The Journal of Systems and Software 222 (2025) 112334

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

# Multivocal study on microservice dependencies ☆

Amr S. Abdelfattah [a], Tomas Cerny [a,*], Md Showkat Hossain Chy [a], Md Arfan Uddin [a], Samantha Perry [a], Cameron Brown [a], Lauren Goodrich [a], Miguel Hurtado [a], Muhid Hassan [a], Yuanfang Cai [c], Rick Kazman [b]

[a] SIE, University of Arizona, Tucson, AZ, USA
[b] University of Hawaii, Honolulu, HI, USA
[c] Drexel University, Philadelphia, USA

## ARTICLE INFO

Dataset link: https://zenodo.org/records/14208294

*Keywords:*
Microservice dependencies
Dependency taxonomy
Software maintenance
Distributed systems
Performance
Root cause analysis
Anomaly detection

## ABSTRACT

**Background:** Understanding dependencies within microservices is essential for maintaining and evolving scalable and efficient software architectures. Dependencies influence how changes in one microservice might propagate to other microservices. With the decentralized nature of microservices, these dependencies might not be explicit to developers and lead to unique challenges in modern software development environments.
**Objective:** The objective of this study is to synthesize existing literature on microservice dependencies, identify the types of dependencies, and examine the strategies employed to manage and analyze these relationships. This effort aims to elucidate how dependencies affect microservice systems and to provide a comprehensive overview of dependency management within microservices.
**Method:** We conducted a multivocal literature review, starting with an initial dataset of 1,733 papers from academic literature (white literature). This corpus was narrowed down through a rigorous filtering process to 45 key publications that address the identification, management, and impacts of dependencies in microservices. Additionally, we incorporated 926 articles from grey literature sources such as Google, Stack Overflow, and Stack Exchange, expanding the scope beyond traditional academic research. After the filtration process, 45 articles were fully synthesized to integrate practical insights and professional experiences into our review.
**Results:** The review identifies several types of dependencies in microservice systems and synthesizes this information into a unified dependency taxonomy. This review highlights a range of approaches to dependency management, revealing a significant gap in systematic catering approaches to generate taxonomies for dependencies and the need for integrated management tools. The findings underscore the fragmented nature of existing dependency management practices and the potential for more holistic approaches.
**Conclusion:** This study provides valuable insights for researchers and practitioners, outlining effective strategies and pointing out areas needing improvement in dependency management. By offering a structured overview of the topic, the study serves as a roadmap for future research and development efforts to enhance the robustness and maintainability of microservices.

**Contents**



☆ Editor: Uwe Zdun.

* Corresponding author.
*E-mail addresses:* amrelsayed@arizona.edu (A.S. Abdelfattah), tcerny@arizona.edu (T. Cerny), chym@arizona.edu (M.S.H. Chy), arfan@arizona.edu (M.A. Uddin), samnperry@arizona.edu (S. Perry), cabrown3076@arizona.edu (C. Brown), laureng5@arizona.edu (L. Goodrich), miguelhurtado@arizona.edu (M. Hurtado), muhidhassan@arizona.edu (M. Hassan), yc349@drexel.edu (Y. Cai), kazman@hawaii.edu (R. Kazman).

https://doi.org/10.1016/j.jss.2025.112334
Received 14 June 2024; Received in revised form 23 November 2024; Accepted 7 January 2025
Available online 13 January 2025
0164-1212/© 2025 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

## 1. Introduction

In the rapidly evolving software development landscape, microservice architecture has emerged as a transformative approach, advocating for decomposing applications into smaller, loosely coupled units. Each microservice is designed to perform a distinct function, operating independently from its peers, which contrasts sharply with the tightly integrated components of traditional monolithic architectures (Hasselbring and Steinacker, 2017). This direction facilitates agile development and deployment practices and aligns with the DevOps philosophy promoting continuous integration and continuous deployment (CI/CD) practices (Newman, 2015). Microservices offer scalability and flexibility, enabling organizations to innovate and adapt to market changes more swiftly.

Despite these benefits, the shift to microservices introduces complex challenges, primarily related to managing the dependencies that inevitably arise among services. Dependencies in microservices can include direct service-to-service communication, shared databases, and even shared code libraries, each adding a layer of complexity to the architecture. The dynamic nature of these dependencies can lead to significant challenges in service maintenance, scalability, and resilience. As services proliferate, the overhead of managing these dependencies can escalate, potentially negating the benefits of the microservice architecture by introducing operational complexity and reducing system reliability (Cerny et al., 2024).

The management of these dependencies is thus a critical area of concern. It requires robust strategies to ensure that the system remains maintainable and that changes in one service do not cascade failures across others. Effective dependency management must address issues of service discovery, load balancing, fault tolerance, and transaction management, ensuring that the microservice architecture retains its intended benefits of modularity and isolation (Dragoni et al., 2016). The literature suggests various approaches to managing these dependencies, from using service meshes to implementing API gateways and employing automated orchestration tools, each contributing differently to the robustness and efficiency of the system.

This systematic literature review aims to delineate the spectrum of dependency types encountered in microservice architecture, identify the challenges they pose, and evaluate the effectiveness of various management strategies that have been proposed or adopted in practice. While literature provides theoretical rigor, systematic methodologies, and well-validated insights, it is complemented by the rapid advancements and innovative solutions emerging from industry practices. The inclusion of grey literature addresses this gap by offering practical, real-world perspectives that reflect the evolving challenges and solutions

directly observed by practitioners. Grey literature sources, such as technical blogs, discussion forums, and white papers, often capture cutting-edge innovations, undocumented industry trends, and problem-solving approaches that are not yet formalized in academic research. This combination of white and grey literature ensures a comprehensive review, bridging the gap between theory and practice. By synthesizing these complementary perspectives, this study provides strong understanding of dependencies and offers actionable insights into how they can be effectively managed to optimize the development process and operational performance of microservice systems.

### 1.1. Objective and research questions (RQs)

The primary objective of this systematic literature review is to identify and categorize the various dependency types in microservices-based systems, explore their management approaches, and address the challenges they present along with proposed solutions. To achieve this objective, the study is guided by the following research questions:

- **RQ1** — What types of dependencies are prevalent in microservice architecture, and how do they impact system design and functionality?
- **RQ2** — What techniques and tools are available to detect and manage these dependencies?
- **RQ2.1** — Which benchmarks have been used in literature to test and demonstrate dependencies?
- **RQ3** — Does the existing literature consider dependencies in addressing software architecture quality attributes?

### 1.2. Contributions

This research illuminates the complex landscape of dependency management in microservices-based systems. Through a meticulous examination of the existing literature, this paper highlights critical insights into the types of dependencies that affect microservices, the strategies employed to manage these dependencies, and the tools and techniques that facilitate their detection and management. By synthesizing this knowledge, the paper provides a robust framework for both practitioners and researchers, guiding the effective navigation of dependency challenges and pinpointing areas ripe for further exploration and technological advancement. The specific contributions of this systematic literature review are outlined as follows:

- **Development of Microservice Dependency Taxonomy:** Establishing clear taxonomy for classifying microservice dependencies. This involves identifying and categorizing the various types of dependencies in microservices and elucidating how they influence system design, quality, and operational efficiency.
- **Evaluation of Management Strategies:** Comprehensive assessment of current methodologies and tools available for dependency management, highlighting their effectiveness, limitations, and areas for improvement.
- **Linking Dependencies with the Impact on Quality:** Exploration of the causal relationships between dependency types and the operational challenges they present. It provides a deeper understanding of the systemic impacts of dependencies in microservices (i.e., causes and symptoms).
- **Exploration of Microservices Benchmarks:** Identified and analyzed microservices-based benchmarks used for dependencies in the literature, providing valuable references for future research.

These contributions are intended to advance the field of microservice architecture by offering a structured approach to dependency management, fostering enhanced system maintainability, stability and scalability. Additionally, by identifying gaps in the current literature, this review sets the stage for future research endeavors, aiming to refine dependency detection techniques and develop more robust management solutions.

### 1.3. Organization of the paper

The rest of the paper is organized as follows: Section 2 discusses background and related work, which provides the basis for this review. Section 3 overviews the research methods conducted in performing the literature review. Section 4 details the review execution. Section 5 outlines found results and answers to research questions. Section 6 discusses the implications of the results. Section 7 notes potential threats to validity. Section 8 concludes the findings and proposes areas of planned future work.

## 2. Background and related work

Microservice architecture have become a fundamental paradigm in modern software engineering, enabling systems to achieve scalability, flexibility, and maintainability. However, their decentralized nature presents substantial challenges in dependency management, which, if neglected, can undermine system performance, reliability, and maintainability. While systematic literature reviews (SLRs) and systematic mapping studies (SMSs) have extensively examined aspects such as scalability, observability, and maintainability, dependency management remains an underexplored domain. Existing studies often focus on theoretical frameworks or isolated case studies, offering limited empirical validation for strategies to detect, categorize, and mitigate dependencies. To address these gaps, this review synthesizes insights from both white and grey literature, providing a multivocal perspective that bridges the gap between theoretical approaches and practical applications.

Ding and Zhang (2022) emphasized the critical need for precise definitions and effective strategies to manage Microservice Architecture Smells (MAS), particularly during transitions from monolithic systems to microservices. They identified significant gaps in current approaches, particularly the lack of automated tools for dependency detection and management, which often result in cascading failures that hinder system scalability and maintainability. Haindl et al. (2024) highlighted challenges in addressing inter-service dependencies, stressing the need for improved frameworks that account for a broader range of dependency types. Their findings reveal that existing taxonomies frequently fail to encompass the interconnected nature of dependencies in microservices, leaving critical gaps in comprehensive management strategies. Torkura et al. (2018) introduced CAVAS, a framework designed to automate assessments of service interactions, demonstrating the potential of tools that integrate dependency management into the development lifecycle. Furthermore, Waseem et al. (2022) proposed decision models to guide the decomposition of monolithic systems into microservices, with a particular emphasis on optimizing dependency management to improve system scalability and maintainability. Collectively, these studies underscore the necessity of advancing tools and methodologies to enhance the detection, categorization, and management of dependencies in microservice architectures.

Despite considerable advancements, existing taxonomies of microservice dependencies continue to exhibit critical limitations. Haindl et al. (2024) emphasized the need to address overlooked aspects of microservice interactions, particularly dependencies arising from dynamic runtime behaviors such as scaling, load balancing, and reconfiguration. These runtime dynamics often evolve unpredictably, creating challenges in maintaining consistent dependency management. Furthermore, Ding and Zhang (2022) and Waseem et al. (2022) highlighted gaps in existing dependency classifications, particularly in their inability to capture the complex interconnections between microservices in large-scale systems. Building upon these foundational insights, this study proposes an extended taxonomy that incorporates the nuanced interdependencies observed in evolving microservice architectures. Tables 1 and 2 summarize research questions and methodologies from related SLRs and SMSs, providing a structured foundation for this work.

**Table 1**
Research questions in related SLR studies — Part (1/2).

| Title | Paper |
|---|---|
| A Systematic Literature Review of Inter-Service Security Threats and Mitigation Strategies in Microservice Architectures | Haindl et al. (2024) |
| RQ1: What characterizes the research, contributions, publication channels, venues, and publication trends in the field of inter-service security of microservices?<br>RQ2: Which inter-service security threats in MSA are discussed in the studies and how can they be categorized?<br>RQ3: Which mitigation strategies for inter-service security threats in MSA are discussed in the studies and how can they be categorized? | |
| Decomposition of Monolith Applications Into Microservice Architecture: A Systematic Review | Abgaz et al. (2023) |
| RQ1: What are the primary phases of monolith-to-microservices decomposition and the major constituent elements of those phases?<br>RQ2: What are the existing approaches, tools, and methods observed in the decomposition of monolith applications into microservices?<br>RQ3: What are the metrics, datasets, and benchmarks used for evaluating and validating monolith decomposition into microservices?<br>RQ4: What research gaps can be identified in the current literature? | |
| Microservices identification strategies: A review focused on Model-Driven Engineering and Domain Driven Design approaches | Schmidt and Thiry (2020) |
| RQ1: Which strategies, techniques or methods have been adopted for domain decomposition into microservices and how are they applied to stablish its optimal granularity?<br>RQ2: Which roles are involved in the proposed microservices identification process and what are their main responsibilities?<br>RQ3: Do the selected proposals apply Model-Driven Engineering (MDE) and which related elements have been explored?<br>RQ4: Do the selected studies cover Domain-Driven Design (DDD) at any level and in what way are its principles, practices and patterns applied? | |
| On Microservice Analysis and Architecture Evolution: A Systematic Mapping Study | Bushong et al. (2021) |
| RQ1: What methods and techniques are used in microservice analysis?<br>RQ2: What are the problems or opportunities that are addressed using microservice analysis techniques?<br>RQ3: Does microservice analysis overlap with other areas of software analysis, or are new methods or paradigms needed?<br>RQ4: What potential future research directions are open in the area of microservice analysis? | |
| Toward the Observability of Cloud-Native Applications: The Overview of the State-of-the-Art | Kosińska et al. (2023) |
| RQ1: What provides the motivations for equipping Cloud-native applications with observability capabilities?<br>RQ2: Which research areas are addressed in the related literature?<br>RQ3: How are observability approaches implemented?<br>RQ4: What are the observability research? | |

### 2.1. Dependency management

Recent advancements in dependency management have introduced tools and methodologies designed to address critical challenges in microservice architectures. Parker et al. (2023) emphasize the necessity of tools capable of dynamically detecting and visualizing dependencies in real-time. Such capabilities allow for improved system responsiveness and operational efficiency. However, existing tools are often limited in their ability to integrate detection and visualization, reducing their effectiveness in proactively managing dependencies. Similarly, Bushong et al. (2021) highlight the importance of adaptability in dependency management tools, noting that many current solutions are static and fail to evolve alongside the architectures they monitor. This limitation leads to potential oversights, which can adversely affect system stability and performance.

Beyond dynamic detection, methodologies such as Model-Driven Engineering (MDE) and Domain-Driven Design (DDD) have been explored for dependency management. Schmidt and Thiry (2020) identify the potential of these approaches but highlight their inconsistent application across development stages, which diminishes their overall effectiveness. In a complementary vein, Kosińska et al. (2023) advocate for the integration of observability into dependency management, emphasizing the need for tools that offer real-time insights into operational dependencies. Furthermore, Mparmpoutis and Kakarontzas (2022) stress the importance of developing automated tools capable of leveraging legacy data to streamline microservice extraction, particularly during the transition from monolithic systems. Despite the promise of these tools and methodologies, a lack of standardization often leads to varied implementations, hindering their ability to fully address dependency-related challenges in a consistent and effective manner.

### 2.2. Dependency categorization

Abgaz et al. (2023) stress the importance of categorizing dependencies accurately. Their research reveals a gap in detailed management strategies for different dependencies, essential for targeted and effective management. This lack of specificity can lead to general solutions that may not address the unique challenges of different dependency types, suggesting the need for more refined categorization methods.

Kosińska et al. (2023) discuss the need for advanced tools that integrate observability with proactive dependency management. Their research identifies a significant gap in existing tools that can effectively manage and monitor operational dependencies in real-time. This deficiency can lead to delayed responses to issues, impacting the performance and reliability of cloud-native applications and highlighting the need for more integrated and responsive observability solutions.

### 2.3. Dependencies and their impact on quality

Schmidt and Thiry (2020) highlight the issues of high coupling and low cohesion. The study notes that not enough strategies directly tackle these fundamental causes of dependency problems, which can severely impact system scalability and maintainability. This gap in specific strategies indicates the need for focused research and development efforts to devise methods that explicitly address these critical issues.

Valdivia et al. (2020) analyze the impact of architectural patterns on dependency management. They highlight a lack of comprehensive guidelines for effectively selecting and implementing patterns that minimize dependency risks. This deficiency in guidelines can lead to the selection of patterns that exacerbate dependency issues, complicating the architecture unnecessarily and leading to increased maintenance challenges.

### 2.4. Related work implications

Söylemez et al. (2024) identify the need for adaptable and dynamic tools for managing transformations in microservice architectures. The studies point out that current tools lack the flexibility to cater to diverse business and technical requirements, indicating a need for solutions that are not only robust but also adaptable to various implementation contexts. This highlights an ongoing demand for development in this area to provide more comprehensive and versatile tools in microservices.

**Table 2**
Research questions in related SLR studies — Part (2/2).

| Title | Paper |
|---|---|
| Patterns Related to Microservice Architecture: a Multivocal Literature Review<br>RQ1. What patterns are identified in microservices systems?<br>RQ2. Which quality attributes are related to the patterns used in microservices?<br>RQ3. What metrics are used to quantify quality attributes related to a microservices pattern? | Valdivia et al. (2020) |
| Microservice reference architecture design: A multi-case study<br>RQ1: How effective is the adopted microservices reference architecture?<br>RQ2: How practical is the adopted microservices reference architecture? | Söylemez et al. (2024) |
| How Can We Cope with the Impact of Microservice Architecture Smells?<br>RQ1: How to define and classify the AS exists in Microservice architecture?<br>RQ2: What issues exist in the migration process from Monolith to Microservice caused by architecture smells? | Ding and Zhang (2022) |
| Visualizing Anti-Patterns in Microservices at Runtime: A Systematic Mapping Study<br>RQ1: What methods of detecting patterns in SOA/ Microservices exist? Can these methods be adapted to detect anti-patterns too?<br>RQ2: How can dynamic analysis be used to analyze anti-patterns in a system built using microservices?<br>RQ3: Which anti-patterns can be visualized in call graphs?<br>RQ4: How can visualizing the architecture of a microservice aid in detecting or preventing anti-patterns within that system?<br>RQ5: Which tools exist for visualizing anti-patterns through the dynamic perspective? | Parker et al. (2023) |
| Using Database Schemas of Legacy Applications for Microservices Identification: A Mapping Study<br>RQ1: During the process of re-architecting a legacy system, are data-driven artifacts like database schema and state of data used for identifying potential microservices?<br>RQ1.1: Can the process of identifying services/microservices from database artifacts be in a complete or partial way automated?<br>RQ2: How data-driven artifacts, like the database, are used in the process of a software migration to services/microservices based architecture? | Mparmpoutis and Kakarontzas (2022) |

These gaps and limitations identified across the studies underline the necessity for ongoing innovation and refinement in tools, techniques, and methodologies within the microservices domain. Addressing these challenges will enhance microservice architectures' detection, management, and overall robustness, contributing to more stable and efficient software systems.

