The dataflow-driven technique [46] is another prominent artifact-driven approach comprising quality attribute evaluation. Greenmicro [72], Microservice Backlog [34], and the Feature Table approach [47] have shown promising experimental results in comparisons with other migration studies. Greenmicro and Microservice Backlog are notable studies that involve comprehensive cross-system analysis.

4.2. Static analysis

The state-of-the-art technique for reengineering software systems into microservices is static analysis. Among the primary studies reviewed, 44% discuss static analysis techniques, with structural analysis dominating over semantic and evolutionary coupling approaches. Prominent structural analysis techniques include CoGCN [22], Cargo-AI Guided Dependency Analysis [23], and dependency-based microservice decomposition [24]. Notably, Cargo-AI Guided Dependency Analysis stands out, as evaluations against benchmark studies confirm its effectiveness. Other approaches, such as microservice identification through topic modeling [25] and the method by Sellami et al. [26], utilize ASTs combined with graph-based and matrix-based algorithms, respectively, for structural and semantic analysis. Evolutionary coupling techniques, though less prevalent, offer significant contributions. For example, MEM [21] constructs logical, semantic, and evolutionary coupling graphs, employing a minimum spanning tree-based algorithm for microservice detection. Similarly, the automatic extraction approach [32] uses fast community graph clustering on graphs generated with structural and semantic information, while Löhnertz and Oprescu [33] integrates static, semantic, and evolutionary coupling graphs, experimenting with multiple clustering algorithms. Their findings highlight the Louvain clustering algorithm as particularly effective. Despite their promise, static analysis techniques face challenges, notably imprecise program analysis, as identified by Nitin et al. [23]. These approaches also rely heavily on existing tools, underscoring the need for advancements in program analysis precision.

4.3. Dynamic analysis

Limited experiments have been conducted using dynamic analysis techniques. One approach supplies software logs as input to the process

Table 12
Limitations of existing migration approaches.

Limitation	Description	Study IDs
Significant effort required to transform software system artifacts	The artifact-driven approach requires significant manual effort to transform system artifacts for further processing. For example, Service Cutter relies on system artifacts such as use cases and domain models, which must be manually converted into JSON format to enable subsequent processing and clustering.	1
Availability of supportive tools	Tools incorporated into the process, such as Disco, can produce inaccurate results, directly impacting the output. Similarly, static analysis tools suffer from imprecise program analysis, which can compromise the quality of the migration process.	2, 24, 25, 27, 28
Applicability of the solution	The applicability of proposed solutions is limited and often context-specific. For instance, solutions that utilize request URLs are applicable only to Web applications, while those based on Java annotations and language keywords are restricted to Java-based systems. Likewise, EJB-based identification methods are exclusively applicable to Java EE EJB-based architectures.	5, 6, 11, 31, 33
Quality of the artifacts	The effectiveness of proposed solutions are highly dependent on the quality of the input artifacts. For example, studies using semantic analysis are heavily influenced by the terminology used in the source code. Similarly, the quality and comprehensiveness of artifacts, such as data flow diagrams, directly affect the quality of the identified microservices. Solutions based on object-oriented principles rely on the correct application of object-oriented programming concepts within the source code.	11, 48, 67
Challenges in database decomposition	Database decomposition remains a significant challenge. ORM relationships in the source code are often leveraged to reduce complexity, but not all source codes support ORM frameworks. Furthermore, existing solutions primarily focus on relational databases, leaving NoSQL databases largely unaddressed in the decomposition process.	76, 86
Coverage of the inputs	The coverage of system inputs directly impacts the quality of the outputs. For example, in dynamic analysis, the extent to which use cases generate system logs significantly influences the results.	2, 39, 108
Complexity of the algorithms	The algorithmic complexity is a key factor contributing to performance limitations. Many existing algorithms and libraries are heavily utilized for clustering and extraction tasks, and the time complexity of these algorithms directly affects the overall performance of the migration process.	1

mining tool Disco for further analysis. However, certain processes have been incorrectly identified by this approach [19,73]. FoSCI [29], FoME [30], and mono2micro [35], Log2MS [74] are the prominent studies in dynamic analysis. Moreover, mono2micro is a commercially available product. It collects software log traces by executing use cases and identifies unique traces to derive direct and indirect calls to generate a similarity matrix followed by hierarchical clustering. Furthermore, its strategy has been compared with FoSCI [29], CoGCN [22], Bunch [75], and MEM [21] to validate the results. FoSCI uses reduced execution traces to identify functional atoms using the NSGA II multi-objective optimization algorithm. FoME collects logs from test executions and generates descriptive log traces for clustering and shared class processing. Both FoSCI and FoME use the Kieker runtime monitoring tool for property evaluation and comparing results against MEM [21], LIMBO [76], and WCA [77]. Log2MS [74] proposes a Model-Driven Development (MDD)-based brownfield design approach for identifying microservices using only execution logs. It utilizes a microservice diagram, microservice sequence diagram, and microservice architecture modeler to generate microservices, drawing inspiration from greenfield software development practices.

4.4. Hybrid analysis

Among the available hybrid analysis studies, microservice extraction using knowledge graphs [78] stands out as a comprehensive approach as an approach that integrates static and artifact-driven analysis. It constructs a graph from diverse inputs, including source code, database schemas, design documents, and API documentation, incorporating data, modules, functions, and resource details. The Louvain community detection algorithm is then applied to identify microservice candidates. Several other hybrid approaches combining static and dynamic analysis have also demonstrated promising results [36,43,55,79,80]. The node attribute network approach [79] uses call graphs to analyze method invocations and employs the Kieker runtime analysis tool to generate a graph structure that is processed using the Leiden community detection algorithm. It is one of the most comprehensively evaluated hybrid methods, compared against techniques like FoSCI [29], the dataflow-driven approach [46], and the distributed representation approach [81]. MonoBreaker [43] combines static structural analysis with runtime monitoring data to generate a graph model. Clustering is performed using the Girvan–Newman algorithm, with evaluations against Service Cutter [31] demonstrating that hybrid analysis yields better results than static analysis alone. Similarly, the Migrating Web Applications approach [55] enhances dependency graphs created through static analysis with dynamic analysis data, using the K-means clustering algorithm to identify microservice candidates. However, this study focuses solely on evaluating the properties of the reengineered system. Other notable hybrid approaches [36,80] employ the NSGA-III multi-objective optimization algorithm to evaluate various system properties, further showcasing the potential of hybrid analysis techniques in improving microservice identification and system reengineering.

4.5. Database analysis

Database migration poses significant challenges, particularly in transitioning from monolithic architectures to microservices-based systems. Key issues include maintaining data consistency, handling distributed transactions, and ensuring seamless integration of diverse database types. To address these challenges, the following patterns have been identified [82,83]:

1. Schema per microservice: Each microservice maintains its schema while sharing the same database server.
2. Database per microservice: Each microservice is assigned a dedicated database, promoting modularity and autonomy.
3. Database as a microservice: The database is encapsulated as a standalone microservice, with all interactions managed via APIs.