## 3. Methods

This section presents a multivocal study designed to explore dependency management in microservice architectures. The study aims to identify and categorize the types of dependencies that exist within these architectures (RQ1), and to examine the techniques and tools available for detecting and measuring these dependencies (RQ2), while it includes a validation assessment of the identified metrics and tools (RQ2.1). Additionally, the study explores the use of dependencies in addressing architecture quality (RQ3).

### 3.1. Search query

To ensure comprehensive coverage of the literature on microservice dependencies, both in white and grey literature, we structured our search around specific, targeted queries. This allowed us to filter and analyze relevant studies efficiently.

Our search queries were crafted to capture the most pertinent studies. We adopted the PICO framework (Population, Intervention, Comparison, Outcome) to structure our queries, ensuring each component was specifically tailored to our research needs (Kitchenham and Charters, 2007). Moreover, these queries were designed to capture a wide array of publications, spanning both theoretical studies and practical applications.

The construction of the search query followed a refinement process. We iteratively reviewed the results until we achieved the most accurate and representative search query for our study. The multiple search queries and the number of papers retrieved from each indexed source are presented in Tables 3–6. The final queries listed in these tables were ultimately employed in our study. These queries were designed to refine the results and focus on the most relevant papers, incorporating insights from the preliminary queries.

#### 3.1.1. White literature search

For academic databases such as Scopus, IEEE Xplore, and ACM Digital Library, we crafted queries tailored to their indexing peculiarities and search algorithms. Our search strings were composed to include a combination of terms relevant to microservices—like "`Microservice`", "`Service-Oriented`", and "`Cloud-native`"—alongside dependency-related keywords such as "`Dependency`", "`Couple`", and "`Metrics`". This dual focus allowed us to capture both broad discussions on microservice architectures and more specific studies on their inter-component dependencies.

In Scopus, our queries were executed using the TITLE-ABS-KEY fields to ensure the retrieved papers prominently discussed the key terms within their titles, abstracts, and keywords. Similar strategies were employed in IEEE Xplore, but adapted to the platform's search capability. Such that, we used the "All Metadata" search capability, which broadens the search scope to include title, abstract, keywords, and all other text fields available in the metadata, thereby capturing a wider array of relevant studies. Our search strings included variations of "`microservice`" combined with key terms like "`dependency`", "`couple`", "`metrics`", and "`coupling`".

The ACM Digital Library required a more refined search strategy, focusing on both title and abstract fields to ensure that only the most pertinent studies were included. This approach was particularly important in a library as diverse as the ACM's, where the potential for retrieving off-topic papers is high due to the broad range of computing topics covered. We have utilized the final search query from Table 5 to extract the related studies.

Moreover, Google Scholar was used as an additional data source, though it includes some less formal publications and has search limitations restricted to titles or full text. Initially, we conducted a title-based search using the *allintitle:* directive with our search terms of "`Dependency`", "`Microservice`", and related keywords, which yielded a smaller and more variable number of results, as shown in Table 6. To expand the scope and align it with other data sources targeting titles, abstracts, and keywords, we performed full-text searches, resulting in two sets: Set #9 (16,200 papers) and Set #10 (20,300 papers). From each set, we accessed and downloaded up to 1000 papers, the maximum allowable by Google Scholar's system, and manually queried their titles, abstracts, and keywords to verify relevance. Although this limitation restricted the breadth of literature directly reviewed from Google Scholar, the platform serves as a collective indexer, including most papers already included from other sources.

**Table 3**
SCOPUS search query generation process.

| Keyword | Search strings | # of papers |
|---|---|---|
| Set #1 (initial) | TITLE-ABS-KEY (“Microservice” OR “Micro service” OR “Micro services” OR “Micro-service” OR “Micro-services” OR “Cloud-native” OR “Service-Oriented” OR “Decentralized”) AND TITLE-ABS-KEY (“Dependency”) | 1704 |
| Set #2 | TITLE-ABS-KEY (“Microservice” OR “Micro service” OR “Micro services” OR “Micro-service” OR “Micro-services” OR “Cloud-native” OR “Service-Oriented”) AND TITLE-ABS-KEY (“Dependency” OR “couple”) | 979 |
| Set #3 | TITLE-ABS-KEY (“Microservice” OR “Micro service” OR “Micro services” OR “Micro-service” OR “Micro-services” OR “Cloud-native”) AND TITLE-ABS-KEY (“Dependency” OR “Couple” OR “Metrics”) | 718 |
| Set #4 | TITLE-ABS-KEY (“Microservice” OR “Micro service” OR “Micro services” OR “Micro-service” OR “Micro-services” OR “Cloud-native”) AND TITLE-ABS-KEY (“Dependency”) | 324 |
| Set #5 | TITLE-ABS-KEY (“Microservice” OR “Micro service” OR “Micro services” OR “Micro-service” OR “Micro-services” OR “Cloud-native” OR “Service-Oriented”) AND TITLE-ABS-KEY (“Dependency” OR “coupling”) AND ( LIMIT-TO (SUBJAREA, “COMP”)) | 1389 |
| Set #6 | TITLE-ABS-KEY (“Microservice” OR “Micro service” OR “Micro services” OR “Micro-service” OR “Micro-services” OR “Cloud-native” OR “Service-Oriented”) AND TITLE-ABS-KEY (“Dependency” OR “coupling”) AND ( LIMIT-TO ( SUBJAREA, “COMP” )) | 1389 |
| Set #7 (final) | (TITLE-ABS-KEY (“Microservice” OR “Micro service” OR “Micro services” OR “Micro-service” OR “Micro-services” OR “Cloud-native”) AND TITLE-ABS-KEY (“Dependency” OR “coupling”)) AND (LIMIT-TO (SUBJAREA, “COMP”)) | 465 |

**Table 4**
IEEE search query generation process.

| Keyword | Search strings | # of papers |
|---|---|---|
| Set #1 (initial) | (“All Metadata”:“Microservice” OR “All Metadata”:“Micro service” OR “All Metadata”:“Micro services” OR “All Metadata”:“Micro-service” OR “All Metadata”:“Micro-services” OR “All Metadata”:“Cloud-native” OR “All Metadata”:“Service-Oriented” OR “All Metadata”:“Decentralized”) AND (“All Metadata”:“Dependency”) | 639 |
| Set #2 | (“All Metadata”:“Microservice” OR “All Metadata”:“Micro service” OR “All Metadata”:“Micro services” OR “All Metadata”:“Micro-service” OR “All Metadata”:“Micro-services” OR “All Metadata”:“Cloud-native” OR “All Metadata”:“Service-Oriented”) AND (“All Metadata”:“Dependency” OR “All Metadata”:“Couple”) | 447 |
| Set #3 | (“All Metadata”:“Microservice” OR “All Metadata”:“Micro service” OR “All Metadata”:“Micro services” OR “All Metadata”:“Micro-service” OR “All Metadata”:“Micro-services” OR “All Metadata”:“Cloud-native”) AND (“All Metadata”:“Dependency” OR “All Metadata”:“Couple” OR “All Metadata”:“Metrics”) | 404 |
| Set #4 | (“All Metadata”:“Microservice” OR “All Metadata”:“Micro service” OR “All Metadata”:“Micro services” OR “All Metadata”:“Micro-service” OR “All Metadata”:“Micro-services” OR “All Metadata”:“Cloud-native”) AND (“All Metadata”:“Dependency”) | 124 |
| Set #5 | (“All Metadata”:“Microservice” OR “All Metadata”:“Micro service” OR “All Metadata”:“Micro services” OR “All Metadata”:“Micro-service” OR “All Metadata”:“Micro-services” OR “All Metadata”:“Cloud-native” OR “All Metadata”:“Service-Oriented”) AND (“All Metadata”:“Dependency” OR “All Metadata”:“Coupling”) | 1223 |
| Set #6 (final) | ((“All Metadata”:“Microservice” OR “All Metadata”:“Micro service” OR “All Metadata”:“Micro services” OR “All Metadata”:“Micro-service” OR “All Metadata”:“Micro-services” OR “All Metadata”:“Cloud-native”) AND (“All Metadata”:“Dependency” OR “All Metadata”:“Coupling”)) | 275 |

#### 3.1.2. Grey literature search

Incorporating grey literature into our study was essential to capture non-academic insights and the latest industry practices. This literature, which includes technical reports, white papers, and blog posts, often precedes formal academic publications and reflects cutting-edge developments and practical experiences in the implementation of microservice architectures.

The search queries for grey literature were designed to cover a wide array of sources and to ensure the relevance of the results. We employed targeted keywords and specific operators to focus on content related to microservice dependencies, aligning with the study’s focus. For example, our queries included combinations of keywords such as “`Microservices`”, “`Microservice`”, and “`Cloud-native`”, alongside terms directly related to the study’s focus like “`Dependency`”, “`Couple`”, and “`Dependencies`”.

Platforms such as Google, Stack Overflow, and Stack Exchange were chosen for their rich repositories of ongoing discussions, code snippets, and technical problem-solving dialogues. The designed search queries are listed in Table 7. These employed search strings aimed to intersect the technical jargon frequently used in software development communities with the more formal terminology of academic research. This approach was intended to bridge the gap between theoretical research and practical application, providing a comprehensive view of the challenges and solutions found in real-world settings.

By navigating these platforms with targeted search strings, we were able to aggregate a substantial amount of grey literature, ranging from informal discussions to more structured technical reports. This method ensured that our research encompassed a holistic view of microservice dependencies, capturing both the academic discourse and the pragmatic dialogue occurring within the tech industry.

### 3.2. Data sources

Our research utilized a combination of well-established academic databases and practical sources to ensure a thorough exploration of the topic of microservice dependencies. For peer-reviewed scholarly content, including articles and conference proceedings, we primarily accessed the *IEEE Xplore Digital Library,*[1] *Scopus,*[2] and the *ACM Digital Library.*[3] These platforms are widely recognized for their extensive collections of research documents in the fields of computing and technology.

In addition to these academic databases, we expanded our search to include grey literature, which helped in identifying theses, dissertations, and less formal yet insightful publications such as technical blogs and white papers. We incorporated content from community-driven platforms like *Stack Overflow*[4] and *Stack Exchange,*[5] which offer a plethora of user-generated content on software development challenges and solutions.

[1] https://ieeexplore.ieee.org, accessed on January 30th, 2024.
[2] https://www.scopus.com, accessed on January 30th, 2024.
[3] https://dl.acm.org, accessed on January 30th, 2024.
[4] https://stackoverflow.com, accessed on January 30th, 2024.
[5] https://stackexchange.com, accessed on January 30th, 2024.

**Table 5**
ACM search query generation process.

| Keyword | Search strings | # of papers |
|---|---|---|
| Set #1 (initial) | Title:(“Microservice” OR “Micro service” OR “Micro services” OR “Micro-service” OR “Micro-services” OR “Cloud-native” OR “service-oriented” OR “decentralized” OR “distributed”) AND Title:(Dependency OR Couple OR Metrics) AND Abstract:(“Microservice” OR “Micro service” OR “Micro services” OR “Micro-service” OR “Micro-services” AND “Dependency*”) | 410 |
| Set #2 | Title:(“Microservice” OR “Micro service” OR “Micro services” OR “Micro-service” OR “Micro-services” OR “Cloud-native” OR “service-oriented” OR “decentralized” OR “distributed”) AND Title:(Dependency OR Couple OR Metrics) OR Abstract:(“Microservice” OR “Micro service” OR “Micro services” OR “Micro-service” OR “Micro-services”) AND (“Dependency*”) | 203 |
| Set #3 | Title:(“Microservice” OR “Micro service” OR “Micro services” OR “Micro-service” OR “Micro-services” OR “Cloud-native” OR “service-oriented”) AND Title:(“Dependency”) OR Abstract:(“Microservice” OR “Micro service” OR “Micro services” OR “Micro-service” OR “Micro-services” ) AND (“Dependency*”) | 151 |
| Set #4 | Title:(“Microservice” OR “Micro service” OR “Micro services” OR “Micro-service” OR “Micro-services” OR “Cloud-native” OR “service-oriented” OR “decentralized” OR “distributed”) AND Title:(“Dependency” OR “Couple” OR “Metrics”) AND Keywords:(“Microservice” OR “Micro service” OR “Micro services” OR “Micro-service” OR “Micro-services” AND “Dependency*”) | 59 |
| Set #5 | Abstract: (“microservice” OR “micro service” OR “micro services” OR “micro-service” OR “micro-services” OR “cloud-native” OR “service-oriented”) AND (Abstract: “dependency” OR “coupling” OR “metrics”) OR Title: (“microservice” OR “micro service” OR “micro services” OR “micro-service” OR “micro-services” OR “cloud-native” OR “service-oriented”) AND (Title: “dependency” OR “coupling” OR “metrics”) OR Keywords: (“microservice” OR “micro service” OR “micro services” OR “micro-service” OR “micro-services” OR “cloud-native” OR “service-oriented”) AND (Keywords: “dependency” OR “coupling” OR “metrics”) | 191 |
| Set #6 (final) | Abstract: (“microservice” OR “micro service” OR “micro services” OR “micro-service” OR “micro-services” OR “cloud-native”) AND (Abstract: “dependency” OR “coupling” OR “metrics”) OR Title: (“microservice” OR “micro service” OR “micro services” OR “micro-service” OR “micro-services” OR “cloud-native”) AND (Title: “dependency” OR “coupling” OR “metrics”) OR Keywords: (“microservice” OR “micro service” OR “micro services” OR “micro-service” OR “micro-services” OR “cloud-native”) AND (Keywords: “dependency” OR “coupling” OR “metrics”) | 84 |

**Table 6**
Google scholar search query generation process.

| Keyword | Search strings | # of papers |
|---|---|---|
| Set #1 (initial) | allintitle: Dependency Microservice OR “Micro service” OR “Micro services” OR “Micro service” OR “Micro services” OR “Cloud native” OR “Service Oriented” OR Decentralized OR Distributed | 199 |
| Set #2 | allintitle: Dependency Couple OR Microservice OR “Micro service” OR “Micro services” OR “Micro service” OR “Micro services” OR “Cloud native” OR “Service Oriented” OR Decentralized OR Distributed | 230 |
| Set #3 | allintitle: Dependency Couple OR Metrics OR Microservice OR “Micro service” OR “Micro services” OR “Micro service” OR “Micro services” OR “Cloud native” OR “Service Oriented” OR Decentralized OR Distributed | 274 |
| Set #4 | allintitle: Dependency Microservice OR “Micro service” OR “Micro services” OR “Micro service” OR “Micro services” OR “Cloud native” | 21 |
| Set #5 | allintitle: Dependency Microservice OR “Micro service” OR “Micro services” OR “Micro service” OR “Micro services” OR “Cloud native” OR “Service Oriented” | 40 |
| Set #6 | allintitle: Coupling Microservice OR “Micro service” OR “Micro services” OR “Micro service” OR “Micro services” OR “Cloud native” OR “Service Oriented” | 44 |
| Set #7 | allintitle: Dependency Microservice OR “Micro service” OR “Micro services” OR “Micro service” OR “Micro services” OR “Cloud native” | 21 |
| Set #8 | allintitle: Coupling Microservice OR “Micro service” OR “Micro services” OR “Micro service” OR “Micro services” OR “Cloud native” | 11 |
| Set #9 (final 1) | Dependency Microservice OR “Micro service” OR “Micro services” OR “Micro service” OR “Micro services” OR “Cloud native” | 16 200 |
| Set #10 (final 2) | Coupling Microservice OR “Micro service” OR “Micro services” OR “Micro service” OR “Micro services” OR “Cloud native” | 20 300 |

**Table 7**
Grey literature search results.

| Source | Search strings | Results |
|---|---|---|
| Google | (“Microservices” OR “Microservice” OR “Micro services” OR “Micro service” OR “Micro-service” OR “Cloud-native”) AND (“Dependency” OR “Couple” OR “Dependencies”) | 12,100,000 |
| Stack Overflow | [microservice] or [microservices] or [microservice] or [microservices] AND (“Dependency” or “dependency” or “Dependencies” or “dependencies”) | 500 |
| Stack Exchange | [microservice] or [microservices] or [microservice] or Dependency or dependencies | 186 |
| Stack Exchange | [microservice] or [microservices] or [microservice] or [microservices] AND (“Dependency” or “dependency” or “Dependencies” or “dependencies”) | 30 |

**Table 8**
Inclusion and exclusion criteria.

| Criteria | Description |
|---|---|
| Inclusion | |
| I1 | Studies or works that include applied techniques or metrics for analyzing dependencies in microservice architectures. |
| I2 | Research supporting or establishing a categorization methodology for microservices dependencies. |
| I3 | Studies or works that identify symptoms or quality impact associated with microservices dependencies. |
| I4 | Research including benchmarks used for evaluating dependency metrics or techniques. |
| I5 | Systematic reviews or comprehensive studies focusing on microservices dependencies. |
| Exclusion | |
| E1 | Studies not written in English. |
| E2 | Duplicated studies or those where an extension has been included in the analysis. |
| E3 | Papers that are off-topic and do not directly address microservices or their dependencies. |
| E4 | Studies not accessible through our institutional access, limiting the ability to perform a thorough review. |
| E5 | Research not specifically within the scope of microservices, such as those focusing solely on broader software engineering or IT infrastructure topics without direct relevance to microservices. |
| E6 | Vision and position papers that do not provide empirical evidence or detailed methodological insights. |
| E7 | Conference proceedings that do not offer in-depth analysis or findings relevant to the current study. |

### 3.3. Inclusion and exclusion criteria

To ensure a focused and relevant collection of literature, we established specific criteria for inclusion and exclusion. These criteria were designed to refine the search results to studies that are directly applicable to our research objectives concerning microservice dependencies. We listed our different criteria in Table 8.