4. Optimized read-only database replica: A replica of the primary database is optimized for read operations, while the primary database handles both reads and writes.

Among these, the "database per microservice" pattern is widely preferred in the literature [23,50,84] due to its alignment with microservices principles. However, it introduces challenges in handling distributed transactions. As a mitigation strategy, eventual consistency is often employed, where failed requests are queued for reattempts [84].

An innovative approach to migrating monolithic databases to multi-model polyglot persistence systems [82] draws inspiration from polyglot programming principles. This approach conceptualizes the database as a microservice, enabling seamless integration of SQL and NoSQL databases through an API. By tailoring database types to the specific data needs of the software system, this approach enhances flexibility and scalability in microservices-based architectures.

Service extraction has also considered the persistence layer in applications [85], including mappings between SQL queries and objects [51,52,86]. One notable study focuses on identifying microservice candidates from business rules embedded in stored procedures [87]. Additionally, an Object Relational Mapping (ORM)-based system has been proposed to evaluate reengineered systems using specific properties [88]. Widely used databases in microservices architectures include Redis, MongoDB, MySQL, PostgreSQL, and MS SQL [82], highlighting the diversity of tools that support modern database management in distributed systems.

4.6. Emerging techniques

Microservices Backlog [34] employs genetic programming to iteratively identify the optimal combination of microservices through the application of an objective function. The objective function utilizes a granularity matrix, incorporating coupling, cohesion, granularity, performance, complexity, and development time. A comprehensive evaluation was conducted [34], wherein the results were cross-compared with those obtained from Service Cutter, Interface Analysis, and FOSCI.

Another use of genetic algorithms is search-based microservices detection using Non-dominated Sorting Genetic Algorithms (NSGA). The NSGA algorithms employ multiple decision-making criteria for mathematical optimization problems involving two or three objective functions to be optimized simultaneously [36,37]. In general, studies have utilized NSGA-II and NSGA-III with two or three criteria [29,36,51,80,86]. The toMicroservice approach stands out as it incorporates five criteria for search-based detection, including coupling, cohesion, feature modularization, network overhead, and reuse.

Microminer [56], and the distributed representation of the source code [81] have introduced machine learning to microservice extraction. Microminer uses a machine learning-based word2vec model with the Louvain community detection algorithm, while the distributed representation of the source code uses a code2vec model with the affinity propagation algorithm. However, these approaches have no cross-comparison with prominent migration techniques. Instead, property calculations were performed to evaluate the proposed solutions.

Reverse engineering of software systems to derive microservices is rarely used. Only three reviewed studies are grounded in reverse engineering of monolithic systems [38,64,89]. The model-driven reverse engineering approach [38] integrates reverse engineering with reinforcement learning to create a mapping between the identified legacy system model and a set of microservices. Applying reverse engineering techniques to uncover the architecture of a system can facilitate the advancement of microservice discovery methods, particularly in cases where legacy systems are hindered by inadequate documentation regarding their architectural structure.

4.7. Evaluation

MicroValid [90] is the only framework identified in the primary studies that offers a validation methodology specifically for microservices. It performs static analysis of the identified microservice attributes to assess the quality of decomposition, focusing on factors such as granularity, coupling, and cohesion. The evaluation of migrated systems has been mainly based on property calculation. Several prominent studies have cross-compared with previous studies [23,26,29,30,34,35,47,72,79,91]. Service Cutter [31] is the classical migration study used for cross-comparison. Interface numbers, inter-partition call percentages, and structural modularity are the widely used properties. Even though coupling has been evaluated in many studies, there is no convergence in the evaluated definitions of this concept. Afferent coupling (measuring incoming dependencies) and efferent coupling (outgoing dependencies) are frequently used coupling measurements. Precision, recall, and F-measure are used for evaluation when a standard decomposition is available for comparison. This can be an available microservice system or an expert decomposition result.

Existing microservice-based benchmark systems like Spring Pet Clinic,[13] Kanban,[14] Money Transfer,[15] Piggy Metrics,[16] Microservices Event Sourcing (MES),[17] Sock Shop[18] have been used for evaluation [27]. Limited studies focused on hyper-parameter optimization [26,59]. Yedida et al. [92] discussed performance improvements by optimizing hyper-parameters.

The majority of the migration frameworks applied their concepts to monolithic open-source projects. JPetStore is the most frequently used project for implementation and testing. Moreover, Acme Air, Cargo Tracking System, and Daytrader applications were used frequently in the reengineering projects. Web-based applications like online shopping systems, learning management systems, banking systems, ERP systems, real-estate applications, web-based IDEs, taxation office systems, and police department systems were also used as proofs of concept. Above 80% of the re-engineered applications in the literature are Java-based projects. Database-oriented applications, like stored procedure decompositions, have been discussed in relatively few studies [51,87].

4.8. Paradigms

Several paradigms for microservices reengineering have been identified during our analysis, such as Domain-Driven Design (DDD), workflow analysis, feature analysis, system semantic analysis, repository analysis, interface analysis, and runtime analysis. Domain-driven design focuses on the business domain and identifies the boundaries of the microservices. Workflow analysis uses business processes and workflows to identify microservices. Analysis and grouping of dependent system features were used in feature analysis. System semantics analysis includes semantics of system features and/or source code semantics analysis. Repository analysis includes the source code structure, version control history, and data source analysis. Interface analysis uses web service definitions and messages disseminated via the interfaces. Finally, runtime analysis includes analysis of execution traces and logs.

Incremental and iterative transitions are the preferred industry approach for migrating legacy systems to microservices [67,68,93–95], as opposed to direct migration. Specifically, the Strangler Fig Pattern [95,96] is inspired by the growth behavior of the Strangler Fig plant, which gradually encircles and overtakes a tree, ultimately leading to the decline of the tree over time. Similarly, microservices are introduced to the legacy system incrementally and can lead to the ultimate decline of the legacy software system.

[13] https://github.com/spring-petclinic/spring-petclinic-microservices.
[14] https://github.com/eventuate-examples/es-kanban-board.
[15] https://github.com/cer/event-sourcing-examples.
[16] https://github.com/sqshq/PiggyMetrics.
[17] https://github.com/chaokunyang/microservices-event-sourcing.
[18] https://github.com/microservices-demo/microservices-demo.

4.9. Gaps and future directions

Next, we highlight several gaps we identified in research on reengineering of software systems into microservices systems and suggest directions for future work in this area.

Dynamic analysis and AI-based techniques remain underutilized and are rarely integrated into existing approaches. This gap presents significant opportunities for innovation and further research.

Existing studies have primarily focused on identifying microservice candidates, with runtime performance evaluations limited to metrics like latency, throughput, availability, and network overhead. However, behavioral consistency in re-engineered systems remains largely underexplored. Given the automated nature of extraction processes, developing robust validation mechanisms is critical to ensuring system integrity and reliability.

Evaluating the dynamic rearrangement of microservices under varying workloads is crucial for improving the efficiency of migrated systems. Under low system loads, maintaining a monolithic system may be more efficient, while transitioning to a microservices-based architecture can optimize performance under higher loads. Future work can study strategies for achieving optimal resource utilization of microservices under different workloads.

While the "database per microservice" pattern is often recommended, the practical challenges of partitioning databases into microservices remain underexplored. Key issues, such as the performance impact of distributed transactions and methods for ensuring data consistency, are yet to be thoroughly investigated. Addressing these gaps is essential for optimizing microservice architectures and ensuring their reliability.

The effort required to redesign functionalities during microservice decomposition has been partially addressed in previous work [97], which introduced a complexity metric for migration. However, there remains a need for a systematic approach to accurately calculate the cost and complexity of the entire migration process, incorporating both technical and resource-related factors.

The impact of the granularity of microservices on system performance has not been a focus in previous studies. The Microservice Backlog approach [34] takes granularity into account. However, the impact of microservice granularity on the performance of the system requires further studies.

While the identification of microservices has been automated, the extraction of microservices from the original monolithic system remains mostly a manual task. Therefore, there is a need for the development of automated code refactoring approaches to facilitate the generation of microservices and their communication interfaces.

5. Conclusion

A broad analysis of existing approaches for reengineering software systems into microservices systems has been performed in this literature review. Initially, 4843 papers were selected from five research paper libraries. After multiple stages of filtering, 117 primary studies were selected for further analysis. The identified studies were analyzed based on multiple perspectives, including employed techniques and tools, data usage, evaluation, limitations, and challenges. We have identified well-explored, state-of-the-art techniques like static analysis and areas with limited focus to date, like dynamic analysis. In addition, the unavailability of convergence in the studies proves that microservice migration research is still in its infancy. Finally, microservice reengineering is a significant study area that can be improved further. Future studies can focus on exploring new techniques and evaluation strategies for microservice discovery, implementation, deployment, and assessment.

CRediT authorship contribution statement

Thakshila Imiya Mohottige: Writing – original draft. **Artem Polyvyanyy:** Writing – review & editing, Writing – original draft, Supervision. **Colin Fidge:** Writing – review & editing, Supervision. **Rajkumar Buyya:** Writing – review & editing, Supervision. **Alistair Barros:** Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Alistair Barros, Colin Fidge, Artem Polyvyanyy reports financial support was provided by Australian Research Council. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was in part supported by the Australian Research Council project DP220101516.

Appendix. Study list

ID	Full reference
1	M. Gysel et al. Service cutter: A systematic approach to service decomposition, in: Service-Oriented and Cloud Computing, 2016 [31].
2	D. Taibi, K. Systä, From monolithic systems to microservices: A decomposition framework based on process mining, International Conference on Cloud Computing and Services Science, 2019 [19].
3	F. Auer et al. From monolithic systems to microservices: An assessment framework, Information and Software Technology, 2021 [18].
4	H. Michael Ayas et al. The migration journey towards microservices, in: Product-Focused Software Process Improvement: 22nd International Conference, 2021 [68].
5	B. Deepali et al. Partial Migration for Re-architecting a Cloud Native Monolithic Application into Microservices and FaaS.
6	A. Muhammad et al. Unsupervised Learning Approach for Web Application Auto-Decomposition into Microservices. Journal of Systems and Software, 2019
7	D. Bajaj et al. Greenmicro: Identifying microservices from use cases in greenfield development, IEEE, 2022 [72].
8	M. H. Gomes Barbosa, P. H. M. Maia, Towards identifying microservice candidates from business rules implemented in stored procedures, IEEE International Conference on Software Architecture Companion, 2020 [87].
9	R. Belafia et al. From Monolithic to Microservice Architecture: The Case of Extensible and Domain-Specific IDEs. ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), 2021.
10	K. Bozan, K. Lyytinen, G. M. Rose, How to transition incrementally to microservice architecture, Commun. ACM, 2020.
11	M. Brito et al. Identification of microservices from monolithic applications through topic modelling, 36th Annual ACM Symposium on Applied Computing, 2021 [25].