This structured approach helps filter the extensive literature to focus exclusively on significant studies that contribute directly to understanding and addressing the complexities associated with dependencies in microservice architectures.

### 3.4. Quality assessment

For a robust review, each selected study underwent a systematic quality assessment to determine its relevance and reliability concerning microservice dependencies. This assessment was guided by specific criteria developed to reflect the key focus areas of our research. The evaluation process helped in ensuring the integrity and applicability of the findings to the domain of microservices.

#### 3.4.1. Quality assessment for white literature (AL-QA)

The academic articles possess specific attributes that can be used to assess their quality. The quality criteria applied to these articles are as follows:

- **AL-QA1 Objective:** Does the paper have a clear objective?
- **AL-QA2 Methodology:** Does the paper follow a specific systematic methodology?
- **AL-QA3 Microservice Focus:** Is the microservice architecture the main focus of the study?
- **AL-QA4 Dependency Focus:** Is the study primarily focused on the dependencies within microservices?
- **AL-QA5 Industry Involvement:** Does the study involve industrial parties or practitioners?
- **AL-QA6 Rigorous Analysis:** Was the data analysis conducted with sufficient rigor?
- **AL-QA7 Findings Clarity:** Is there a clear statement of findings or contributions?
- **AL-QA8 Conclusion Alignment:** Does the paper provide a clear conclusion that aligns with its objectives and findings?

Responses were assigned as follows: Y (Yes) if fully met, P (Partly) if partially met, and N (No) if not met. Each criterion was valued at Y = 1,P = 0.5, and N = 0. This scoring facilitated a quantitative assessment of each paper's quality:

- **Objective Clarity (AL-QA1):** Y (yes) if the paper's objective is clearly stated in the abstract or introduction and is closely followed throughout the research; P (partly) if the objective is implied or can be inferred from the content; N (no) if the objective is ambiguous or not identifiable.
- **Methodological Rigor (AL-QA2):** Y (yes) if the paper follows a specific, systematic methodology detailed in the text; P (partly) if the methodology is mentioned but not detailed; N (no) if there is no clear methodology described.
- **Microservice Focus (AL-QA3):** Y (yes) if microservices are the central theme of the study; P (partly) if microservices are a secondary focus alongside other architectures; N (no) if microservices are not a significant focus of the study.
- **Dependency Focus (AL-QA4):** Y (yes) if the study primarily explores dependencies within microservices; P (partly) if dependencies are discussed but not as the main focus; N (no) if the study does not focus on dependencies.
- **Industrial Involvement (AL-QA5):** Y (yes) if industrial parties or practitioners are actively involved in the research; N (no) if the study is purely academic without industry involvement.
- **Analytical Rigor (AL-QA6):** Y (yes) if the data analysis is thorough; P (partly) if the study is present but lacks depth; N (no) if the analysis is minimal or superficial.
- **Clear Findings (AL-QA7):** Y (yes) if there is a clear statement of findings or contributions that are relevant to the field; P (partly) if the findings are present but not clearly articulated; N (no) if the paper lacks a clear presentation of results.
- **Conclusive Alignment (AL-QA8):** Y (yes) if the conclusions are well-supported by the data and align with the objectives; P (partly) if the conclusions are present but only somewhat related to the objectives or data; N (no) if the conclusions are unrelated or absent.

Two independent researchers conducted the assessments to mitigate bias. Discrepancies were resolved through consensus or adjudication by a third reviewer. This methodical evaluation ensures that our literature review is based on scientifically sound and relevant research.

#### 3.4.2. Quality assessment for grey literature (GL-QA)

Grey literature articles were subjected to a systematic quality assessment to determine its relevance and reliability in the context of microservice dependencies. This assessment was informed by criteria specifically tailored to address the distinctive characteristics of unstructured and non-peer-reviewed sources, such as contributions on platforms like Stack Overflow and Stack Exchange. The evaluation process was critical in ensuring that the insights derived from these sources were both integrative and applicable to the domain of microservices.

- **AL-QA1 Author Reputation:** Does the author have a substantial reputation score on these platforms?
- **AL-QA2 Community Engagement**: Has the contribution received significant interaction such as upvotes?

Responses to these criteria were assigned as follows: Y (Yes) if fully met, P (Partly) if partially met, and N (No) if not met. Each criterion was valued at Y = 1,P = 0.5, and N = 0. This scoring system facilitated a quantitative assessment of each piece of grey literature's quality:

- **Reputable Author (GL-QA1):** To evaluate the credibility of authors, we utilized reputation metrics from platforms such as Stack Overflow and Stack Exchange (Stack Overflow, 2024; Meta Stack Exchange, 2008). A reputation score of 316 or higher, which aligns with the average user reputation on these platforms in 2023 (Stack Overflow Meta, 2023), was used as the threshold for a Y (Yes) rating. For Medium articles, the number of claps and comments was considered a reliable indicator of the author's reputation (Medium, 2024). However, for regular blogs, it is more challenging to quantify the author's reputation due to the absence of standardized metrics. In such cases, we relied on upvotes, likes, or similar indicators of community engagement on the post to gauge its credibility. This structured approach ensures that sources with significant community acknowledgment are included in our grey literature review, reflecting both expertise and practical relevance.
- **Engaging Article (GL-QA2):** Community engagement with grey literature was assessed by analyzing interaction levels across different platforms. On Stack Overflow and Stack Exchange, posts with at least two upvotes by community members were assigned a Y (Yes) rating, signifying substantial community endorsement of their relevance and utility (Stack Overflow, 2024; Meta Stack Exchange, 2008). Posts with one upvote received a P (Partly) rating, while those with no engagement were rated N (No), reflecting limited perceived value. For Medium articles, metrics such as the number of claps and comments were used to measure audience interaction and approval, with higher counts suggesting greater community engagement (Medium, 2024). Similarly, for Google search results, likes or upvotes on blog articles were considered indicators of relevance and audience validation. By incorporating these metrics, the evaluation framework ensures that the selected sources are not only credible but also actively engaged with by their respective communities.

To maintain impartiality, two independent reviewers conducted these assessments, resolving any discrepancies through consensus or with input from a third reviewer if necessary. This rigorous evaluation approach ensures that our synthesis of grey literature is based on contributions of verified credibility and relevance, enhancing the validity of insights into microservice dependency management.

### 3.5. Search procedure workflow

The methodology for our comprehensive literature review integrated both white and grey literature sources to ensure a thorough collection of insights on microservice dependencies. The workflow is illustrated in Fig. 1. It begins with the search execution stage, which utilizes selected data sources and tailored search queries for each source. The process for developing these components has been detailed in previous subsections. This stage yielded search results from white and grey literature sources.

For white literature, the process involved multiple stages: *title and abstract screening*, followed by a *full-text review*, and concluding with a *snowballing process* to identify additional relevant studies. In contrast, due to its distinct nature, grey literature underwent a single *full-text review* stage. These stages were guided by predefined inclusion and exclusion criteria, ensuring alignment with the study's objectives and the relevance of the included articles. Each of these steps progressively refined the pool of articles, filtering them to retain only those most relevant to the study objectives.

The final step involved a *quality assessment* for both white and grey literature, based on the criteria outlined in the previous subsection (Quality Assessment). Articles meeting the quality standards were included in the dataset for further analysis. To ensure rigor and minimize bias, each step in the screening and selection process was conducted using a dual-reviewer approach, with a third reviewer available to resolve any conflicts.

The white literature detailed stages of this workflow are outlined below.

- **Title and Abstract Screening:** Initially, titles and abstracts of academic papers were screened for relevance based on our predefined inclusion and exclusion criteria. This step ensured that only pertinent studies were considered for further review.
- **Full-Text Review:** Papers that met the initial screening criteria were subjected to a thorough full-text review to evaluate their coverage of microservice dependencies, ensuring alignment with our inclusion and exclusion criteria, fulfillment of the study objectives, and relevance to provide answers to any of the research questions.
- **Snowballing:** We performed both forward and backward snowballing on the articles included in our review so far in the process. It identified additional relevant studies that might have been overlooked in the initial search. For backward snowballing, we examined the references cited in each selected article. For forward snowballing, we used Google Scholar to explore articles that cited the selected publications. This process continues through multiple rounds until no additional papers are identified. Any additional articles identified through this process were subjected to the full screening and review stages to ensure their relevance to our study.

The grey literature detailed stage of this workflow is outlined below.

- **Grey Literature Full-Text Review:** Grey literature, encompassing blog posts, technical reports, and discussions on platforms such as Stack Overflow, Stack Exchange, and Google Search, often exists in unstructured and diverse formats. We thoroughly screened and reviewed available content, including titles, bodies, questions, answers, and comments, to extract practical insights and emerging trends from non-academic sources.

This integrated approach allowed us to capture a wide spectrum of knowledge, from peer-reviewed academic studies to real-world applications and discussions, providing a comprehensive view of the current state and challenges of managing dependencies in microservice architectures.

### 3.6. Data extraction

The data extraction phase commenced upon the successful identification and selection of relevant articles through our detailed search process. This stage involved systematically organizing the extracted information to facilitate a comprehensive review of existing methodologies and tools for managing microservice dependencies. Special attention was paid to cataloging information on dependency types, the management strategies proposed in the literature, and their reported efficacies.

This structured approach enabled us to distill significant insights and synthesize the data effectively to address our research objectives. The extraction process was designed to gather not only the types and characteristics of dependencies but also the broader impacts of these dependencies on system design and operational challenges. We documented the categories and specifics of the extracted data attributes to clarify the process and provide transparency about our methodological rigor in Table 9.

To ensure data accuracy and maintain process quality, each data attribute was extracted independently by two authors. In cases of discrepancies, a third author collaborated to resolve any conflicts.

## 4. Study execution and data synthesis

This section provides an in-depth explanation of the search procedure workflow, outlining each step of its execution. It also describes the methodology employed to present the synthesized results effectively.

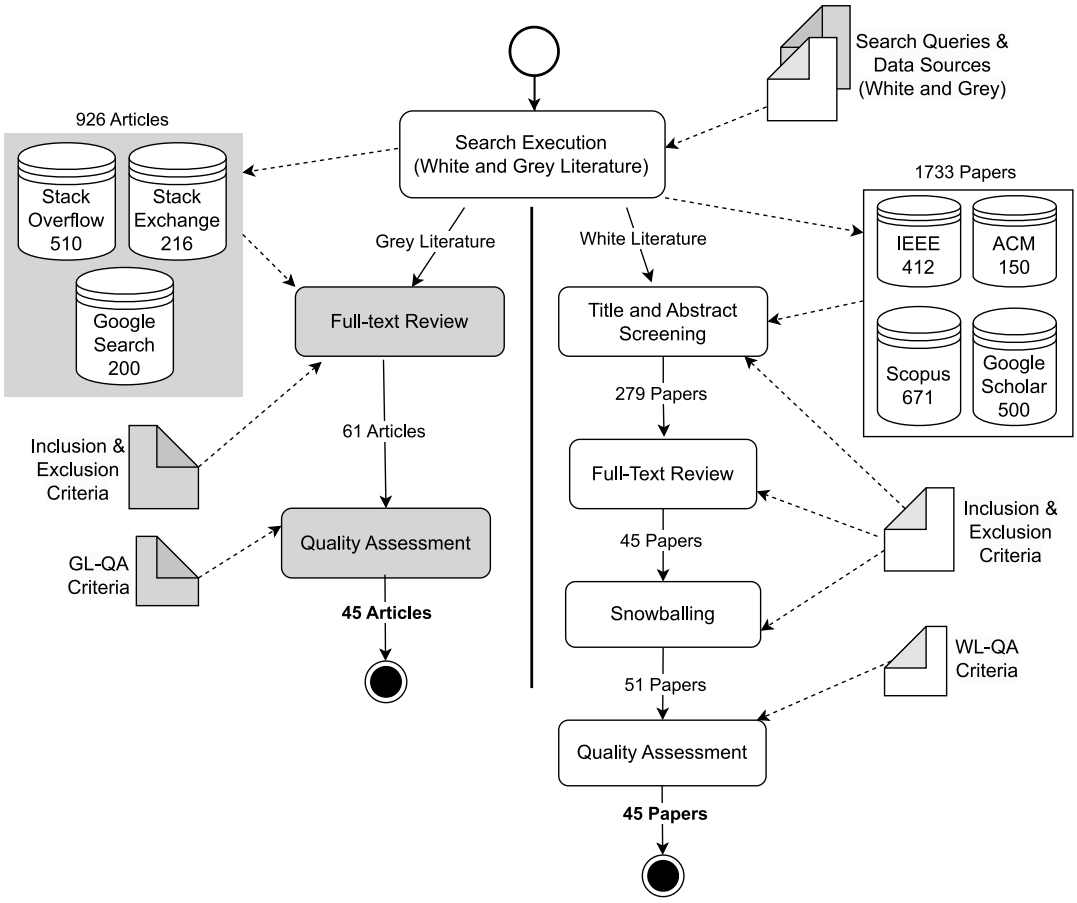


**Fig. 1.** The Multivocal search procedure workflow.

**Table 9**
Summary of extracted data attributes.

| Attribute | Details |
|---|---|
| Dependency types | Types of dependencies identified in microservices-based systems such as: control and data dependencies. And also, their associated artifacts like endpoint calls and data entities. |
| Management strategies | Techniques and methodologies employed to manage dependencies, including service discovery, API gateways, and circuit breakers. |
| Impact on system design | How dependencies influence the architecture, scalability, and resilience of microservice systems. |
| Challenges | Common challenges encountered in managing dependencies, such as increased complexity and communication overhead. |
| Benefits | Reported advantages of effective dependency management, such as improved fault isolation and enhanced modularity. |
| Tools and frameworks | Tools and frameworks utilized to identify, manage, and mitigate the impact of dependencies. |
| Case studies and benchmarks | Examples from industry or academia where dependency management strategies have been implemented, including the utilized systems benchmark. |

### 4.1. Search procedure workflow execution

The execution process[6] for the search procedure workflow includes stages for both white and grey literature, as depicted in Fig. 1.

#### 4.1.1. White literature execution

Our search for white literature commenced by retrieving a total of 1733 papers from prominent academic databases, focusing on publications from the last decade relevant to microservice architecture and dependencies. Specifically, we collected 412 papers from IEEE Xplore, 671 from Scopus, and 150 from the ACM Digital Library. For Google Scholar, we initially retrieved 2000 documents using two separate queries to address its pagination limitation, which restricts results to a maximum of 1000 per query. After removing duplicates already found in IEEE and ACM, this number was refined to 965 unique papers. To perform a targeted title and abstract search within these results, we extracted full abstracts and applied our search inquiry to both the titles and abstracts. This process yielded 500 relevant documents originating from various platforms, including MDPI, Springer, arXiv, Usenix, IOPscience, and ScienceDirect. This step was critical in ensuring we captured a wide array of research contributions spanning different aspects of microservices.

To manage and streamline our dataset, we identified and removed duplicates based on DOI and title comparisons. This deduplication process effectively reduced the collection to 1077 unique papers, setting the stage for a more focused and in-depth review aligned with our research questions.

[6] This search was conducted in January 2024.

The *title and abstract screening* stage, applying inclusion criteria such as relevance to microservice architecture, publication in English, and academic rigor, reduced the pool to 275 papers. This step involved independent reviews by two researchers to ensure unbiased selection and adherence to quality standards. The *full-text review* phase assessed the depth and relevance of discussions related to microservice dependencies in the 275 papers, which resulted in 41 papers that met all criteria. The *snowballing process*, applied to the references of these articles, identified an additional six papers, bringing the total to 47 relevant studies. This snowballing process concluded after two rounds of both forward and backward techniques, as no new papers were identified in the second round of each approach.

The final *quality assessment* was rigorously performed using a scoring system with a threshold of 3.5 points or above on an 8-point scale, focusing on factors such as methodological soundness, citation impact, and relevance to the research questions. This assessment process eliminated 2 papers and refined the selection to 45 high-quality papers.

#### 4.1.2. Grey literature execution

Parallel to our academic database search, we conducted an extensive search for grey literature primarily focusing on community-driven platforms like Stack Overflow and Stack Exchange, and utilized Google Advanced Search to access broader web content. We specifically targeted these sources to capture practical insights and real-world experiences that are not typically included in academic publications.

Regarding Stack Overflow and Stack Exchange, we systematically performed a *full-text review* on all relevant questions and answers, selecting those that were closely aligned with our research questions. The relevancy of contributions was judged based on the votes and reputations of the contributors, ensuring that we considered only the most authoritative and relevant information. For Google Advanced Search, we examined the first 200 links generated by our refined search queries, detailed in Table 7. This number was chosen to balance comprehensiveness with manageability, ensuring a thorough review without overwhelming the selection process. Each link underwent a full review where our inclusion and exclusion criteria were applied rigorously.

Furthermore, we performed a *quality assessment* of these sources by evaluating their relevance and impact. A total QA score of 1 or higher was used to determine the inclusion of content, ensuring consistency in evaluating grey literature. Six articles were eliminated in this assessment. This streamlined approach facilitated the selection of reliable and high-quality sources. It concluded with 45 grey literature articles are selected for data analysis and synthesis.

By integrating diverse materials and maintaining rigorous screening standards, our review effectively captured a comprehensive selection of literature, offering valuable insights into microservice architecture and dependencies.