12	V. Bushong et al. “Using Static Analysis to Address Microservice Architecture Reconstruction,” International Conference on Automated Software Engineering, 2021.
13	L. Cao, C. Zhang, Implementation of domain-oriented microservices decomposition based on node-attributed network, 11th International Conference on Software and Computer Applications, 2022 [79].
14	A. Carrasco et al. Migrating towards microservices: Migration and architecture smells, 2nd International Workshop on Refactoring, 2018 [94].
15	L. Carvalho et al. Analysis of the criteria adopted in industry to extract microservices, 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice, 2019 [98].
16	C. Tomas Cerny et al. On isolation-driven automated module decomposition.Conference on Research in Adaptive and Convergent Systems, 2018.
17	C. Nacha et al. Software Architectural Migration: An Automated Planning Approach. ACM Trans. Softw. Eng. Methodol, 2021.
18	Christoforou, A. et al. Supporting the Decision of Migrating to Microservices Through Multi-layer Fuzzy Cognitive Maps. Service-Oriented Computing. ICSOC, 2017.
19	da Silva, C.E. et al. SPReaD: service-oriented process for reengineering and DevOps. SOCA, 2022.
20	Hugo H. O. S. da Silva et al.,Towards a Roadmap for the Migration of Legacy Software Systems to a Microservice based Architecture. 9th International Conference on Cloud Computing and Services Science, 2019 [48].
21	M. Daoud et al.,Towards an Automatic Identification of Microservices from Business Processes, 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2020.
22	Dattatreya, V. et al. Design Patterns and Microservices for Reengineering of Legacy Web Applications, 2021.
23	de Almeida, M.G., Canedo, The Adoption of Microservices Architecture as a Natural Consequence of Legacy System Migration at Police Intelligence Department.ICCSA 2022.
24	A. A. C. De Alwis et al. Discovering microservices in enterprise systems using a business object containment heuristic, On the Move to Meaningful Internet Systems. OTM 2018 [51].
25	A. A. C. De Alwis et al. Function-splitting heuristics for discovery of microservices in enterprise systems, Service-Oriented Computing, 2018 [52].
26	A. A. C. De Alwis et al. Availability and scalability optimized microservice discovery from enterprise systems, On the Move to Meaningful Internet Systems: OTM 2019 [86].
27	De Alwis, A.A.C. et al. Remodularization Analysis for Microservice Discovery Using Syntactic and Semantic Clustering. Advanced Information Systems Engineering. CAiSE 2020.
28	De Alwis, A.A.C. et al. Microservice Remodularisation of Monolithic Enterprise Systems for Embedding in Industrial IoT Networks. CAiSE 2021.
29	U. Desai et al. Graph neural network to dilute outliers for refactoring monolith application, Conference on Artificial Intelligence, 2021 [22].
30	E. Djogic, S. Ribic and D. Donko, “Monolithic to microservices redesign of event driven integration platform,” 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2018.
31	D. Escobar et al. Towards the understanding and evolution of monolithic applications as microservices, XLII Latin American Computing Conference, 2016 [89].
32	S. Eski, F. Buzluca, An automatic extraction approach: Transition to microservices architecture from monolithic application, 19th International Conference on Agile Software Development, 2018 [32].
33	F. Freitas et al. A. Ferreira, J. Cunha, Refactoring Java Monoliths into Executable Microservice-Based Applications, 2021 [85].
34	A. Furda et al. Migrating enterprise legacy source code to microservices: On multitenancy, statefulness, and data consistency, IEEE Software 35, 2018 [71].
35	Gutiérrez-Fernández et al. Redefining a Process Engine as a Microservice Platform. Business Process Management Workshops, 2016.
36	A. O. R. Ishida et al. K., Extracting Micro Service Dependencies Using Log Analysis,IEEE 29th Annual Software Technology Conference (STC),2022.
37	Md Rofiqul Islam and Tomas Cerny. Business process extraction using static analysis. 36th IEEE/ACM International Conference on Automated Software Engineering, 2021.
38	A. Janes and B. Russo, Automatic Performance Monitoring and Regression Testing During the Transition from Monolith to Microservices, ISSREW, 2019.
39	W. Jin et al. Service candidate identification from monolithic systems based on execution traces, IEEE Transactions on Software Engineering 47, 2021 [29].
40	M. I. Joselyne et al. A systematic framework of application modernization to microservice based architecture, International Conference on Engineering and Emerging Technologies (ICEET), 2021 [44].
41	I. J. Munezero et al. Partitioning Microservices: A Domain Engineering Approach, IEEE/ACM Symposium on Software Engineering in Africa, 2018.
42	A. K. Kalia et al. Mono2Micro: A practical and effective tool for decomposing monolithic Java applications to microservices, 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021 [35].
43	Anup K. Kalia et al. Mono2Micro: an AI-based toolchain for evolving monolithic enterprise applications to a microservice architecture. 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020.
44	M. Kamimura et al. Extracting Candidates of Microservices from Monolithic Application Code, 25th Asia-Pacific Software Engineering Conference, 2018.
45	Khoshnevis, S. A search-based identification of variable microservices for enterprise SaaS. Front. Comput. Sci. 17, 2023
46	A. Krause et al. Microservice decomposition via static and dynamic analysis of the monolith, IEEE International Conference on Software Architecture Companion, 2020 [53].

47	N. Lapuz et al. Digital transformation and the role of dynamic tooling in extracting microservices from existing software systems, Systems, Software and Services Process Improvement,2021 [54].
48	S. Li, H. Zhang et al. A dataflow-driven approach to identifying microservices from monolithic applications, Journal of Systems and Software 157, 2019 [46].
49	Z. Li, C. Shang, J. Wu, Y. Li, Microservice extraction based on knowledge graph from monolithic applications, Information and Software Technology 150,2022 [78].
50	B. Liu et al. Method of Microservices Division for Complex Business Management System Based on Dual Clustering, International Conference on Mechanical, Control and Computer Engineering, 2020
51	J. Löhnertz, A. Oprescu, Steinmetz: Toward automatic decomposition of monolithic software into microservices, 2020 [33].
52	T. Matias et al. Determining microservice boundaries: A case study using static and dynamic software analysis, Software Architecture,2020 [43].
53	V. Nitin et al, CARGO: AI-guided dependency analysis for migrating monolithic applications to microservices architecture, 37th IEEE/ACM International Conference on Automated Software Engineering, 2023 [23].
54	park, J. et al. Approach to Identify Microservices based on Analysis Class Model. International Journal of Advanced Science and Technology, 28, 2019.
55	I. Pigazzini et al. Tool support for the migration to microservice architecture: An industrial case study, European Conference on Software Architecture, 2019 [58].
56	T. Prasandy et al.,Migrating Application from Monolith to Microservices, International Conference on Information Management and Technology 2020.
57	Z. Ren et al. Migrating web applications from monolithic structure to microservices architecture, 10th Asia-Pacific Symposium on Internetware, 2018 [55].
58	Y. Romani et al.,Towards Migrating Legacy Software Systems to Microservice-based Architectures: a Data-Centric Process for Microservice Identification, 19th International Conference on Software Architecture Companion (ICSA-C), 2022.
59	Saidi, M. et al. Automatic Microservices Identification Across Structural Dependency. Hybrid Intelligent Systems. 2022.
60	N. Santos, A. Rito Silva, A complexity metric for microservices architecture migration, 2020 IEEE International Conference on Software Architecture (ICSA), 2020 [97].
61	S. Santos, A. R. Silva, Microservices identification in monolith systems: Functionality redesign complexity and evaluation of similarity measures, Journal of Web Engineering 21, 2022 [99].
62	Sarita and S. Sebastian, Transform Monolith into Microservices using Docker, International Conference on Computing, Communication, Control and Automation, 2017.
63	Casper Schröder et al. Search-based software re-modularization: a case study at Adyen. 43rd International Conference on Software Engineering: Software Engineering in Practice, 2021.
64	Christoph Schröer, Towards Microservice Identification Approaches for Architecting Data Science Workflows, Procedia Computer Science, 2021.
65	C. Schroer, S. Wittfoth and J. M. Gomez, A Process Model for Microservices Design and Identification, IEEE 18th International Conference on Software Architecture Companion, 2021.
66	K. Sellami et al. A Hierarchical DBSCAN Method for Extracting Microservices from Monolithic Applications, 26th International Conference on Evaluation and Assessment in Software Engineering, 2022 [26].
67	Selmadji, A. et al. Re-architecting OO Software into Microservices. Service-Oriented and Cloud Computing, 2018.
68	A. L. Shastry et al. Approaches for migrating non cloud-native applications to the cloud, IEEE 12th Annual Computing and Communication Workshop and Conference, 2022.
69	T. D. Stojanovic et al. Identifying microservices using structured system analysis, 24th International Conference on Information Technology (IT), 2020 [100].
70	X. Sun et al. Expert system for automatic microservices identification using API similarity graph, Expert Systems, 2022 [65].
71	D. Taibi et al. Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation, IEEE Cloud Computing 4, 2017 [10].
72	D. Taibi, K. Systä, A decomposition and metric-based evaluation framework for microservices, Cloud Computing and Services Science, 2020 [73].
73	M. Tusjunt, W. Vatanawood, Refactoring orchestrated web services into microservices using decomposition pattern, IEEE 4th International Conference on Computer and Communications (ICCC), 2018 [49].
74	S. Tyszberowicz et al. Identifying microservices using functional decomposition, Dependable Software Engineering. Theories, Tools, and Applications, 2018 [50].
75	F. H. Vera-Rivera et al. Microservices backlog—A model of granularity specification and microservice identification, 17th International Conference, 2020 [91].
76	E. Volynsky et al. Architect: A Framework for the Migration to Microservices, International Conference on Computing, Electronics Communications Engineering (iCCECE), 2022.
77	Y. Wei et al. A feature table approach to decomposing monolithic applications into microservices, 12th Asia-Pacific Symposium on Internetware, 2021 [47].
78	R. Yedida et al. Lessons learned from hyper-parameter tuning for microservice candidate identification, 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021 [92].
79	P. Zaragoza et al. Leveraging the layered architecture for microservice recovery, IEEE 19th International Conference on Software Architecture (ICSA), 2022 [64].
80	O. Al-Debagy, P. Martinek, Dependencies-based microservices decomposition method, International Journal of Computers and Applications 44, 2021 [24].
81	Almeida, J.F., Silva, A.R., Monolith Migration Complexity Tuning Through the Application of Microservices Patterns. ECSA 2020.