### 4.2. Data synthesis

We systematically reviewed and synthesized data from a comprehensive collection of scholarly articles and grey literature. The complete dataset documenting the process execution is available in this reproducible package.[7] Each selected article, along with its title, authors, and year of publication, can be found in Tables 21, 22, 23.We identified various tools, techniques, and methods used to detect and manage microservice dependencies.

The data synthesis process began with extracting relevant attributes and their associated values from the included articles, as summarized in Table 9. To refine the data for each extracted attribute, we removed duplicates and standardized synonymous terms and concepts, such as aligning "Data" with "Data Entity" and "Remote Call" with "Endpoint Call". This step ensured a unified foundation for deeper analysis.

To perform deeper synthesis for deriving more meaningful insights, we employed a structured thematic analysis approach (Cruzes and Dyba, 2011). This method was used to develop a dependency taxonomy based on the extracted dependency types and their associated artifacts. The process followed three stages: Code Data, Translate Codes Into Themes, and Create a Model of Higher-Order Themes (Taxonomy). Five authors worked collaboratively, engaging in iterative rounds of group discussions to refine the codes, categories, and taxonomy. Each phase involved revisiting relationships and connections, with adjustments made until consensus was reached. This iterative methodology ensured the taxonomy provided clarity to the complex and unstructured dependencies identified across multiple literature sources.

*Stage 1: Code Data.* During this stage, we utilized the open-coding technique (Strauss and Corbin, 2004) to analyze the extracted dependency type names, their descriptions, and artifacts. Codes were assigned as concepts emerged, focusing on whether the dependencies were centered around similar artifacts. This inductive approach allowed categories to naturally form from the data rather than relying on predefined classifications.

*Stage 2: Translate Codes Into Themes.* In this stage, we applied axial coding (Kendall, 1999) to systematically group the initial codes into more abstract categories. Axial coding enabled us to connect and refine the relationships between codes, creating meaningful clusters. This iterative process identified core themes and clarified the connections between different dependency types. It established clear relationships within the coded dependency types.

*Stage 3: Create a Model of Higher-Order Themes (Taxonomy).* After constructing themes, we further explored them to develop a taxonomy. This formal classification system organized the themes into higher-order categories and identified their interrelationships. Axial coding continued to play a crucial role in this step, ensuring a consistent structure and highlighting the connections among the dependency types and artifacts.

## 5. Results

This section presents the findings from our comprehensive analysis of microservice dependencies and their management tools and techniques. We will address our research questions, discuss our findings across different types of dependencies, and review the various tools and techniques used to detect, analyze, reveal, manage, optimize, model, and recover dependencies in microservices-based systems. Additionally, we detail the benchmarks used in the literature to test and demonstrate dependencies. This multi-faceted approach ensures a robust and thorough understanding of the current state of microservice dependency management.

The relationships among the findings of the research questions are illustrated in Fig. 2, with dependencies at the center of this study and the results of each research question. RQ1 introduces the dependency taxonomy, detailing various dependency types and their connections to system artifacts. These dependencies serve as a foundation, linking to the tools and techniques discussed in RQ2, along with the benchmarks (RQ2.1) that these tools use to identify dependencies within the system. Finally, the dependencies are analyzed in relation to their quality impacts on different architectural aspects, as explored in RQ3.

### 5.1. RQ1: What dependencies related to microservices are recognized in the literature?

The literature recognizes various forms of dependencies, which are critical for understanding both individual and collective interactions within microservices. Dependencies are typically characterized as the explicit or implicit connections between components or artifacts, such as data entities, service endpoints, or methods within these systems.

[7] Reproducible Package: https://zenodo.org/records/14208294.

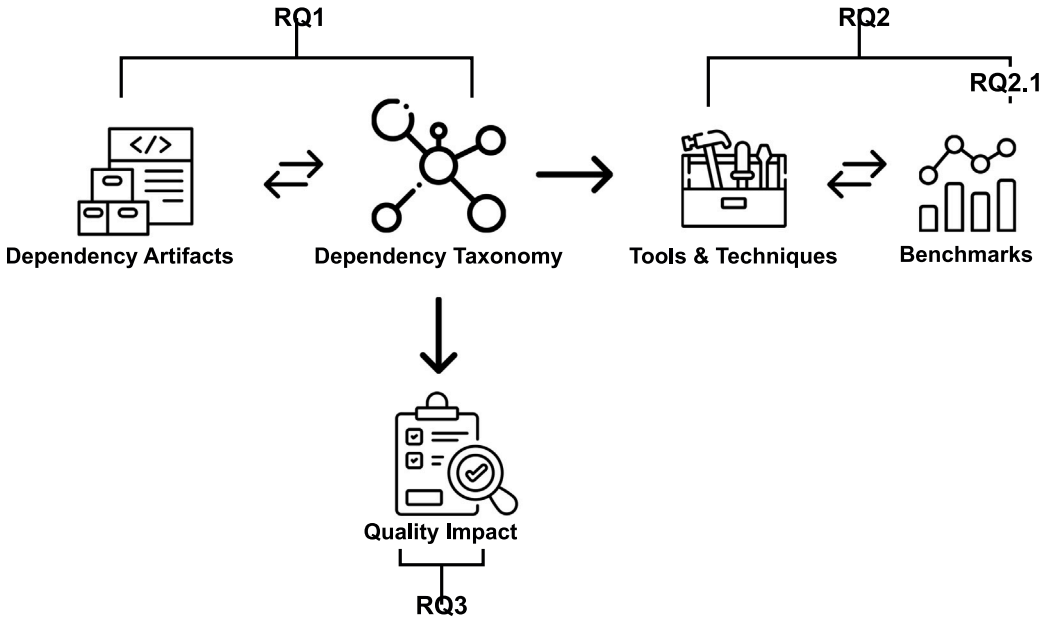


**Fig. 2.** Research questions overview.

These are closely linked to the concept of coupling, which pertains to the degree of direct knowledge one microservice has of another. This influences the flexibility and scalability of the system, with tight coupling often leading to maintenance and scalability challenges due to high interdependency.

Despite the complex nature of dependencies within microservices-based systems, there appears to be a notable gap in the existing literature concerning their categorization. Dependencies, which manifest in diverse forms, ranging from data dependencies dictating the flow of information within systems to inter-microservice dependencies that manage how components interact, lack a systematic categorization framework. Instead, researchers consider different categories depending on their specific requirements. Additionally, while the literature may not explicitly mention categorizations, it is evident that each of these dependencies possesses unique characteristics.

To address this research question, we started by identifying and analyzing the various artifacts present in the literature. Next, we extracted and examined the different dependency types and their associated artifacts from the selected articles. Finally, we synthesized this information to develop a dependency taxonomy, filling the gap in existing categorizations within the literature.

#### 5.1.1. Dependency artifacts

Dependency artifacts are the components that establish and sustain dependencies between microservices. For instance, control dependencies may arise through endpoint artifacts when one microservice relies on another via endpoint calls. Similarly, control dependencies can also occur through database transaction artifacts, where transactions form the basis of the dependency. Data dependencies, on the other hand, may emerge between microservices that share overlapping data entity artifacts. Consequently, changes to these artifacts directly impact the connected microservices, necessitating corresponding modifications.

We identified these artifacts from the literature, synthesized the data to eliminate duplicates, and analyzed definitions to group synonyms, ensuring clarity and focus. The artifacts and their definitions are summarized and presented in Table 10.

#### 5.1.2. Dependency types found in the literature

The dependency types are articulated in the literature with varying perspectives. A dependency between two entities occurs when a change in one requires a corresponding change in the other. To analyze this, we reviewed the selected literature, extracted the specified relevant information and attributes, synthesized the data to remove duplicates, and grouped synonyms for consistency. Additionally, we identified the relationship between each dependency type and its associated artifacts within the system.

As shown in Tables 11, 12, 13, the Dependency (Literature) column lists the dependency types that are identified in the literature, the Definition column provides their definitions, the Associated Artifacts column lists the corresponding artifacts involved in each dependency relationship, and the Reference column cites the sources addressing these dependency types.

It is important to note that the values in the Associated Artifacts column are referenced from Table 10. However, some rows include the term *Various*, indicating that multiple artifacts of differing natures and categories could be associated with that dependency type. For instance, an Endpoint artifact may represent a control dependency, whereas a Data Entity artifact may reflect a data dependency. This variation is a key factor in classifying such dependencies as symptoms of Various Dependencies in our developed taxonomy (Dependency (Developed Taxonomy) column). This developed taxonomy classification will be discussed later in this section.

#### 5.1.3. Dependency taxonomy development

In microservice architectures, each type of dependency influences system operations and is closely linked to various system artifacts. A review of the dependencies in the literature revealed overlaps among them and highlighted some misconceptions, where certain symptoms were repeatedly mentioned as actual causes for dependencies. Therefore, we employed a thematic analysis approach to develop a taxonomy categorizing the different dependencies in microservice architectures mentioned the literature.

The taxonomy is centered around system artifacts, acknowledging that dependencies can also be classified from various perspectives, such as operational stages, execution phases, direction relationships like direct or indirect, and analysis techniques like static or dynamic analysis. However, we adopted a concept-oriented approach for categorizing dependencies, focusing on the definitions and the types of artifacts involved in the dependency relationships. Other perspectives are addressed in some detail, particularly within the symptom categories.

**Table 10**
Identified dependency artifacts found in the literature. *Note:* a.k.a ((As Known As)) indicates synonyms.

| Artifacts | Definition | Reference |
|---|---|---|
| Method Definition | Represents the declaration of methods, outlining their functionality and purpose. Relevant for microservice interaction and implementation tracking. | Daoud et al. (2021) and Saidi et al. (2023) |
| Method Call | Indicates dependencies arising from one method invoking another, a common interaction in microservice communication. | Daoud et al. (2021), Saidi et al. (2023), Link (2018) and Nitin et al. (2023) |
| Input Parameter | Dependency through parameters provided as input to services, influencing service behavior and outcomes. | Grohmann (2022) |
| Data Entity (**a.k.a.** Database Table) | Refers to the specific tables or entities within a database that microservices share, leading to dependencies between services. | Engel et al. (2018), Abdelfattah and Cerny (2023b), Manish Kumar (2018), KitKarson (2021) and Abrar (2019) |
| Database Instance | Dependency based on shared database instances, where multiple microservices interact with the same database. | Shah (2015) |
| Database Transaction | Arises from operations spanning multiple microservices, requiring atomicity in database interactions. | Nitin et al. (2023) |
| Container Instance | It is responsible for hosting and executing microservices within a containerized environment (e.g., Kubernetes), ensuring service isolation, scalability, and efficient management. | Pai (2023) |
| Message Events | Dependency through asynchronous message exchanges between services, often facilitated by message brokers. | Engel et al. (2018), Tsonev (2017), Kriens (2018) and MicroMaster (2019) |
| Endpoint (**a.k.a.** Remote Call, Rest Call, HTTP Call) | Dependency caused by microservices interacting through defined API endpoints, including REST and HTTP calls. | Daoud et al. (2021), Saidi et al. (2023), Nitin et al. (2023), Abdelfattah and Cerny (2023b), Tsonev (2017), Link (2018), Kriens (2018), MicroMaster (2019), Zuo et al. (2021), Song et al. (2023), Zhou et al. (2023), Abdelfattah and Cerny (2023a), Apolinário and de França (2021), Wang et al. (2023), Silva et al. (2021), Gortney et al. (2022), Zhong et al. (2023), Genfer and Zdun (2021), Lv et al. (2024), Yang et al. (2023), Fekete et al. (2023), Engel et al. (2018), Raj and Sadam (2021), Rahman et al. (2019) and Wilta and Kistijantoro (2023) |
| Business Process | Dependencies inherent in orchestrating complex business workflows across multiple microservices. It includes multiple artifacts of Method Call, Method Definition, Endpoint, and Data Entity. | Saidi et al. (2023) and Flater (2021) |
| Library (**a.k.a.** Shared Code, Shared Module) | Dependency resulting from common codebases or libraries used across microservices. | Link (2018) and Rewari (2020) |
| Environment Variable | A configuration setting stored outside the service, controlling behavior like database connections or API keys, and influencing service operations across environments. | Fekete et al. (2023) and Pai (2023) |
| Network Resource | The underlying network components, such as transmission channels, enable exchanging information and requests across the system. | Chugh and Ahire (2022) |
| Requirement Documents | Dependencies originating from shared requirement specifications that inform multiple microservices. | Daoud et al. (2021) |
| SLA Document | Dependency through shared service-level agreements that outline expectations for service operation. | Daoud et al. (2021) |
| Policy Document | Outlining rules and guidelines that govern the behavior and interactions of microservices, such as the GDPR policy document, which dictates how data must be handled to ensure privacy and regulatory compliance within the system. | Pai (2023) |

The taxonomy identified 28 distinct dependency types from the literature after they were refined by removing duplicates and merging synonyms. It is organized into six primary dependency categories (D1 to D6) and four symptom categories (S1 to S4) associated with these dependencies. These taxonomy categories are detailed in Tables 11, 12, 13, and illustrated in Figs. 3, 4.

In **D1. Data Dependency**, a relationship is formed where the system's functionality is influenced by shared or linked data between

**Table 11**
Summary of dependencies and definitions identified — Part (1/3).

| Dependency (Developed taxonomy) | Dependency (Literature) | Definition | Associated artifacts | Ref. |
|---|---|---|---|---|
| D1. Data Dependency | Data Schema (**a.k.a** Data Dependency) | Represents dependencies arising from shared or linked data, including specific tables or schemas within a SQL database that multiple microservices rely on. These dependencies may also stem from data entity classes in the source code. | Data Entity | Abdelfattah and Cerny (2023b), Manish Kumar (2018), KitKarson (2021), Abrar (2019) and Engel et al. (2018) |
| | Parametric Dependency | Represents the relationship between input parameters and their impact on the system's performance or behavior. | Input parameter | Grohmann (2022) |
| D2. Control Dependency | Control Flow Dependency (**a.k.a** Control Dependency) | Represents dependencies based on the execution order and logical conditions governing the flow of activities. | Method Call, Endpoint | Daoud et al. (2021), Saidi et al. (2023) and Link (2018) |
| | Synchronous Dependency (**a.k.a** Endpoint Dependency) | Represents a dependency where one service must wait for a response from another service to proceed. | Endpoint | Abdelfattah and Cerny (2023b), Tsonev (2017), Zuo et al. (2021), Song et al. (2023), Zhou et al. (2023), Abdelfattah and Cerny (2023a), Apolinário and de França (2021), Wang et al. (2023), Silva et al. (2021), Gortney et al. (2022), Zhong et al. (2023), Genfer and Zdun (2021), Lv et al. (2024), Yang et al. (2023), Fekete et al. (2023), Engel et al. (2018), Raj and Sadam (2021), Rahman et al. (2019) and Wilta and Kistijantoro (2023) |
| | Asynchronous Dependency (**a.k.a** Event Dependency) | Represents a dependency where services communicate via message brokers or remote calls, enabling asynchronous interactions and operational independence. | Message Event, Endpoint | Engel et al. (2018), Tsonev (2017), Mangano (2022) and Bräutigam (2019) |
| | Call-Return Dependency | Represents dependencies arising from method invocations between source and target methods in object-oriented programming, typically following a call-return interaction pattern. | Method Call, Endpoint | Nitin et al. (2023) |
| | Transaction Dependency | Represents dependencies related to code fragments that are involved in database transactions | Database Transaction | Nitin et al. (2023) |
| D3. Resource Dependency (**a.k.a** Technology) | Source Code Dependency (**a.k.a** Code-level Dependency) | Represents a dependency arising from shared code or libraries between different components or services in the system. | Library | Link (2018), CodeScene Engineering Team (2023) and Zhang et al. (2024) |
| | Core Dependency | Represents critical dependencies that are essential for the core functionality of a system. For example, dependencies like those between .NET and .NET Core frameworks. | Library | Rewari (2020) |
| | Starter Dependency | Represents dependencies required for system initialization, such as frameworks or templates that streamline project setup and configuration. For example, in Spring Boot, starter dependencies provide pre-configured setups for common functionalities like web applications. | Library | Rewari (2020) |
| | Database Dependency | Represents dependencies that arise when multiple services share a common database, creating inter-service dependencies based on data access. | Database Instance | Shah (2015) |

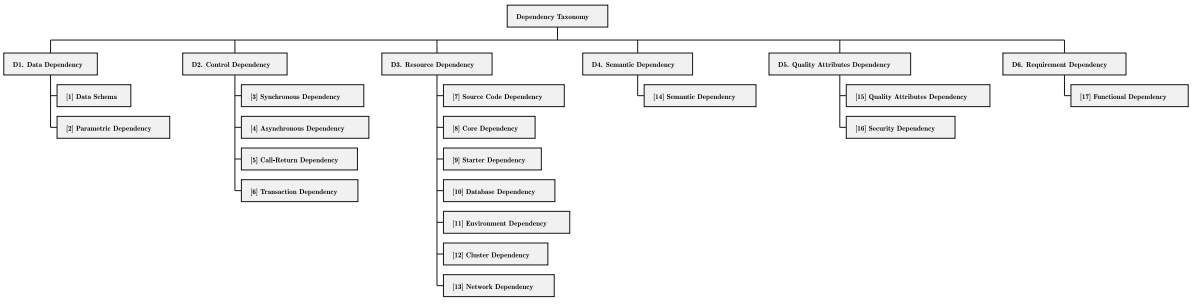


**Fig. 3.** Dependency taxonomy.

microservices. It encompasses two types of dependencies: Data Schema Dependency and Parametric Dependency. The *Data Schema Dependency* arises when multiple microservices interact or overlap specific data entities, such as database schemas or source code entity classes (e.g., ORM and DTO). As highlighted in the literature (Abdelfattah and Cerny, 2023b; Manish Kumar, 2018, KitKarson, 2021; Abrar, 2019; Engel et al., 2018), changes to schema design can have cascading effects on services relying on these entities. On the other hand, *Parametric*

**Table 12**
Summary of dependencies and definitions identified — Part (2/3).

| Dependency (Developed taxonomy) | Dependency (Literature) | Definition | Associated artifacts | Ref. |
|---|---|---|---|---|
| D3. Resource Dependency (**a.k.a** Technology) | Environment Dependency (**a.k.a** Mount Dependency, Certificate Dependency) | Represents dependencies based on environment variables, such as configuration settings or credentials. Changes in these variables can impact service behavior across different environments (e.g., development, production). | Environment Variable | Fekete et al. (2023) and Pai (2023) |
| | Cluster Dependency (**a.k.a** Container Dependency, Pod Dependency) | Represents dependencies that occur when services rely on specific containerized environments, clusters, or pods to function, impacting their deployment, scaling, and communication within the infrastructure. | Container Instance | Pai (2023) and Zhang et al. (2024) |
| | Network Dependency | Represents dependencies that occur when microservices exchange data or trigger actions through network protocols (e.g., HTTP, gRPC, or messaging queues). The reliance on the network as a medium for communication between microservices and each other. | Network Resource | Chugh and Ahire (2022) |
| D4. Semantic Dependency | Semantic Dependency | Represents dependencies based on functional similarities between activity names, used to group related functions that serve similar purposes within the system. | Method Definition | Daoud et al. (2021) and Saidi et al. (2023) |
| D5. Quality Attributes Dependency | Quality Attributes Dependency | Represents dependencies related to quality attributes such as security, scalability, and maintainability, which define the non-functional requirements and overall performance characteristics of a system. | SLA Document, Requirement Document | Daoud et al. (2021) |
| | Security Dependency | Represents dependencies on security mechanisms, such as authentication, authorization, encryption, and access control, that ensure the confidentiality, integrity, and availability of microservices. | Policy Document, Requirement Document | Pai (2023) |
| D6. Requirements Dependency | Functional Dependency (**a.k.a** Feature Dependency, Domain Dependency) | Represent dependencies based on business process flows, where one service depends on another to complete its tasks. This type of dependency involves both data and control dependencies and is often categorized within Domain-Driven Design (DDD) to structure system sub-domains and activities. | Business Process | Saidi et al. (2023), Flater (2021), Walpita (2020) and Pai (2023) |

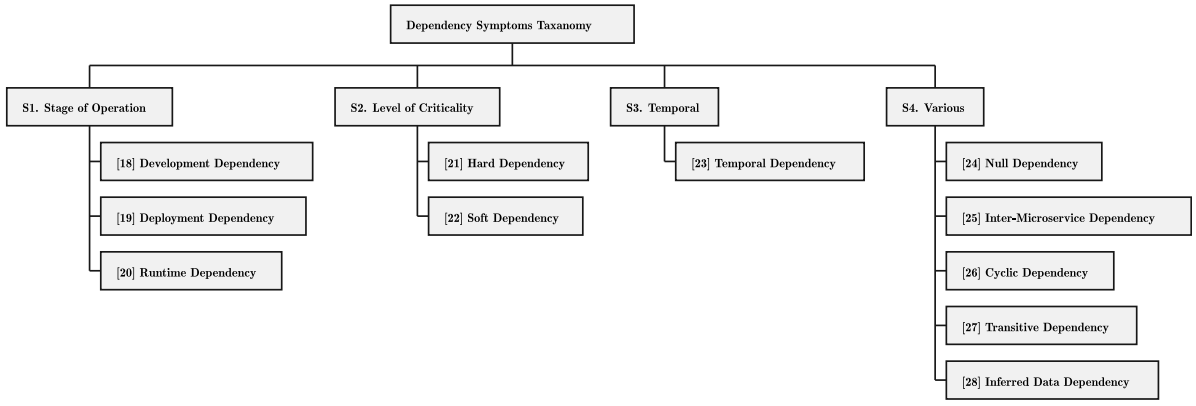


**Fig. 4.** Dependency symptoms taxonomy.

*Dependency* refers to how input parameters influence the dependencies between microservices during communication. It also describes the relationship between the input parameters of a component and its performance properties. As Grohmann (2022) describe, parameter-based configurations define a service's behavior and impact its execution.

The **D2. Control Dependency** refers to dependencies that are derived from the interaction between microservices and each other. It includes six dependencies extracted from the literature. The *Control Flow Dependency* involves dependencies based on the flow of execution and logical conditions. It emphasizes how the sequence of operations in one service can dictate the flow in another. Method calls and endpoint interactions serve as artifacts in these dependencies (Daoud et al., 2021; Saidi et al., 2023, Link, 2018). *Synchronous Dependency* describes cases where one service waits for a response from another before proceeding. This is common in tightly coupled systems where immediate responses are required for subsequent actions. The Endpoint artifact facilitates this dependency by establishing synchronous communication (Abdelfattah and Cerny, 2023b; Tsonev, 2017, Zuo et al., 2021, Song et al., 2023; Zhou et al., 2023; Abdelfattah and Cerny, 2023a; Apolinário and de França, 2021; Wang et al., 2023; Silva et al., 2021; Gortney et al., 2022; Zhong et al., 2023; Genfer and Zdun, 2021; Lv et al., 2024; Yang et al., 2023; Fekete et al., 2023; Engel et al., 2018; Raj and Sadam, 2021; Rahman et al., 2019; Wilta and Kistijantoro, 2023). However, *Asynchronous Dependency* occurs when communication between microservices does not require immediate acknowledgment or response from the receiving service. This type of dependency often involves message queues or event-driven architectures but can also occur through remote calls that return immediately without a response being consumed by the caller. These interactions allow services to operate independently, achieving temporal and execution decoupling.

**Table 13**
Summary of dependencies and definitions identified — Part (3/3) (*Note:* Various refer to components in multiple dependencies (e.g., Data, Endpoint)).

| Dependency (Developed taxonomy) | Dependency (Literature) | Definition | Associated artifacts | Ref. |
|---|---|---|---|---|
| S1. Symptom (Stage of Operation) | Development Dependency (**a.k.a** Design-time Dependency, Implementation Dependency) | Represents the stage of development where services must be designed, developed, and tested together. This type of dependency can indicate various relationships, such as code, data, or endpoint dependencies, that must be managed during the development process to ensure successful integration. It also includes testing dependencies between tests. | Various | Euphoric (2020), Pai (2023), Richardson (2021, 2023), Walpita (2020) and Smith et al. (2023) |
| | Deployment Dependency (**a.k.a** Packaging Dependency) | Represents the stage of deployment where services must be deployed together. This type of dependency highlights the need for coordination between services during deployment, often involving infrastructure and resource dependencies to ensure proper integration in the production environment. | Various | Euphoric (2020), Walpita (2020), Pai (2023) and Zhang et al. (2024) |
| | Runtime Dependency (**a.k.a** Observability dependency) | Represents the runtime stage where services operate as a cohesive unit, and the failure of one service can impact others. Observability platforms track and correlate this telemetry in real-time, aiding in troubleshooting and providing insights into system performance and issues. | Various | Flater (2021), Next (2019), Pai (2023), Richardson (2021, 2023), Rossom (2017), Raj and Sadam (2021) and Gortney et al. (2022) |
| S2. Symptom (Level of Criticality) | Hard Dependency | Represents a level of dependency where the failure of one service completely halts the operation of another. Hard dependencies are critical components that a system cannot function without, directly impacting the system's availability. These can include hardware, software, or other distributed systems. | Various | Abdolhosseini (2022) |
| | Soft Dependency | Represents a dependency where the failure of one service degrades functionality but does not halt others. Soft dependencies are less critical, often going unnoticed or tolerated temporarily. These can include non-essential components like optional services, hardware, or software. | Various | Abdolhosseini (2022) |
| S3. Symptom (Temporal) | Temporal Dependency | Represents dependencies where the current state or value of a microservice is influenced by its past states, often involving timing or sequence of events that affect service behavior over time. | Various | Zuo et al. (2021) and Walpita (2020) |
| S4. Symptom (Various) | Null Dependency | Represents dependencies on non-existent or irreplaceable nodes within the system, which can arise as a consequence of various types of dependencies. | Various | Ma et al. (2019b) |
| | Inter-Microservice Dependency (**a.k.a** Service Dependency, Application Dependency) | Represents interactions that fulfill business processes across multiple microservices, involving various types of dependencies such as data, control, and functional dependencies, to enable seamless integration and functionality between services. | Various | Zheng et al. (2023), Zhong et al. (2023), JonasH (2021), Chugh and Ahire (2022), Razorops, Inc. (2023), Mangano (2022), Burghardt (2023), Martin (2020), Pai (2023), Zuo et al. (2021), Song et al. (2023), Zhou et al. (2023), Abdelfattah and Cerny (2023a), Wang et al. (2023), Silva et al. (2021), Gortney et al. (2022), Genfer and Zdun (2021), Lv et al. (2024), Yang et al. (2023), Fekete et al. (2023), Engel et al. (2018), Raj and Sadam (2021), Rahman et al. (2019) and Wilta and Kistijantoro (2023) |
| | Cyclic Dependency (**a.k.a** Circular Dependency) | It occurs when components depend on each other, forming a loop. It often results from control or data dependencies, hindering system flexibility and scalability. | Various | Hijazi (2016), Shiloah (2023), Mangano (2022), Sebrechts (2016), theDmi (2016), gsf (2017), Chugh and Ahire (2022), Genfer and Zdun (2021), Kaushik et al. (2022) and Apolinário and de França (2021) |
| | Transitive Dependency (**a.k.a** Indirect Dependency) | This dependency occurs when services rely on each other through intermediary services, such as message queues. This can be a consequence of control dependencies | Various | Kriens (2018), MicroMaster (2019) |
| | Inferred Data Dependency | Inferred data dependencies occur when one microservice requires data generated by another microservice, often inferred indirectly through interactions. This can be a consequence of control or data dependencies. | Various | downtownthe (2018) |

Key artifacts in this dependency type include Message Events and Endpoints (Engel et al., 2018, Tsonev, 2017; Mangano, 2022; Bräutigam, 2019). *Call-Return Dependency* emphasizes method invocations between source and target methods in an object-oriented programming context. It describes the interaction pattern where one service calls another and expects a return, often triggering further operations (Nitin et al., 2023). In the other hand, *Transaction Dependency* arises when services interact within database transactions, making them dependent on each other for consistency. This is crucial in systems where operations span multiple services and need atomicity in transactions (Nitin et al., 2023).

Moreover, **D3. Resource Dependency (a.k.a Technology Dependency)** occurs when services are interdependent based on shared technologies or resources like code libraries, databases, or even environment-specific settings. This category includes seven dependencies from the literature. *Source Code Dependency* is where shared libraries or code fragments between services lead to tight coupling. For example, dependencies on specific libraries or source code fragments across microservices lead to tighter integration between them (Link, 2018; CodeScene Engineering Team, 2023). Similarly, *Core Dependency* identifies critical shared resources essential for a system's functionality, such as core frameworks or libraries. Furthermore, these core dependencies are fundamental for system operation and cannot be easily decoupled (Rewari, 2020). *Starter Dependency* appears from initialization components required to set up services, such as framework configurations. In systems like Spring Boot, starter dependencies enable rapid setup of common functionality (Rewari, 2020). *Database Dependency* indicates when multiple services access the same shared database, leading to dependencies based on data access. Shared database instances serve as the artifact in these dependencies (Shah, 2015). *Environment Dependency* refers to configuration settings that vary across environments, such as development or production. This includes environment variables, such as credentials or database connections, which are used to control service behavior. Additionally, it includes Mount Dependency, which involves dependencies on mounted file systems like configurations file or configurations map. It is also considered as a Certificate Dependency, which pertains to reliance on specific security certificates for communication or authentication. These dependencies highlight the critical role of environment-specific configurations in ensuring smooth and secure service operations. As highlighted in Fekete et al. (2023) and Pai (2023), changes in these variables can significantly impact the deployment and functionality of services. *Cluster Dependency* explains how services depend on specific containerized clusters, containers, or pods within platforms like Docker and Kubernetes. These dependencies influence deployment, scaling, and service communication across microservices (Pai, 2023). *Network Dependency* refers to the dependency on the communication medium and channels that facilitate interactions between microservices. When microservices rely on network protocols like HTTP, gRPC, or message queues for data exchange, they form a network dependency. The Network Resource artifact underpins these interactions, ensuring services can communicate over the network as highlighted in Chugh and Ahire (2022).

**D4. Semantic Dependency** emerges from functional similarities between services or activities that group related services, facilitating the structuring of services based on common functionality. It is mentioned in the literature as it connects services that perform similar tasks or belong to similar functional domains. As shown in Daoud et al. (2021) and Saidi et al. (2023), these dependencies help in indicating related functions together, optimizing their deployment and operation across microservices. It also highlights the importance of maintaining consistency across semantically similar components, requiring careful governance to ensure business alignment throughout the system.

The Quality Attributes Dependency and Requirements Dependency includes artifacts at the documentation level, highlighting the presence of these dependencies from the system's inception. These dependencies persist throughout the system's evolution during different phases. **D5. Quality Attributes Dependency** is concerned with non-functional aspects such as security, scalability, and maintainability that impact how services behave in production environments (Daoud et al., 2021). However, the literature mentioned *Security Dependency* as a separate dependency in the system. It underscores the reliance of services on security mechanisms such as authentication, authorization, and encryption to protect data and service interactions. Security policies and requirement documents define the rules and guidelines that govern these dependencies, ensuring compliance and safeguarding system integrity (Pai, 2023). On the other hand, **D6. Requirement Dependency** is concerned with how business processes or domain-driven designs shape the dependencies between services, ensuring that services collaborate effectively to achieve business goals. It includes *Functional Dependency* that reflects how one service depends on another to complete tasks within a business process. These dependencies are typically mapped to business workflows and domain-driven design (DDD) principles (Saidi et al., 2023, Flater, 2021; Walpita, 2020; Pai, 2023).

Furthermore, the taxonomy incorporates symptoms of dependencies that are identified in the literature as dependency types, highlighting misconceptions and the lack of clear guidelines in both academic and industrial contexts. These symptoms represent dependencies that emerge as a consequence or effect of other dependencies (D1–D6) within the system. The taxonomy considers four categories of these symptoms as follows: operational stages, levels of criticality, temporal factors, and cross-service interactions. Each dependency type is further contextualized with insights from the literature and its associated artifacts.

**S1. Symptom (Stage of Operation)** discusses that different levels in the team operation include multiple dependencies rather than considering them as dependencies by their own. Three different stages of operations are considered in the literature. *Development Dependency* refers to dependencies that must be managed during the design and implementation phases. These dependencies often involve shared code, data entities, or Endpoints that require careful integration to avoid issues later in the lifecycle. Testing dependencies, where test cases rely on the availability or correctness of specific services, are a critical component. As mentioned in Euphoric (2020), Pai (2023), Richardson (2021, 2023), Walpita (2020) and Smith et al. (2023), these dependencies highlight the need for robust modularity during the development phase. *Deployment Dependency* arises when services require coordinated deployment to function properly. This can be due to shared infrastructure, configurations, or interdependent services. As described in Euphoric (2020), Walpita (2020) and Pai (2023), such dependencies pose challenges in CI/CD pipelines and often necessitate container orchestration tools to streamline deployments. *Runtime Dependency* encapsulates the interconnections between microservices during runtime execution. Failures in one service can cascade to others, making observability and real-time monitoring essential for resilience. Observability tools track runtime dependencies, as highlighted by Flater (2021), Next (2019), Pai (2023), Richardson (2021, 2023), Rossom (2017) and Raj and Sadam (2021), enabling rapid issue resolution and system performance analysis.

Moreover, **S2. Symptom (Level of Dependency)** emphasizes that the literature identifies dependencies based on their criticality level, categorizing them as either Hard Dependencies or Soft Dependencies, as discussed in Abdolhosseini (2022). *Hard Dependency* represents an essential connection where the failure of one component halts the system's operation entirely. For example, a database service critical for order processing exemplifies a hard dependency. Conversely, *Soft Dependency* indicates a non-critical relationship, where failures degrade, rather than completely disrupt, functionality. These distinctions are crucial for designing fault-tolerant architectures.

For taking state changes in dependency consideration, **S3. Symptom (Temporal)** focuses on the time-sensitive relationships where the current behavior of a microservice depends on its historical states.

Such dependencies are common in event-driven systems or workflows requiring sequential task execution. As highlighted in Zuo et al. (2021) and Walpita (2020), managing these dependencies often involves event sourcing or log-based replay mechanisms to ensure consistency and correctness.

**S4. Symptom (Various)** represents dependencies discussed in the literature that are. However, we classified them as manifestations of one or more other dependencies outlined in the taxonomy (D1–D6). *Null Dependency* reflects scenarios where services depend on non-existent or unavailable components, often resulting from version mismatches or deployment issues Ma et al. (2019b). This dependency can arise due to the absence of control dependencies, data dependencies, or any other dependencies. *Inter-Microservice Dependency* captures the multifaceted interactions required to fulfill complex business processes across services. These dependencies often combine data, control, and functional dimensions, as highlighted in Zheng et al. (2023), Zhong et al. (2023), JonasH (2021), Chugh and Ahire (2022), Razorops, Inc. (2023), Mangano (2022), Burghardt (2023), Martin (2020), Pai (2023), Zuo et al. (2021), Song et al. (2023), Zhou et al. (2023), Abdelfattah and Cerny (2023a), Wang et al. (2023), Silva et al. (2021), Gortney et al. (2022), Zhong et al. (2023), Genfer and Zdun (2021), Lv et al. (2024), Yang et al. (2023), Fekete et al. (2023), Engel et al. (2018), Raj and Sadam (2021), Rahman et al. (2019) and Wilta and Kistijantoro (2023). Literature emphasizes strategies like contract testing to manage these dependencies effectively. *Cyclic Dependency*, or circular dependency, occurs when services form a dependency loop between endpoints, data, or any other shared communications. Such configurations, discussed in Hijazi (2016), Shiloah (2023), Mangano (2022), Sebrechts (2016), theDmi (2016), gsf (2017), Chugh and Ahire (2022), Kaushik et al. (2022) and Apolinário and de França (2021), are detrimental to scalability and flexibility, often necessitating architectural refactoring to resolve. *Transitive Dependency* represents indirect relationships where one service depends on another through intermediaries, such as message brokers (Kriens, 2018; Hijazi, 2016). *Inferred Data Dependency* arises when services indirectly depend on data from other services, often through interaction patterns, shared data, or resources (downtownthe, 2018).