82	W. K. G. Assunção et al. Analysis of a many-objective optimization approach for identifying microservices from legacy systems, Empirical Softw. Engg. (2022)[36].
83	W. K. G. Assunção et al. A multi-criteria strategy for redesigning legacy features as microservices: An industrial case study, IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2021 [80].
84	Wesley K. G. Assunção et al. Variability management meets microservices: six challenges of re-engineering microservice-based webshops. 24th ACM Conference on Systems and Software Product Line, 2020.
85	L. Baresi et al. Microservices identification through interface analysis, Service-Oriented and Cloud Computing, 2017 [28].
86	G. Mazlami et al. Extraction of microservices from monolithic software architectures, IEEE International Conference on Web Services (ICWS), 2017 [21].
87	O. Al-Debagy, P. Martinek, A new decomposition method for designing microservices, Period. Polytech. Electr. Eng. Comput. Sci. 63, 2019 [59].
88	W. Jin et al. Functionality-oriented microservice extraction based on execution trace clustering, IEEE International Conference on Web Services (ICWS), 2018 [30].
89	I. Trabelsi et al. From legacy to microservices: A type-based approach for microservices identification using machine learning and semantic analysis, Journal of Software: Evolution and Process, 2022 [56].
90	O. Al-Debagy, A microservice decomposition method through using distributed representation of source code, Scalable Comput. Pract.Exp., 2021 [81].
91	Levcovitz, A. et al. Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems. 2016.
92	M. J. Amiri, "Object-Aware Identification of Microservices," IEEE International Conference on Services Computing(SCC), 2018.
93	S. Agarwal et al. Monolith to Microservice Candidates using Business Functionality Inference, 2021 IEEE International Conference on Web Services (ICWS), 2021.
94	C. Bandara and I. Perera, Transforming Monolithic Systems to Microservices - An Analysis Toolkit for Legacy Code Evaluation, 20th International Conference on Advances in ICT for Emerging Regions (ICTer), 2020.
95	M. Camilli et al. Domain metric driven decomposition of data-intensive applications, IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2020 [83].
96	L. Carvalho et al. On the performance and adoption of search-based microservice identification with toMicroservices, IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020 [37].
97	C. Andreas et al. Migration of Software Components to Microservices: Matching and Synthesis, 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), 2019.
98	M. Cojocaru et al. MicroValid: A Validation Framework for Automatically Decomposed Microservices, IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2019.
99	M. Dehghani et al. Facilitating the migration to the microservice architecture via model-driven reverse engineering and reinforcement learning, Softw. Syst. Model.,2022 [38].
100	F. -D. Eyitemi and S. Reiff-Marganiec, System Decomposition to Optimize Functionality Distribution in Microservices with Rule Based Approach, IEEE International Conference on Service Oriented Systems Engineering (SOSE), 2020.
101	G. Filippone et al. Migration of Monoliths through the Synthesis of Microservices using Combinatorial Optimization, IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2021.
102	N. Ivanov and A. Tasheva, A Hot Decomposition Procedure: Operational Monolith System to Microservices, International Conference Automatics and Informatics (ICAI), 2021.
103	J. Kazanavičius et al. An Approach to Migrate a Monolith Database into Multi-Model Polyglot Persistence Based on Microservice Architecture: A Case Study for Mainframe Database [82]
104	D. Kuryazov et al. Towards Decomposing Monolithic Applications into Microservices, IEEE 14th International Conference on Application of Information and Communication Technologies (AICT), 2020.
105	L. De Lauretis, From Monolithic Architecture to Microservices Architecture, IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2019.
106	C.-Y. Li et al. Microservice migration using strangler fig pattern: A case study on the green button system, International Computer Symposium (ICS), 2020 [95].
107	J. Lin et al. Migrating web applications to clouds with microservice architectures, International Conference on Applied System Innovation (ICASI), 2016.
108	Log2MS: a framework for automated refactoring monolith into microservices using execution logs [74] B. Liu et al. Log2ms: a framework for automated refactoring monolith into microservices using execution logs, IEEE International Conference on Web Services(ICWS), 2022 [74].
109	S. A. Maisto et al. From monolith to cloud architecture using semi-automated microservices modernization, Advances on P2P, Parallel, Grid, Cloud and Internet Computing, 2020 [69].
110	R. Mishra et al. Transition from Monolithic to Microservices Architecture: Need and proposed pipeline, International Conference on Futuristic Technologies (INCOFT), 2022.
111	Nunes, L. et al.,From a Monolith to a Microservices Architecture: An Approach Based on Transactional Contexts, ECSA, 2019.
112	Ana Santos and Hugo Paula, Microservice decomposition and evaluation using dependency graph and silhouette coefficient, 15th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS), 2021.

113	A. Selmadji et al. From Monolithic Architecture Style to Microservice one Based on a Semi-Automatic Approach, IEEE International Conference on Software Architecture (ICSA), 2020.
114	A. Shimoda and T. Sunada, Priority Order Determination Method for Extracting Services Stepwise from Monolithic System, 7th International Congress on Advanced Applied Informatics (IIAI-AAI), 2018.
115	F. H. Vera-Rivera et al. Microservices backlog—A genetic programming technique for identification and evaluation of microservices from user stories, IEEE Access. 2021 [34].
116	Z. Yang et al. A Microservices Identification Approach based on Problem Frames, IEEE 2nd International Conference on Software Engineering and Artificial Intelligence (SEAI), 2022.
117	J. Zhao and K. Zhao, Applying Microservice Refactoring to Object-2riented Legacy System, 8th International Conference on Dependable Systems and Their Applications (DSA), 2021.

Data availability

Data will be made available on request.

References

[1] R.A. Schmidt, 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, 2020.

[2] C. Schröer, F. Kruse, J. Marx Gómez, A qualitative literature review on microservices identification approaches, in: Service-Oriented Computing, 2020, pp. 151–168.