### 5.2. RQ2: What techniques and approaches have been used to detect dependencies, and what tools have been used?

The literature explores the various techniques and tools utilized to detect dependencies within software systems. We aim to investigate the methodologies and approaches employed and to provide insights into how developers identify and manage dependencies effectively. In this section, examinations of some outstanding tools will be detailed.

The tools were divided into categories based on their primary functionalities and objectives and are listed in Tables 14, 15, 16, and 17 The number of tools placed in each category is listed in Fig. 5. Some tools were placed in multiple categories.

*(1) Dependency analysis tools*

These tools focus on examining existing dependencies within software systems, often employing static or dynamic analysis techniques. They provide insights into the structure and nature of dependencies, thus facilitating a deeper understanding of the software's architecture. By identifying dependencies, developers can anticipate potential risks associated with changes or updates to the system and make informed decisions to mitigate them. Additionally, these tools aid in identifying redundant or unnecessary dependencies, allowing for optimization of the software's structure and enhancing its overall performance and maintainability.

ChainsFormer (Song et al., 2023) detects dependencies by analyzing communication-based interactions among microservices. It identifies critical chains and nodes within the microservice architecture to optimize resource allocation. This approach enables ChainsFormer to dynamically scale CPU and memory resources based on system state and chain characteristics, thereby improving resource provisioning efficiency. In contrast, GSMART (Ma et al., 2019a) facilitates dependency analysis by creating service dependency graphs (SDGs) to visualize microservice-based systems. By parsing microservice projects, GSMART generates SDGs that enable developers to visualize system dependencies, trace relationships among microservices, and select regression test cases. Additionally, GSMART aids in microservice retrieval for reuse, promoting system robustness and maintainability.

SYMBIOTE (Apolinário and de França, 2021) focuses on monitoring service coupling metrics to detect dependencies and potential architectural degradation. It continuously analyzes service coupling metrics to warn developers and architects of coupling issues that may negatively impact system maintainability. By distinguishing regular increases in coupling from harmful degradation, SYMBIOTE promotes system robustness over time. In contrast, TraceNet (Yang et al., 2023) specializes in root cause localization by analyzing tracing data to identify microservice interactions and dependencies. It accurately locates the cause of failures by analyzing and tracing data and addressing challenges related to dynamic invocations, anomaly propagation, and diverse microservices. TraceNet's lightweight root cause localization methods contribute to effective failure diagnosis in microservice architectures.

Comprehensive Microservice Extraction Approach (CMEA) (Bajaj et al., 2024) identifies dependencies through static code analysis of monolithic applications. It analyzes class-level and method-level dependencies to identify cohesive sets of classes and methods that can form microservices. By grouping business classes and methods based on dependency patterns, CMEA addresses challenges related to microservice extraction, promoting maintainable and scalable microservice architectures.

Triple (Ren et al., 2023) offers an interpretable anomaly detection approach based on deep learning, while also contributing to dependency analysis. Triple leverages trace's dependency relationships to construct a unified graph representation, where nodes' features are processed logs and metrics. Then, Triple trains an STGCN-based Deep SVDD model, which extracts key information from the unified graph representation and constructs a data-enclosing hypersphere in output space. Triple judges whether the time-series data is anomalous based on its score, which is the distance between the data and the hypersphere's center. In contrast to other tools that primarily focus on dependency analysis, Triple integrates anomaly detection capabilities into its framework, providing a comprehensive solution for understanding system behavior within microservices.

MicroQA (Wilta and Kistijantoro, 2023) facilitates automatic evaluation of architectural design during the development phase by generating property scores using corresponding metrics with minimal user input dependency. It ensures that new features align with microservice principles without compromising quality. Unlike other tools that focus primarily on analyzing dependencies or monitoring system metrics, MicroQA provides a comprehensive evaluation of system quality, encompassing aspects such as architecture, code quality, and maintainability. By integrating seamlessly into CI/CD workflows, MicroQA facilitates automated quality evaluation, enabling developers to make informed decisions and prevent disruptions to the microservice architecture during development.

GDC-DVF (Graph Deep Clustering based on Dual View Fusion) (Qian et al., 2023) introduces a novel approach to microservice extraction by constructing both structural dependency and business function views of a monolithic application. It uses runtime trace data to generate a class invocation graph and applies random walk algorithms for a comprehensive representation of business functions. The fused feature embedding representations derived from these views are then clustered to propose microservice extractions. Unlike other tools, GDC-DVF employs a graph attention adaptive residual neural network, which enhances the embedding quality and accelerates the convergence speed. This dual-view fusion and deep learning approach distinguishes GDC-DVF

**Table 14**
Summary of various tools and their characteristics — Part (1/4).

| Tool | Dependencies | Tool purpose | Static or Dynamic | Type of validation | Programming languages | Detect or Measure | Year | Input | Ref. |
|---|---|---|---|---|---|---|---|---|---|
| Chains-Former | Control Flow Dependency, Call-Return Dependency | It enhances microservice resource provisioning by analyzing inter-service dependencies and using reinforcement learning to optimize resource allocation. | Dynamic | Academic | Python | Both | 2023 | Traces | Song et al. (2023) |
| TraceNet | Synchronous Dependency, Control Flow Dependency, Call-Return Dependency | It enhances root cause localization in microservice architectures by analyzing service dependencies through trace data, enabling accurate identification of anomalies and their sources. | Dynamic | Academic | Go, C#, Node.js, Python, Java | Both | 2023 | Traces | Yang et al. (2023) |
| GDC-DVF | Control Flow Dependencies, Call-Return Dependency | It facilitates the migration of monolithic applications to microservice architectures by analyzing both structural dependencies and business functionalities to identify cohesive modules suitable for extraction as independent microservices. | Both | Academic | Java | Detect | 2023 | Source Code, Traces | Qian et al. (2023) |
| CMEA | Data Schema Dependencies, Control Flow Dependencies | It aims to decompose monolithic applications into cohesive microservices by analyzing class and method dependencies, as well as partitioning database entities, to enhance system modularity and maintainability. | Static | Academic | Java | Both | 2024 | Source Code | Bajaj et al. (2024) |
| Triple | Runtime Dependencies, Synchronous Dependency, Inter-Microservice Dependency, Inferred Data Dependency, Functional Dependency | It aims to enhance anomaly detection in microservice architectures by integrating and analyzing data from traces, metrics, and logs, providing interpretable insights into system behavior. | Dynamic | Academic | N/A | Detect | 2023 | Traces, Processed Logs, Metrics | Ren et al. (2023) |
| MicroQA | Control Flow Dependencies, Synchronous, Asynchronous Dependencies | It is designed to automate the measurement of microservice architecture quality by performing static and dynamic analyses based on cohesion, coupling, and complexity metrics, integrating seamlessly with CI/CD pipelines. | Dynamic | Industrial, Prototype | Python | Both | 2023 | Source Code, Configuration Files | Wilta and Kistijantoro (2023) |

from other dependency analysis tools, providing a more robust framework for understanding and transforming monolithic architectures into microservices.

FSB-RWRank (Forward, Spin, and Backward Random Walk Algorithm) (Sun et al., 2021) proposes an algorithm aimed at detecting dependencies within microservices. The algorithm constructs a service dependency graph based on conditional independence tests, facilitating a deeper understanding of the software's architecture. By automatically generating checkable dependent paths, FSB-RWRank enables developers to identify critical chains and nodes within the microservice architecture, similar to ChainsFormer. However, unlike ChainsFormer, which analyzes communication-based interactions among microservices, FSB-RWRank utilizes random walk algorithms to generate fault diagnosis paths.

Sakai et al. (2021) focus on extending methods for estimating inter-service dependencies to handle non-deterministic processes and constructing a comprehensive service-graph model based on distributed trace data in microservice architectures. The proposed approach, leveraging an extended Petri net model, is validated through its application to the HipsterShop demo application. This facilitates a detailed understanding of runtime behaviors and enhances architecture recovery by reconstructing the complete service structure.

Genfer and Zdun (2021) focuses on domain-based cyclic dependencies defined as API-level communication chains that form cycles. These dependencies create challenges for deployment and maintenance, particularly in strongly coupled systems. The study proposes a novel static analysis approach using modular, reusable source code detectors. These detectors rely on predefined patterns to identify cyclic dependencies

**Table 15**
Summary of various tools and their characteristics — Part (2/4).

| Tool | Dependencies | Tool purpose | Static or Dynamic | Type of validation | Programming languages | Detect or Measure | Year | Input | Ref. |
|---|---|---|---|---|---|---|---|---|---|
| FSMB-RWRank | Control Flow Dependency, Call-Return Dependency | It aims to identify and rank inter-microservice dependencies by analyzing static and dynamic interactions, enhancing the understanding of service relationships to improve system performance and reliability. | Both | Prototype | N/A | Detect | 2021 | Traces, Runtime Data | Sun et al. (2021) |
| SYM-BIOTE | Synchronous Dependency, Control Flow Dependencies | It is designed to monitor the coupling evolution in microservice-based systems by collecting runtime coupling metrics, enabling the detection of architectural degradation during software evolution. | Dynamic | Industrial | Java | Both | 2021 | Runtime Data | Apolinário and de França (2021) |
| GSMART | Synchronous Dependency, Inter-Microservice Dependencies | It aims to enhance microservice architecture management by automatically generating Service Dependency Graphs (SDGs), enabling developers to analyze dependencies, detect anomalies, and trace service interactions effectively. | Static | Academic, Prototype, Validation | Java | Detect | 2019 | Source Code, Re-quirement Scenarios | Ma et al. (2019a) |
| Deep-Scaler | Synchronous Dependency, Call-Return Dependency, Transaction Dependency | It enhances autoscaling in microservices by accurately detecting and managing inter-service dependencies, thereby optimizing resource allocation and ensuring Service-Level Agreement (SLA) compliance. | Dynamic | Academic | Python | Both | 2023 | Traces, Telemetry Data | Meng et al. (2023) |
| MAFM | Control Flow Dependency, Data Schema Dependency, Synchronous Dependency | It aims to enhance the availability of microservices in container orchestration platforms by detecting unresponsive APIs and initiating corrective actions to restore functionality. | Static | Academic | Java | Detect | 2023 | Source Code | Wang et al. (2023) |
| MAAT | Data Schema Dependencies, Control Flow Dependencies, Synchronous, Asynchronous Dependencies | It evaluates microservice architectures to ensure adherence to principles like loose coupling and high cohesion. | Both | Academic, Prototype | Zipkinb (Java, C#, and Python) | Both | 2018 | Communi-cation Data | Engel et al. (2018) |

within microservice APIs and are suitable for continuous integration pipelines.

Additionally, FSB-RWRank, similar to other tools like SYMBIOTE and TraceNet, contributes to the detection of dependencies by continuously monitoring service coupling metrics and identifying potential architectural degradation. By distinguishing regular increases in coupling from harmful degradation, FSB-RWRank promotes system robustness over time. In addition to the above tools, recent techniques have been introduced to further enhance dependency analysis, Custodio (2016) suggests mapping service dependencies as a Directed Acyclic Graph (DAG) to ensure proper loading order, particularly in C# environments. These techniques complement existing tools by offering practical approaches to monitor and manage service dependencies, ensuring robustness and system reliability.

Collectively, these tools and techniques contribute to improving system understanding, maintainability, and evolution within microservices by addressing various aspects of dependency analysis, resource management, failure diagnosis, and microservice extraction.

*(2) Dependency revealing tools*

These tools aim to uncover hidden or implicit dependencies within software artifacts, helping developers identify overlooked connections and potential sources of issues. By analyzing codebases and repositories, these tools can detect dependencies that may not be explicitly declared or documented, such as implicit function calls or shared resources. By bringing these dependencies to light, developers can address potential sources of bugs or vulnerabilities and ensure the robustness and reliability of the software. Additionally, dependency-revealing tools can aid in enforcing coding standards and best practices by flagging instances of tightly coupled or overly complex dependencies for further review and refactoring.

The Endpoint Dependency Matrix(EDM) and Data Dependency Matrix(DDM) tool (Abdelfattah and Cerny, 2023b) captures dependencies

**Table 16**
Summary of various tools and their characteristics — Part (3/4).

| Tool | Dependencies | Tool purpose | Static or Dynamic | Type of validation | Programming languages | Detect or Measure | Year | Input | Ref. |
|---|---|---|---|---|---|---|---|---|---|
| Endpoint Dependency Matrix (EDM) | Synchronous Dependency | It aims to visualize and analyze inter-microservice dependencies by mapping HTTP endpoint interactions, enhancing understanding of service communications. | Static | Academic, Prototype | SpringBoot | Detect | 2023 | Source Code | Abdelfattah and Cerny (2023b) |
| Data Dependency Matrix (DDM | Data Schema Dependencies | It visualizes data dependencies among microservices by identifying shared data entities, aiding in managing data consistency and integrity. | Static | Academic, Prototype | SpringBoot | Detect | 2023 | Source Code | Abdelfattah and Cerny (2023b) |
| Code Compass | Control Flow Dependency | It is designed to assist developers in comprehending large-scale codebases by providing deep parsing, visualization, and efficient navigation features. | Static | Academic, Industrial | C/C++, JavaScript, Java, TypeScript, CMake | Detect | 2023 | Helm Charts (YAML Files) | Fekete et al. (2023) |
| Go Parser | Source Code Dependency, Control Flow Dependency, Call-Return Dependency | It enables parsing of Go source code into an abstract syntax tree (AST) for programmatic analysis and manipulation. | Static | Academic | Go | Both | 2023 | Source Code | Chen et al. (2023) |
| MCI Tool | Control Flow Dependency, Synchronous Dependency, Asynchronous Dependency | The MCI Tool aims to enhance the management of microservice architectures by identifying and visualizing various dependencies to optimize system performance and maintainability. | Static | Academic | Java | Both | 2023 | Source Code | Zhong et al. (2023) |
| ChainDet | Control Flow Dependency, Synchronous Dependency | It aims to accurately detect and model microservice dependencies by analyzing network traffic patterns, providing valuable insights for network managers. | Dynamic | Academic | N/A | Detect | 2023 | Raw Traffic Data | Zheng et al. (2023) |

within microservice systems by analyzing both endpoints and data entities. Utilizing static analysis, it extracts direct endpoint calls and shared data entities from the source code, thereby revealing the flow of dependencies between endpoints and data entities. This comprehensive approach helps in understanding the interplay between different microservices at a granular level. In contrast, CodeCompass (Fekete et al., 2023) is an open-source framework that uses static analysis to process source code and version control data, offering textual and visual support for code comprehension. Although not primarily focused on dependency detection, it aids in identifying dependencies by making the codebase more understandable, thus indirectly helping to map out dependencies.

The Go Parser Tool (Chen et al., 2023) targets the decomposition of monolithic Go projects into individual components. By analyzing the abstract syntax tree (AST) and dependencies among components, it identifies the necessary dependencies to execute export functions, including imported packages, global variables, and other functions. This method ensures a detailed breakdown of dependencies within monolithic codebases. The MCI Tool (Zhong et al., 2023), on the other hand, measures the degree of dependency and coupling between microservices, offering relative metrics that highlight the extent of interdependencies. By focusing on these relative measurements, the MCI Tool provides valuable insights into the coupling of microservices, which is critical for maintaining a loosely coupled and flexible architecture.

ChainDet (Zheng et al., 2023) is aimed at detecting microservice interactions within specific network areas by analyzing network traffic data to reveal dependencies. It employs algorithms such as Time Series Lag Correlation (TSLC) and Traffic Pattern Discovery (TPD) to model these interactions effectively. This tool's reliance on real-time traffic data makes it adept at uncovering dynamic dependencies that static analysis might miss.

$\mu$Viz (Silva et al., 2021) offers a comprehensive visualization solution for microservice dependencies. It leverages tracing data to extract system morphology at different scales, connecting these views with infrastructure metrics. $\mu$Viz provides four main views: a logical view showing microservices and their dependencies, a physical view displaying microservice instances and their statuses, a traces view decomposing traces by service instances, and a workflow view aggregating system traces to reveal patterns. Utilizing techniques like Flip Zooming and progressive detail enhancement, $\mu$Viz allows users to navigate complex microservice architectures while maintaining context. This integrated approach addresses the challenges of visualizing large-scale, multi-dimensional tracing data, making it easier to understand the system architecture and identify potential issues.