[3] M. Cojocaru, A. Oprescu, A. Uta, Attributes assessing the quality of microservices automatically decomposed from monolithic applications, in: 2019 18th International Symposium on Parallel and Distributed Computing, ISPDC, 2019, pp. 84–93.

[4] R. Capuano, H. Muccini, A systematic literature review on migration to microservices: A quality attributes perspective, in: 2022 IEEE 19th International Conference on Software Architecture Companion, ICSA-C, 2022, pp. 120–123.

[5] F. Ponce, G. Márquez, H. Astudillo, Migrating from monolithic architecture to microservices: A rapid review, in: 2019 38th International Conference of the Chilean Computer Science Society, SCCC, 2019, pp. 1–7.

[6] J. Fritzsch, J. Bogner, A. Zimmermann, S. Wagner, From monolith to microservices: A classification of refactoring approaches, 2018, CoRR abs/1807.10059.

[7] D. Wolfart, W.K.G. Assunção, I.F. da Silva, D.C.P. Domingos, E. Schmeing, G.L.D. Villaca, D.d.N. Paza, Modernizing legacy systems with microservices: A roadmap, in: Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering, EASE 2011, 2021, pp. 149–159.

[8] M. Abdellatif, A. Shatnawi, H. Mili, N. Moha, G.E. Boussaidi, G. Hecht, J. Privat, Y.-G. Guéhéneuc, A taxonomy of service identification approaches for legacy software systems modernization, J. Syst. Softw. 173 (2021) 110868.

[9] K. Bennett, Legacy systems: Coping with success, IEEE Softw. 12 (1995) 19–23.

[10] D. Taibi, V. Lenarduzzi, C. Pahl, Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation, IEEE Cloud Comput. 4 (2017) 22–32.

[11] J. Fritzsch, J. Bogner, S. Wagner, A. Zimmermann, Microservices migration in industry: Intentions, strategies, and challenges, in: 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME, 2019, pp. 481–490.

[12] S. Newman, Building Microservices: Designing Fine-Grained Systems, O'Reilly Media, Inc., 2015.

[13] J. Lewis, M. Fowler, Microservices: A definition of this new architectural term, 2014, https://martinfowler.com/articles/microservices.html. [Online; Accessed 27 July 2023].

[14] D. Arcelli, V. Cortellessa, D. Di Pompeo, Performance-driven software model refactoring, Inf. Softw. Technol. (2018) 366–397.

[15] A. De Lucia, G. Di Lucca, A. Fasolino, P. Guerra, S. Petruzzelli, Migrating legacy systems towards object-oriented platforms, in: 1997 Proceedings International Conference on Software Maintenance, 1997, pp. 122–129.

[16] S. Adjoyan, A.-D. Seriai, A. Shatnawi, Service identification based on quality metrics—Object-oriented legacy system migration towards SOA, in: International Conference on Software Engineering and Knowledge Engineering, 2014.

[17] C. Abrams, R. Schulte, Service-oriented architecture overview and guide to SOA research, 2008, https://doveltech.com/wp-content/uploads/2017/10/serviceoriented_architecture.pdf. [Online; Accessed 27 July 2023].

[18] F. Auer, V. Lenarduzzi, M. Felderer, D. Taibi, From monolithic systems to microservices: An assessment framework, Inf. Softw. Technol. 137 (2021).

[19] D. Taibi, K. Systä, From monolithic systems to microservices: A decomposition framework based on process mining, in: International Conference on Cloud Computing and Services Science, 2019, pp. 153–164.

[20] D.L. Parnas, On the criteria to be used in decomposing systems into modules, Commun. ACM 15 (1972).

[21] G. Mazlami, J. Cito, P. Leitner, Extraction of microservices from monolithic software architectures, in: 2017 IEEE International Conference on Web Services, ICWS, 2017, pp. 524–531.

[22] U. Desai, S. Bandyopadhyay, S.G. Tamilselvam, Graph neural network to dilute outliers for refactoring monolith application, in: AAAI Conference on Artificial Intelligence, 2021, pp. 72–80.

[23] V. Nitin, S. Asthana, B. Ray, R. Krishna, CARGO: AI-guided dependency analysis for migrating monolithic applications to microservices architecture, in: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2023.

[24] O. Al-Debagy, P. Martinek, Dependencies-based microservices decomposition method, Int. J. Comput. Appl. 44 (2021) 814–821.

[25] M. Brito, J. Cunha, J. Saraiva, Identification of microservices from monolithic applications through topic modelling, in: Proceedings of the 36th Annual ACM Symposium on Applied Computing, 2021, pp. 1409–1418.

[26] K. Sellami, M.A. Saied, A. Ouni, A hierarchical DBSCAN method for extracting microservices from monolithic applications, in: Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering, 2022, pp. 201–210.

[27] X. Sun, S. Boranbaev, S. Han, H. Wang, D. Yu, Expert system for automatic microservices identification using API similarity graph, Expert Syst. (2022).

[28] L. Baresi, M. Garriga, A. De Renzis, Microservices identification through interface analysis, in: Service-Oriented and Cloud Computing, 2017, pp. 19–33.

[29] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, Q. Zheng, Service candidate identification from monolithic systems based on execution traces, IEEE Trans. Softw. Eng. 47 (2021) 987–1007.

[30] W. Jin, T. Liu, Q. Zheng, D. Cui, Y. Cai, Functionality-oriented microservice extraction based on execution trace clustering, in: 2018 IEEE International Conference on Web Services, ICWS, 2018, pp. 211–218.

[31] M. Gysel, L. Kölbener, W. Giersche, O. Zimmermann, Service cutter: A systematic approach to service decomposition, in: Service-Oriented and Cloud Computing, 2016, pp. 185–200.

[32] S. Eski, 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, 2018.

[33] J. Löhnertz, A. Oprescu, Steinmetz: Toward automatic decomposition of monolithic software into microservices, in: Seminar on Advanced Techniques and Tools for Software Evolution, 2020.

[34] F.H. Vera-Rivera, E. Puerto, H. Astudillo, C.M. Gaona, Microservices backlog—A genetic programming technique for identification and evaluation of microservices from user stories, IEEE Access (2021) 117178–117203.

[35] A.K. Kalia, J. Xiao, R. Krishna, S. Sinha, M. Vukovic, D. Banerjee, Mono2Micro: A practical and effective tool for decomposing monolithic Java applications to microservices, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 1214–1224.

[36] W.K.G. Assunção, T.E. Colanzi, L. Carvalho, A. Garcia, J.A. Pereira, M.J. de Lima, C. Lucena, Analysis of a many-objective optimization approach for identifying microservices from legacy systems, Empir. Softw. Engg. (2022).