The T-Rank tool proposed by Ye et al. (2021) employs a spectrum-based performance diagnosis approach that dynamically analyzes system performance metrics, particularly focusing on request latency, to localize root causes of performance issues. By analyzing the ranked

**Table 17**
Summary of various tools and their characteristics — Part (4/4).

| Tool | Dependencies | Tool purpose | Static or Dynamic | Type of validation | Programming languages | Detect or Measure | Year | Input | Ref. |
|---|---|---|---|---|---|---|---|---|---|
| $\mu$Viz | Control Flow Dependency, Synchronous Dependency | It visualizes inter-service dependencies, identifying cyclic and transitive relationships, and ensuring adherence to architectural principles like loose coupling and high cohesion. | Dynamic | Academic, Prototype | N/A | Detect | 2021 | Traces | Silva et al. (2021) |
| ID-GCN | Synchronous Dependency, Inter-Microservice Dependencies | It aims to enhance anomaly detection in microservice architectures by constructing an implicit dependency graph and utilizing a Graph Convolutional Network for effective classification. | Dynamic | Academic | N/A | Both | 2023 | Dependency graphs, real-time microservice states | Tang et al. (2023) |
| MicroDep-Graph | Synchronous Dependency, Inter-Microservice Dependencies | It analyzes microservices-based projects to identify and visualize inter-service dependencies, aiding in understanding and managing complex architectures. | Static | Academic, Prototype | Java | Detect | 2019 | Source code, Yaml | Rahman et al. (2019) and Driessen et al. (2023) |
| New Relic | Database Dependencies, Network Dependencies, Synchronous Dependency, Inter-Microservice Dependencies | It provides comprehensive observability into applications and infrastructure, enabling teams to monitor, debug, and improve performance across complex systems. | Dynamic | Industry | Java, .NET, Node.js, PHP, Python, Ruby, Go | Both | 2008 | Traces, Telemetry Data, Logs, Metrics | Ragan (2020) |
| Ortelius | Inter-Microservice Dependency, Data Schema Dependency, Control Flow Dependency, Synchronous Dependency, Asynchronous Dependency, Database Dependency | It simplifies the implementation and management of microservices by providing a central catalog of services with their deployment specifications, enabling application teams to easily consume and deploy services across clusters. | Dynamic | Industry | Java, JavaScript, Python | Both | 2017 | Component metadata, Deployment specifications | Ramesh (2018) |
| T-rank | Inter-Microservice Dependency, Runtime Dependency | T-Rank is designed to efficiently identify the root causes of failures in microservice systems by analyzing execution traces and ranking microservices based on their likelihood of being faulty. | Dynamic | Industry | N/A | Detect | 2021 | Traces | Ye et al. (2021) |

suspicious scores of microservices, T-Rank efficiently detects critical dependencies related to RPC communication. The technique leverages dynamic analysis to account for the evolving nature of service interactions and adapts to changes in the system architecture. T-Rank demonstrates high accuracy and low computational cost, making it a lightweight yet effective solution for dependency diagnosis in microservice systems.

MicroDepGraph (Rahman et al., 2019; Driessen et al., 2023) is a tool designed to analyze and reveal microservice dependencies by examining Docker configurations and service metadata. It constructs a dependency graph using Neo4j, where nodes represent microservices, and edges signify their dependency relationships. The tool also generates an SVG visualization of the dependency graph, providing a clear representation of inter-service dependencies and their structural relationships. By leveraging container metadata, MicroDepGraph uncovers deployment-level dependencies, which are critical for understanding the operational and architectural dynamics of microservice-based systems. This approach is particularly beneficial for managing microservice dependencies in containerized environments, offering a lightweight and automated method to visualize and analyze interactions. By enhancing the visibility of microservice dependencies, MicroDepGraph supports informed architectural decisions, improves maintainability, and facilitates the optimization of complex microservice ecosystems.

New Relic (Relic, 2024, Ragan, 2020) and Ortelius (Community, 2024, Ramesh, 2018) offer complementary approaches to managing and visualizing dependencies in microservice architectures. New Relic provides a Dependencies UI that visualizes upstream and downstream connections among applications, services, databases, and hosts. This tool aids in understanding system architecture, identifying performance bottlenecks, and enhancing monitoring and troubleshooting efforts,

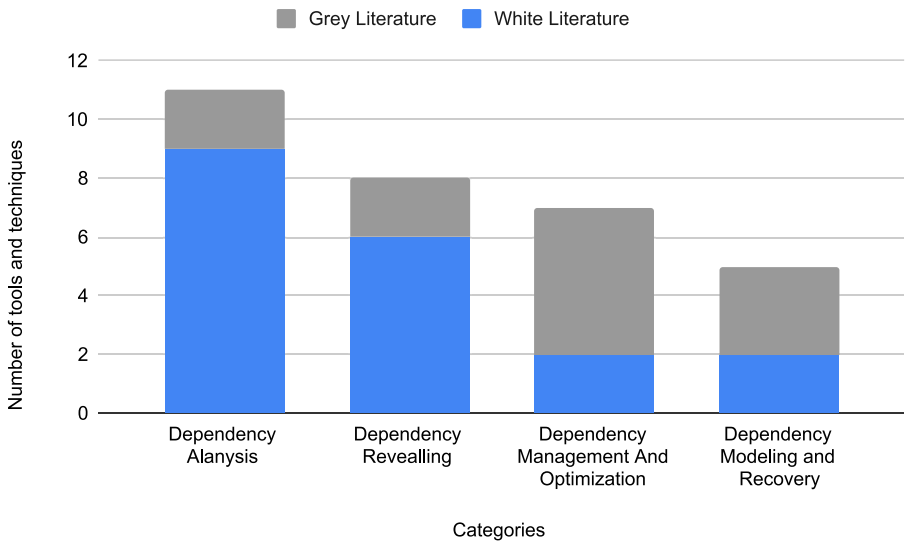


**Fig. 5.** Frequency of dependency related tools and techniques in each category.

with support for multiple programming languages such as Java, .NET, Node.js, PHP, Python, Ruby, and Go. Ortelius, on the other hand, is an open-source platform that focuses on managing deployment, component, and package dependencies. It provides a centralized catalog of services and deployment specifications, mapping dependencies between applications, components, and packages. Ortelius also tracks application versions, maintains dependency histories, and offers unique capabilities like tracking microservice inventories across clusters. Together, these tools enhance the visibility, performance, and maintainability of microservice-based systems, addressing both dependency visualization and version control needs.

Each of these tools has its unique strengths tailored to different aspects of dependency detection and visualization. EDM/DDM and Go Parser tools primarily rely on static analysis of source code, while ChainDet uses network traffic data to uncover dynamic dependencies. The MCI Tool combines static and dynamic data, and $\mu$Viz integrates tracing data with infrastructure metrics for a comprehensive view of system dependencies. Code Compass and $\mu$Viz emphasize visualization to enhance understanding, with Code Compass focusing on code comprehension and $\mu$Viz offering multiple views (logical, physical, traces, workflow) to represent different facets of microservice systems, using advanced techniques like Flip Zooming for context preservation. New Relic further expands dependency visualization by providing a Dependencies UI to map upstream and downstream connections among applications, services, databases, and hosts, aiding in system monitoring and performance optimization. Ortelius complements this by focusing on managing deployment, component, and package dependencies through a centralized catalog of services, tracking versions, and mapping dependencies across clusters. Together, tools like EDM/DDM, Go Parser, ChainDet, MCI Tool, $\mu$Viz, New Relic, and Ortelius provide comprehensive insights into microservice dependencies at various levels, contributing uniquely to the management and optimization of microservice architectures.

In addition to these tools, recent techniques and methodologies further enhance dependency management practices. For example, Siemion (2015) advocates for abstracting database dependencies by exposing a REST API through a dedicated service, thereby eliminating direct access to the database by consumer, reporting, and curation services. This approach enforces service-level dependency on the API, improving modularity, encapsulation, and maintainability. Similarly, vaibhav (2017) introduces the use of scoped npm packages in NodeJS environment to manage shared utilities, avoiding direct file-level dependencies. This technique is complemented by private npm registries, which host reusable components and enable controlled updates, reducing the risks associated with dependency conflicts.

These additions highlight the importance of abstraction, modularity, and controlled dependency management in microservice architectures. By integrating these approaches with existing tools, developers can achieve more scalable, maintainable, and robust systems.

*(3) Dependency management and optimization tools*

These tools ensure the efficient handling of dependencies within software systems. Beyond merely identifying dependencies, these tools offer functionalities such as dependency resolution, version management, and conflict detection. By automating these tasks, developers can streamline the process of managing dependencies and minimize the likelihood of compatibility issues or conflicts arising during development or deployment. Moreover, these tools often provide insights into dependency usage patterns, enabling developers to optimize dependencies for better performance and resource utilization.

ChainsFormer (Song et al., 2023) employs lightweight machine learning techniques to analyze the dynamic nature of microservice inter-dependencies, detecting critical chains and nodes within the system. By leveraging these techniques, ChainsFormer identifies complex dependencies among microservices, facilitating resource provisioning based on reinforcement learning. In contrast, DeepScaler (Meng et al., 2023) utilizes a deep learning-based approach, employing techniques such as expectation maximization-based learning and attention-based graph convolutional networks. DeepScaler adapts to complex service dependencies by dynamically generating affinity matrices and extracting spatio-temporal features of microservices. These advanced techniques enable DeepScaler to accurately quantify dependencies and optimize resource allocation in microservice architectures.

Several techniques have been proposed to ensure smoother dependency management in evolving systems. For instance, backward-compatible APIs are widely recognized as essential for minimizing risks during updates or refactoring (says Reinstate Monica, 2017; Tayade, 2021). This approach allows services to transition seamlessly, ensuring system stability while reducing disruptions. The use of feature flags further complements this strategy by enabling the gradual rollout of changes without breaking dependencies (Tayade, 2021). Additionally, Naros (2015) proposes a "hub" service model to manage data from interdependent services, effectively decoupling core services and fostering a modular architecture.

The paper by Ding et al. (2023) introduces TraceDiag, a framework for root cause analysis (RCA) in large-scale microservice systems. The authors identified service dependency graphs as critical artifacts for

diagnosing root causes of anomalies. The pruning of these graphs to remove redundant components is highlighted as a key step in the analysis process. TraceDiag leverages reinforcement learning (RL) for pruning service dependency graphs, enabling the elimination of redundant components. The framework includes causal-based methods that efficiently analyze pruned graphs.

Other key recommendations for dependency management include interface-first development, staged artifact releases, and adherence to the dependency inversion principle (Sutty1000, 2017). Automation tools are also emphasized as critical for maintaining consistency and efficiency. Furthermore, managing shared dependencies through service-specific implementations, DTO (Data Transfer Object) segregation, and dependency growth monitoring has been highlighted as essential for maintaining architectural consistency (Frost, 2020). These strategies are underpinned by adherence to foundational principles such as DRY (Don't Repeat Yourself) and domain context segregation, which promote clean and maintainable architectures.

These methodologies, coupled with tools such as ChainsFormer and DeepScaler, illustrate the importance of abstraction, modularity, and proactive dependency management in modern microservice architectures. By integrating these strategies and tools, developers can build scalable, resilient, and maintainable systems while mitigating the risks associated with complex dependencies. This comprehensive approach not only optimizes performance but also ensures long-term system stability and adaptability.

*(4) Dependency modeling and recovery tools*

These tools extract, analyze, and visualize dependencies within complex microservice architectures. These tools focus on understanding the relationships between various components, services, and modules within a distributed environment, aiding developers in managing architectural complexity and ensuring system maintainability. By providing insights into intra-service and inter-service dependencies, these tools facilitate architectural decision-making, dependency management, and system optimization. They enable developers to identify dependencies, detect architectural hotspots, and assess the impact of changes on system behavior. Additionally, they support the recovery of architectural models from source code or communication data, offering valuable insights into the structure and dynamics of microservice-based systems.

Microservice Architecture Feature Model (MAFM) (Wang et al., 2023) primarily focuses on the extraction and representation of microservice architecture features from source code. It identifies intra-service and inter-service features, including modules, dependencies, deployment configurations, service interfaces, and invocation relationships. By utilizing a feature representation tree, MAFM facilitates the recovery of microservice architecture information, enabling developers to understand the structural and non-structural aspects of their microservice-based systems. MAFM prioritizes the extraction of architecture-independent code, streamlining the evaluation process and aiding in architectural decision-making.

In contrast, Microservice Architecture Analysis Tool (MAAT) (Engel et al., 2018) is designed as a comprehensive tool for evaluating and visualizing microservice architectures. It provides functionalities for data retrieval, dependency analysis, and metric calculation. MAAT collects communication data from both synchronous and asynchronous interactions between microservices, utilizing the Open Tracing standard for synchronous data retrieval and implementing a custom interface for asynchronous data retrieval. The tool then generates a dependency graph representing the architecture, with nodes representing microservices and edges indicating (a)synchronous dependencies. MAAT offers an interactive dashboard for visualizing the architecture and metric results, enabling developers to identify hotspots, analyze system behavior, and ensure architectural quality.

On the other hand, recent advancements in dependency modeling and recovery emphasize modularization, event-driven architectures, and domain-oriented designs. Villar (2019) proposes a systematic approach to achieve loosely coupled systems, including steps such as mapping and identifying boundaries, decoupling integrations, and initiating modularization efforts. This structured process aids in reducing architectural complexity and enhancing maintainability. Furthermore, structuring microservices around critical business domains, such as payments or inventory management, has been identified as a key strategy for ensuring recoverability and minimizing the business impact of failures (Chase, 2018). Event sourcing (SebS, 2019) has also emerged as a powerful technique for capturing state changes within services, ensuring consistency, and enabling historical data recovery when required.

In comparison, MAFM emphasizes static analysis of source code to extract architectural features, while MAAT focuses on dynamic analysis of communication data to construct dependency graphs and evaluate metrics. Together, these tools address different aspects of dependency modeling and recovery, offering complementary insights into microservice architectures. By integrating these tools with techniques such as modularization, domain-oriented structuring, and event sourcing, developers can achieve a robust framework for managing dependencies, optimizing system behavior, and ensuring architectural resilience in modern microservice environments.

Additionally, Hasselbring and Steinacker (2017) highlight that coupling in monolithic architectures is mitigated in microservice designs through vertical decomposition and granularity adjustments. In microservice design Dependencies are managed using continuous integration and deployment pipelines. While the paper does not focus explicitly on dependency detection tools, it describes automated quality assurance practices integrated into the CI/CD pipelines to monitor and maintain microservice dependencies effectively.

### 5.3. Tool-supported dependency summary

In exploring specific examples of microservice dependencies, several tools, and studies target distinct types. For instance, tools like 'CMEA' focus on *data dependencies*, analyzing how data connections and interactions between business classes and methods impact service interactions and system performance. This analysis is crucial for optimizing data management within microservices. We provided a summary of all tools in Tables 14-17.

*Functional* and *call-return dependencies* are another focus, with tools such as 'TraceNet' and 'GDC-DVF' detecting dependencies that affect system performance and failure diagnostics. 'GDC-DVF' further maps invocation relationships, aiding in the identification and extraction of dependencies, which is vital for decomposing monolithic applications into microservices efficiently.

*Latent service dependencies* are examined by 'ChainsFormer' and 'DeepScaler', which analyze microservice chains and affinity matrices to uncover underlying interactions that affect communication and scalability. For example, 'DeepScaler' utilizes affinity matrices to scale microservices based on service affinities, influencing deployment strategies and performance optimization.

*Endpoint* and *inter-microservice dependencies* are addressed by tools like 'GSMART' and 'MAFM'. 'GSMART' generates a Service Dependency Graph (SDG) to visualize and analyze these dependencies, helping identify potential anomalies and facilitating regression testing. Meanwhile, 'MAFM' uses a Feature Representation Tree to model architectural features, assisting in understanding and refactoring based on *inter-microservice dependencies*.

The identification and analysis of these dependencies play crucial roles beyond simple recognition. They enhance maintainability and performance, assist in failure mitigation, anomaly detection, change impact analysis, adherence to Service Level Agreements (SLAs), Quality of Service (QoS) enhancements, and bottleneck identification. Each type of dependency provides unique insights that aid in optimizing the microservice architecture for improved resilience and efficiency.

In conclusion, the taxonomy of dependencies within microservices is both varied and complex, reflecting the diverse configurations and interconnections possible in distributed systems. Despite some ambiguity in definitions explored in the literature, these dependencies fundamentally represent the interconnectedness essential for the functioning of microservices. This understanding is invaluable for developers and architects in creating robust, scalable, and maintainable microservice architectures.

### *5.4. RQ2.1: Which benchmarks have been used in literature to test and demonstrate dependencies?*

Analyzing dependencies within microservices is crucial for ensuring system performance, scalability, and maintainability. To effectively evaluate these dependencies, benchmarks are employed in academic (white literature) and industrial or technical (grey literature) contexts. Benchmarks provide standardized environments and scenarios to test and demonstrate the impact of dependencies on system behavior and performance. This review explores the various benchmarks used across white and grey literature to assess microservice dependencies, highlighting their significance and applications in real-world scenarios. A summary of these benchmarks is provided in Table 18, and 19.

ChainsFormer (Song et al., 2023) leverages the Train-Ticket benchmark to assess dependencies within microservices, particularly focusing on latency-related dependencies. The benchmark allows ChainsFormer to gather real-world tracing data, construct an execution graph, and apply a weighted longest path algorithm to identify the critical chain with the longest end-to-end latency. This approach specifically targets dependencies related to request processing times between microservices, enabling ChainsFormer to pinpoint critical nodes within the critical chain that significantly influence latency performance. By utilizing the Train-Ticket application as a test bed, ChainsFormer can accurately evaluate and detect latency-based dependencies(service dependencies), which are crucial for optimizing microservice performance.

Tracenet's benchmark (Yang et al., 2023) evaluation relies on datasets A, B, and C from the Hipster-Shop application that contain network-type failures that cause latency issues to assess dependencies at the operation level within microservices. These datasets simulate network-type failures causing latency issues, such as network delays and losses, showcasing real-world scenarios where dependencies between operations impact system performance. Tracenet's three modules, Dependency Constructor, Anomaly Evaluator, and Culprit Locator, work in tandem to detect and analyze anomalies propagated through microservices. The Dependency Constructor constructs a Service Dependency Graph (SDG) at the operation level, capturing intricate dependencies between operations. The Anomaly Evaluator assesses abnormality within SDG edges, considering both inner and outer abnormality to mitigate anomaly propagation. Finally, the Culprit Locator identifies root cause microservices by analyzing anomaly candidates and classifying microservices based on their invocation structures. This comprehensive evaluation using realistic failure scenarios demonstrates Tracenet's ability to detect and analyze operation-level dependencies(functional dependencies) that impact system latency effectively.