[37] L. Carvalho, A. Garcia, T.E. Colanzi, W.K.G. Assunção, J.A. Pereira, B. Fonseca, M. Ribeiro, M.J. de Lima, C. Lucena, On the performance and adoption of search-based microservice identification with toMicroservices, in: 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME, 2020, pp. 569–580.

[38] M. Dehghani, S. Kolahdouz-Rahimi, M. Tisi, D. Tamzalit, Facilitating the migration to the microservice architecture via model-driven reverse engineering and reinforcement learning, Softw. Syst. Model. (2022) 1115–1133.

[39] B. Kitchenham, S. Charters, Guidelines for Performing Systematic Literature Reviews in Software Engineering, Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007.

[40] B. Kitchenham, P. Brereton, A systematic review of systematic review process research in software engineering, Inf. Softw. Technol. 55 (2013).

[41] M. Gusenbauer, N.R. Haddaway, Which academic search systems are suitable for systematic reviews or meta-analyses? Evaluating retrieval qualities of Google Scholar, PubMed, and 26 other resources, Res. Synth. Methods 11 (2) (2020).

[42] R.C. Nickerson, U. Varshney, J. Muntermann, A method for taxonomy development and its application in information systems, Eur. J. Inf. Syst. 22 (3) (2013) 336–359.

[43] T. Matias, F.F. Correia, J. Fritzsch, J. Bogner, H.S. Ferreira, A. Restivo, Determining microservice boundaries: A case study using static and dynamic software analysis, in: Software Architecture, 2020, pp. 315–332.

[44] M.I. Joselyne, G. Bajpai, F. Nzanywayingoma, A systematic framework of application modernization to microservice based architecture, in: 2021 International Conference on Engineering and Emerging Technologies, ICEET, 2021, pp. 1–6.

[45] M. Weske, Business Process Management: Concepts, Languages, Architectures, Springer, 2012.

[46] S. Li, H. Zhang, Z. Jia, Z. Li, C. Zhang, J. Li, Q. Gao, J. Ge, Z. Shan, A dataflow-driven approach to identifying microservices from monolithic applications, J. Syst. Softw. 157 (2019) 110380.

[47] Y. Wei, Y. Yu, M. Pan, T. Zhang, A feature table approach to decomposing monolithic applications into microservices, in: Proceedings of the 12th Asia-Pacific Symposium on Internetware, 2021, pp. 21–30.

[48] M. Daoud, A.E. Mezouari, N. Faci, D. Benslimane, Z. Maamar, A.E. Fazziki, Towards an automatic identification of microservices from business processes, in: 2020 IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE, 2020, pp. 42–47.

[49] M. Tusjunt, W. Vatanawood, Refactoring orchestrated web services into microservices using decomposition pattern, in: 2018 IEEE 4th International Conference on Computer and Communications, ICCC, 2018, pp. 609–613.

[50] S. Tyszberowicz, R. Heinrich, B. Liu, Z. Liu, Identifying microservices using functional decomposition, in: Dependable Software Engineering. Theories, Tools, and Applications, 2018, pp. 50–65.

[51] A.A.C. De Alwis, A. Barros, C. Fidge, A. Polyvyanyy, Discovering microservices in enterprise systems using a business object containment heuristic, in: On the Move To Meaningful Internet Systems. OTM 2018 Conferences, 2018, pp. 60–79.

[52] A.A.C. De Alwis, A. Barros, A. Polyvyanyy, C. Fidge, Function-splitting heuristics for discovery of microservices in enterprise systems, in: Service-Oriented Computing, 2018, pp. 37–53.

[53] A. Krause, C. Zirkelbach, W. Hasselbring, S. Lenga, D. Kröger, Microservice decomposition via static and dynamic analysis of the monolith, in: 2020 IEEE International Conference on Software Architecture Companion, ICSA-C, 2020, pp. 9–16.

[54] N. Lapuz, P. Clarke, Y. Abgaz, Digital transformation and the role of dynamic tooling in extracting microservices from existing software systems, in: Systems, Software and Services Process Improvement, 2021, pp. 301–315.

[55] Z. Ren, W. Wang, G. Wu, C. Gao, W. Chen, J. Wei, T. Huang, Migrating web applications from monolithic structure to microservices architecture, in: Proceedings of the 10th Asia-Pacific Symposium on Internetware, 2018.

[56] I. Trabelsi, M. Abdellatif, A. Abubaker, N. Moha, S. Mosser, S. Ebrahimi-Kahou, Y.-G. Guéhéneuc, From legacy to microservices: A type-based approach for microservices identification using machine learning and semantic analysis, J. Softw.: Evol. Process. (2022).

[57] P. Kherwa, P. Bansal, Topic modeling: A comprehensive review, EAI Endorsed Trans. Scalable Inf. Syst. 7 (2018) e2.

[58] I. Pigazzini, F.A. Fontana, A. Maggioni, Tool support for the migration to microservice architecture: An industrial case study, in: European Conference on Software Architecture, 2019, pp. 247–263.

[59] O. Al-Debagy, P. Martinek, A new decomposition method for designing microservices, Period. Polytech. Electr. Eng. Comput. Sci. 63 (2019) 274–281.

[60] M. Newman, M. Girvan, Finding and evaluating community structure in networks, Phys. Rev. E, Stat. Nonlinear, Soft Matter Phys. 69 (2004) 026113.

[61] N. Raghavan, R. Albert, S. Kumara, Near linear time algorithm to detect community structures in large-scale networks, Phys. Rev. E, Stat. Nonlinear, Soft Matter Phys. 76 (2007) 036106.

[62] P. Chaudhari, A.K. Thakur, R. Kumar, N. Banerjee, A. Kumar, Comparison of NSGA-III with NSGA-II for multi objective optimization of adiabatic styrene reactor, Mater. Today: Proc. 57 (2022) 1509–1514.

[63] H. Ishibuchi, R. Imada, Y. Setoguchi, Y. Nojima, Performance comparison of NSGA-II and NSGA-III on various many-objective test problems, in: 2016 IEEE Congress on Evolutionary Computation, CEC, 2016, pp. 3045–3052.

[64] P. Zaragoza, A.-D. Seriai, A. Seriai, A. Shatnawi, M. Derras, Leveraging the layered architecture for microservice recovery, in: 2022 IEEE 19th International Conference on Software Architecture, ICSA, 2022, pp. 135–145.

[65] X. Sun, S. Boranbaev, S. Han, H. Wang, D. Yu, Expert system for automatic microservices identification using API similarity graph, Expert Syst. (2022).

[66] L. Nunes, N. Santos, A. Rito Silva, From a monolith to a microservices architecture: An approach based on transactional contexts, in: Software Architecture, Springer International Publishing, 2019.

[67] K. Bozan, K. Lyytinen, G.M. Rose, How to transition incrementally to microservice architecture, Commun. ACM (2020) 79–85.

[68] H. Michael Ayas, P. Leitner, R. Hebig, The migration journey towards microservices, in: Product-Focused Software Process Improvement: 22nd International Conference, PROFES 2021, Turin, Italy, November 26, 2021, Proceedings, 2021, pp. 20–35.

[69] S.A. Maisto, B. Di Martino, S. Nacchia, From monolith to cloud architecture using semi-automated microservices modernization, in: Advances on P2P, Parallel, Grid, Cloud and Internet Computing, 2020.

[70] J. Doležal, A. Buchalcevová, Migration from monolithic to microservice architecture : research of impacts on agility, in: IDIMT, 2022.

[71] A. Furda, C. Fidge, O. Zimmermann, W. Kelly, A. Barros, Migrating enterprise legacy source code to microservices: On multitenancy, statefulness, and data consistency, IEEE Softw. 35 (2018) 63–72.

[72] D. Bajaj, A. Goel, S.C. Gupta, GreenMicro: Identifying microservices from use cases in greenfield development, IEEE Access 10 (2022) 67008–67018.

[73] D. Taibi, K. Systä, A decomposition and metric-based evaluation framework for microservices, in: Cloud Computing and Services Science, 2020, pp. 133–149.

[74] B. Liu, J. Xiong, Q. Ren, S. Tyszberowicz, Z. Yang, Log2MS: a framework for automated refactoring monolith into microservices using execution logs, in: 2022 IEEE International Conference on Web Services, ICWS, 2022, pp. 391–396.

[75] B. Mitchell, S. Mancoridis, On the automatic modularization of software systems using the Bunch tool, IEEE Trans. Softw. Eng. (2006) 193–208.

[76] P. Andritsos, V. Tzerpos, Information-theoretic software clustering, IEEE Trans. Softw. Eng. (2005) 150–165.

[77] M. Chatterjee, S.K. Das, D. Turgut, WCA: A weighted clustering algorithm for mobile ad hoc networks, Clust. Comput. (2002) 193–204.

[78] Z. Li, C. Shang, J. Wu, Y. Li, Microservice extraction based on knowledge graph from monolithic applications, Inf. Softw. Technol. 150 (2022).

[79] L. Cao, C. Zhang, Implementation of domain-oriented microservices decomposition based on node-attributed network, in: Proceedings of the 2022 11th International Conference on Software and Computer Applications, 2022, pp. 136–142.

[80] W.K.G. Assunção, T.E. Colanzi, L. Carvalho, J.A. Pereira, A. Garcia, M.J. de Lima, C. Lucena, A multi-criteria strategy for redesigning legacy features as microservices: An industrial case study, in: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER, 2021, pp. 377–387.

[81] O. Al-Debagy, A microservice decomposition method through using distributed representation of source code, Scalable Comput. Pr. Exp. (2021) 39–52.

[82] J. Kazanavičius, D. Mazeika, D. Kalibatiene, An approach to migrate a monolith database into multi-model polyglot persistence based on microservice architecture: A case study for mainframe database, Appl. Sci. 12 (2022) 6189.

[83] M. Camilli, C. Colarusso, B. Russo, E. Zimeo, Domain metric driven decomposition of data-intensive applications, in: 2020 IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW, 2020, pp. 189–196.

[84] R. Mishra, N. Jaiswal, R. Prakash, P.N. Barwal, Transition from monolithic to microservices architecture: Need and proposed pipeline, in: 2022 International Conference on Futuristic Technologies, INCOFT, 2022, pp. 1–6.

[85] F. Freitas, A. Ferreira, J. Cunha, Refactoring java monoliths into executable microservice-based applications, 2021, pp. 100–107.

[86] A.A.C. De Alwis, A. Barros, C. Fidge, A. Polyvyanyy, Availability and scalability optimized microservice discovery from enterprise systems, in: On the Move To Meaningful Internet Systems: OTM 2019 Conferences, 2019, pp. 496–514.

[87] M.H. Gomes Barbosa, P.H. M. Maia, Towards identifying microservice candidates from business rules implemented in stored procedures, in: 2020 IEEE International Conference on Software Architecture Companion, ICSA-C, 2020, pp. 41–48.

[88] S. Santos, A. Silva, Microservices identification in monolith systems: Functionality redesign complexity and evaluation of similarity measures, J. Web Eng. (2022).

[89] D. Escobar, D. Cárdenas, R. Amarillo, E. Castro, K. Garcés, C. Parra, R. Casallas, Towards the understanding and evolution of monolithic applications as microservices, in: 2016 XLII Latin American Computing Conference, CLEI, 2016, pp. 1–11.

[90] M. Cojocaru, A. Uta, A.-M. Oprescu, MicroValid: A validation framework for automatically decomposed microservices, in: 2019 IEEE International Conference on Cloud Computing Technology and Science, CloudCom, 2019, pp. 78–86.

[91] F.H. Vera-Rivera, E.G. Puerto-Cuadros, H. Astudillo, C.M. Gaona-Cuevas, Microservices backlog-a model of granularity specification and microservice identification, in: Services Computing-SCC 2020: 17th International Conference, Held As Part of the Services Conference Federation, SCF 2020, Honolulu, HI, USA, September 18–20, 2020, Proceedings, 2020, pp. 85–102.

[92] R. Yedida, R. Krishna, A. Kalia, T. Menzies, J. Xiao, M. Vukovic, Lessons learned from hyper-parameter tuning for microservice candidate identification, in: 2021 36th IEEE/ACM International Conference on Automated Software Engineering, ASE, 2021, pp. 1141–1145.

[93] A. Levcovitz, R. Terra, M. Valente, Towards a technique for extracting microservices from monolithic enterprise systems, 2016, URL https://doi.org/10.48550/arXiv.1605.03175.

[94] A. Carrasco, B.v. Bladel, S. Demeyer, Migrating towards microservices: Migration and architecture smells, in: Proceedings of the 2nd International Workshop on Refactoring, 2018, pp. 1–6.

[95] C.-Y. Li, S.-P. Ma, T.-W. Lu, Microservice migration using strangler fig pattern: A case study on the green button system, in: 2020 International Computer Symposium, ICS, 2020, pp. 519–524.

[96] M. Fowler, Strangler fig application, 2024, URL https://martinfowler.com/bliki/StranglerFigApplication.html. (Accessed 23 December 2024).
[97] N. Santos, A. Rito Silva, A complexity metric for microservices architecture migration, in: 2020 IEEE International Conference on Software Architecture, ICSA, 2020, pp. 169–178.
[98] L. Carvalho, A. Garcia, W. K. G. Assunção, R. de Mello, 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, 2019, pp. 22–29.
[99] S. Santos, A.R. Silva, Microservices identification in monolith systems: Functionality redesign complexity and evaluation of similarity measures, J. Web Eng. 21 (2022) 1543–1582.
[100] T.D. Stojanovic, S.D. Lazarevic, M. Milic, I. Antovic, Identifying microservices using structured system analysis, in: 2020 24th International Conference on Information Technology, IT, 2020, pp. 1–4.