The benchmarks in GDC-DVF (Qian et al., 2023), namely Acmeair, Daytrader, Plants, and Jpetstore, were used to test dependencies by analyzing various aspects of their runtime behavior. This included studying the frequency and patterns of function calls, module interactions, and business function dependencies within the monolithic applications. The benchmarks provided a diverse set of scenarios to evaluate how well the proposed approach could identify and extract these dependencies accurately. For example, by examining the runtime traces of method invocations and class interactions in these benchmarks, the tool could detect structural dependencies related to function calls and module interactions, as well as business function dependencies associated with specific application functionalities. This comprehensive analysis enabled the tool to assess its effectiveness in identifying and handling different types of dependencies present in monolithic systems, contributing to the validation of the proposed microservice extraction approach.

The Comprehensive Microservice Extraction Approach (CMEA) (Bajaj et al., 2024) benchmarks, such as JPetStore, AcmeAir, Cargo tracking system, and TFWA, illustrate a diverse range of dependencies essential for microservice architecture. These dependencies span various categories, including data dependencies where method invocations and class interactions within Java codebases play a crucial role. Additionally, the benchmarks showcase interactions between services and their associated data entities, especially within databases like MongoDB, highlighting the importance of service-data interactions. Furthermore, the analysis delves into the ownership and management of specific data entities by microservices, emphasizing the significance of data ownership in microservice design and management. This comprehensive view across the benchmarks underscores the complexity and interplay of dependencies that need to be considered for effective microservice architecture.

The Spinnaker benchmark, integrated with the SYMBIOTE (Apolinário and de França, 2021) method, serves as a case study in analyzing microservice coupling and dependency trends. Spinnaker, a continuous delivery platform for cloud deployment, was used to identify and categorize dependencies within its architecture. These dependencies span service interactions, data dependencies, and architectural patterns, reflecting the complexity of modern software systems. Through SYMBIOTE's framework, the analysis focused on service dependencies like orca, fiat, and Kayenta, highlighting critical service interactions. Metrics such as edge density revealed the level of interconnections, while assessments of service importance and architectural changes provided insights into system evolution and potential degradation. This approach showcases how real-world applications like Spinnaker offer rich insights into dependency management and architectural health.

The GSMART tool (Ma et al., 2019a) conducted an efficiency testing experiment by simulating on a benchmark of 15 microservice-based applications of varying sizes, ranging from 10 to 5000 microservices. These applications were generated synthetically since real-world applications' source code and configurations were not available. The setup included randomly generating service endpoints based on APIs.guru's distribution and allocating service calls according to the Netflix system's distribution. The tool then visualized the Service Dependency Graph (SDG) for each simulated project and recorded computation times. Although the specifics of the dependencies tested were not explicitly mentioned, the experiment aimed to evaluate the performance of the GSMART system in handling microservice projects of different scales, focusing on efficiency in SDG generation.

The DeepScaler tool (Meng et al., 2023) employed three diverse benchmarks – TrainTicket, Online-Boutique, and BookInfo – to comprehensively evaluate its functionality. DeepScaler interacted with each benchmark by simulating dynamic workloads with varying patterns, reflecting the unpredictable nature of microservices in production environments. It utilized performance monitoring to collect tracing and telemetry data, which was then processed using its adaptive learning module. This module trained the resource estimator and learned affinity matrices, revealing time-varying service dependencies within the benchmarks. By predicting future resource demands and allocating resources accordingly, DeepScaler assessed its ability to handle complex dependencies, ensure service-level agreements (SLAs), and optimize resource utilization across different microservice architectures.

The benchmarks in the MAFM (Wang et al., 2023) were used to test and demonstrate dependencies in the context of microservice architecture recovery, specifically focusing on architectural dependencies within software systems. These benchmarks, comprising both open-source and industry projects, were analyzed using tools like MoJoSim and a new recovery technique called DDP, aiming to measure the accuracy of recovered microservice architectures and their ability to help

**Table 18**
Summary of benchmarks and their associated tools — Part (1/2).

| Benchmark link | Tool(s) | Programming languages | Benchmark purpose | Benchmark type | Number of microservices | GitHub collaborators | Stars | Year | Open source | Ref. |
|---|---|---|---|---|---|---|---|---|---|---|
| Acmeair | GDC-DVF, CMEA | Java | Performance evaluation | Academic | Various | N/A | N/A | N/A | Yes | Qian et al. (2023) and Bajaj et al. (2024) |
| ArtifactHub | Code Compass | Multiple | Validation | Industrial, Academic | 288 | 30 | 490 | 2014 | Yes | Fekete et al. (2023) |
| Bigfans-cloud | MAFM | Java | Validation | Academic | N/A | N/A | N/A | N/A | Yes | Wang et al. (2023) |
| BookInfo | GSMART, DeepScaler | N/A | Performance evaluation | Academic | N/A | N/A | N/A | N/A | Yes | Ma et al. (2019a) and Meng et al. (2023) |
| China Mobile | T-Rank | Python | Fault Localization, Performance | Industrial, Academic | Various | N/A | N/A | 2021 | N/A | Ye et al. (2021) |
| CMEA | CMEA | Java | Performance evaluation | Academic | Various | N/A | N/A | Various | Yes | Bajaj et al. (2024) |
| Consul demo | MicroDepGraph | Java | Validation, Evaluation | Academic | 5 | 2 | 11 | 2019 | Yes | Rahman et al. (2019) |
| CQRS microservice application | MicroDepGraph | Java | Validation, Evaluation | Academic | 7 | 2 | 11 | 2019 | Yes | Rahman et al. (2019) |
| Daytrader | GDC-DVF | Java | Performance evaluation | Academic | N/A | N/A | N/A | N/A | Yes | Qian et al. (2023) |
| dddsample-core | CMEA | Java | Validation | Academic | N/A | N/A | N/A | N/A | Yes | Bajaj et al. (2024) |
| E-Commerce App | MicroDepGraph | Java | Validation, Evaluation | Academic | 7 | 2 | 11 | 2019 | Yes | Rahman et al. (2019) |
| Enterprise Planner | MicroDepGraph | Java | Validation, Evaluation | Academic | 5 | 2 | 11 | 2019 | Yes | Rahman et al. (2019) |
| eShopOn - Containers | MicroDepGraph | Java | Validation, Evaluation | Academic | 25 | N/A | N/A | N/A | Yes | Rahman et al. (2019) |
| FTGO - Restaurant Management | MicroDepGraph | Java | Validation, Evaluation | Academic | 13 | N/A | N/A | N/A | Yes | Rahman et al. (2019) |
| Hipster-Shop | TraceNet | Go, C#, Node.js, Python, Java | Performance evaluation | Academic | 10 | N/A | 19 | 2020 | Yes | Yang et al. (2023) |
| jPetStore | GDC-DVF, CMEA | Java | Performance evaluation | Academic | N/A | N/A | N/A | N/A | Yes | Qian et al. (2023) and Bajaj et al. (2024) |
| Lakeside Mutual Insurance Company | MicroDepGraph | Java | Validation, Evaluation | Academic | 8 | 2 | 11 | 2019 | Yes | Rahman et al. (2019) |
| Lelylan-Open Source Internet of Things | MicroDepGraph | Java | Validation, Evaluation | Academic | 14 | N/A | N/A | N/A | Yes | Rahman et al. (2019) |
| Lorca | Go Parser | Go | Validation | Academic | N/A | 24 | 8k | N/A | Yes | Chen et al. (2023) |
| mall-swarm | MAFM | Java | Validation | Academic | N/A | N/A | N/A | N/A | Yes | Wang et al. (2023) |

improve human understanding and speed up comprehension of system dependencies. The paper evaluated dependencies related to service interactions, code entities, and architectural structures, demonstrating the effectiveness of the recovery technique in identifying and representing these dependencies accurately, aiding developers in comprehending the logical structure of complex microservice systems.

The benchmark/application used in the paper to test dependencies for the MAAT tool (Engel et al., 2018) specifically involves an evaluation approach encompassing four main steps. Initially, the required architectural data is retrieved, followed by the construction of an architectural model representing microservice communication dependencies. This model serves as the basis for evaluating the architecture design against identified principles using corresponding metrics. The results are then visualized through a dependency graph with color-coded indicators highlighting adherence to principles. The tool utilizes data retrieval mechanisms from communication logs and application monitoring tools, capturing both synchronous and asynchronous messages to identify communication endpoints as interfaces. Additionally, logical aggregation of microservices into clusters enhances the model's understandability. Overall, the benchmark/application facilitates the

**Table 19**
Summary of benchmarks and their associated tools — Part (2/2).

| Benchmark link | Tool(s) | Programming languages | Benchmark purpose | Benchmark type | Number of microservices | GitHub collaborators | Stars | Year | Open source | Ref. |
|---|---|---|---|---|---|---|---|---|---|---|
| Microservice Architecture for blog post | MicroDepGraph | Java | Validation, Evaluation | Academic | 9 | 2 | 11 | 2019 | Yes | Rahman et al. (2019) |
| Microservices book | MicroDepGraph | Java | Validation, Evaluation | Academic | 6 | 2 | 11 | 2019 | Yes | Rahman et al. (2019) |
| ms-platform | MAFM | Java | Validation | Academic | N/A | N/A | N/A | N/A | Yes | Wang et al. (2023) |
| mogu_blog_v2 | MAFM | Java | Validation | Academic | N/A | N/A | N/A | N/A | Yes | Wang et al. (2023) |
| Online-Boutique | GSMART, DeepScaler | N/A | Validation | Academic | N/A | N/A | N/A | N/A | Yes | Ma et al. (2019a) and Meng et al. (2023) |
| Open-loyalty | MicroDepGraph | Java | Validation, Evaluation | Academic | 5 | N/A | N/A | N/A | Yes | Rahman et al. (2019) |
| Pig | MAFM | Java | Validation | Academic | N/A | N/A | N/A | N/A | Yes | Wang et al. (2023) |
| Pitstop --- Garage Management System | MicroDepGraph | Java | Validation, Evaluation | Academic | 13 | 2 | 11 | 2019 | Yes | Rahman et al. (2019) |
| Robot Shop | MicroDepGraph | Java | Validation, Evaluation | Academic | 12 | N/A | N/A | N/A | Yes | Rahman et al. (2019) |
| Share bike (Chinese) | MicroDepGraph | Java | Validation, Evaluation | Academic | 9 | N/A | N/A | N/A | Yes | Rahman et al. (2019) |
| Spinnaker | SYMBIOTE, MicroDepGraph | N/A | Performance evaluation | Industrial | 10 | N/A | 9.2k | 2015 | Yes | Apolinário and de França (2021) and Rahman et al. (2019) |
| Spring Blade | MAFM | Java | Validation | Academic | N/A | N/A | N/A | N/A | Yes | Wang et al. (2023) |
| Spring Cloud Microservice Example | MicroDepGraph | Java | Validation, Evaluation | Academic | 10 | N/A | 11 | 2019 | Yes | Rahman et al. (2019) |
| Spring PetClinic | MicroDepGraph | Java | Validation, Evaluation | Academic | 8 | 2 | 11 | 2019 | Yes | Rahman et al. (2019) |
| Spring-cloud netflix-example | MicroDepGraph | Java | Validation, Evaluation | Academic | 9 | 2 | 11 | 2019 | Yes | Rahman et al. (2019) |
| SOP | MAFM | Java | Validation | Academic | N/A | N/A | N/A | N/A | Yes | Wang et al. (2023) |
| Tap-And-Eat (Spring Cloud) | MicroDepGraph | Java | Validation, Evaluation | Academic | 5 | 2 | 11 | 2019 | Yes | Rahman et al. (2019) |
| Train-Ticket | ChainsFormer, EDM, DDM, DeepScaler, ChainDet | Python, SpringBoot | Performance evaluation | Academic | 47 | 11 | 642 | 2022 | Yes | Song et al. (2023), Abdelfattah and Cerny (2023b), Meng et al. (2023) and Zheng et al. (2023) |
| Vehicle tracking | MicroDepGraph | Java | Validation, Evaluation | Academic | 8 | 2 | 11 | 2019 | Yes | Rahman et al. (2019) |
| Xueyuan | MAFM | Java | Validation | Academic | N/A | N/A | N/A | N/A | Yes | Wang et al. (2023) |

tool's functionality in automatically assessing microservice architectures based on established principles and metrics, leading to actionable insights for architectural optimization and refinement.

The benchmark used in this Endpoint Dependency Matrix and Data Dependency Matrix (Abdelfattah and Cerny, 2023b) is the train-ticket system, comprising 47 microservices with most of them based on Java using the Spring Boot framework. These microservices communicate through REST API calls and follow enterprise conventions with distinct layers for controllers, services, and repositories. The prototype tool developed for this study utilized static code analysis to extract endpoint and data dependencies within the train-ticket system. For endpoint dependencies, the tool scanned project files for JAX-RS annotations detected HTTP calls between services, and applied signature-matching techniques to establish dependencies. Regarding data dependencies, the tool identified common data entities shared between microservices based on persistence annotations and entity relationships. This approach allowed for a comprehensive analysis of system dependencies, demonstrating the tool's effectiveness in understanding and visualizing

the interconnectedness of endpoints and data dependencies within the microservice architecture of the train-ticket system.

The integration of the new algorithm (Fekete et al., 2023) into the CodeCompass tool for analyzing microservice-based systems involved a range of benchmarks to validate its capabilities in detecting dependencies. The benchmarks included open-source projects from ArtifactHub and microservice-based projects from a multinational telecommunications company. These benchmarks were essential for testing the tool's capabilities across different microservice architectures and complexities. The tool utilized static analysis techniques on Helm charts and Kubernetes configuration files within these benchmarks to detect various dependencies, including service dependencies, config maps, secrets, certificates, and required resource usage such as CPU, memory, and storage.

The benchmark utilized was analyzing the Lorca project involving goParser (Chen et al., 2023) to detect dependencies within the monolithic application. goParser parsed the source code of the Lorca project, identifying imported packages, global variables, and function calls necessary for executing export functions. This analysis was crucial for understanding the structure and interdependencies within the application. The benchmarking process included evaluating different values for the similarity measure (K) using the Ochiai coefficient, calculated based on the detected dependencies. This allowed for optimizing module sizes and determining an appropriate granularity level for the resulting microservices.

The benchmarks utilized for evaluating the Microservice Coupling Index (MCI) tool (Zhong et al., 2023) encompass a diverse range of projects, including Recruita, Enarxb, Cangjinggec, Mall-swarmd, Xaviuse, Madao-servicef, Mall-cloudg, RuoYih, Light-readingi, EPRj, Zscatk, Snowyl, Job-offersm, Rpushn, and Groceryo. These projects were carefully selected to offer a varied representation of microservices across different domains, sizes, and ages. The evaluation aimed to analyze how MCI metrics behave in measuring microservice couplings, addressing questions related to their correlation with existing coupling values, their discriminative power in distinguishing high and low-coupled microservices, and their implications on service independence and change impacts. Through this assessment, the effectiveness and practical utility of MCI metrics in understanding and improving microservice-level coupling were explored and demonstrated. Additionally, the evaluation delved into the association between higher MCI values and reduced independence of microservices, particularly focusing on change impacts during the evolution of microservices.

Rahman et al. (2019) introduced a comprehensive dataset that serves as a benchmark for evaluating microservice dependencies. This dataset comprises 20 open-source projects spanning diverse domains and scales, ranging from small demonstration systems to large-scale industrial applications. Each project is characterized by its use of Java-based microservices deployed via Docker, with inter-service communications and dependencies meticulously mapped using tools such as MicroDepGraph. The dataset enables the analysis of various dependency types, including service-to-service interactions, shared resource dependencies, and architectural patterns. Visualizing these dependencies through formats like GraphML and SVG, the dataset provides a standardized and reproducible environment for studying the complexities of microservice architectures. This benchmark significantly supports both academic research and industrial practices by enabling the assessment of dependency management strategies, scalability challenges, and architectural design decisions in realistic microservice environments, thereby advancing the field of microservices dependency analysis.

The benchmark used for detecting dependencies in the ChainDet (Zheng et al., 2023) tool involved deploying the TrainTicket microservice application. TrainTicket was chosen as the benchmark system due to its standard microservice structure, which provided a suitable environment for evaluating ChainDet's effectiveness in detecting inter-microservice dependencies and microservice chains. The evaluation process simulated various scenarios, including isolated and open-world environments, using the EVE-NG platform. This platform allowed for hardware-in-the-loop virtualization simulations, enabling the deployment of ChainDet on forwarding devices across different domains within the simulated network. The experiment settings included running request scripts to simulate user and admin business queries, covering all functionalities of the TrainTicket application within a specified timeframe. The detection performance of ChainDet was assessed based on its accuracy in identifying inter-microservice dependencies compared to the monitoring results of Skywalking, a control system for intrusive monitoring. Additionally, the completeness of ChainDet's detection of microservice chains was evaluated to gauge its ability to detect complete chains within regular business requests. Through this benchmark setup, the paper demonstrated ChainDet's capability to accurately and comprehensively detect microservice dependencies in realistic microservice applications.

The paper does not specify particular benchmarks or systems employed for evaluating $\mu Viz$. Nevertheless, it details the methodology and functionalities that $\mu Viz$ utilizes to analyze microservice dependencies within a system. By leveraging mechanisms for data collection, techniques for inferring dependencies, analysis of topology, examination of traces, and potential integration with tracing tools, $\mu Viz$ enhances the visualization and understanding of inter-microservice dependencies. Through these methods, $\mu Viz$ aims to provide a comprehensive perspective on system architecture, service interactions, and workflow patterns, thereby enabling users to navigate and analyze complex dependencies in microservice-based applications.

Despite the extensive review of grey literature, no standardized benchmarks were identified for testing and demonstrating microservice dependencies. This gap can be attributed to the inherently informal and practice-driven nature of grey literature, such as blog posts, developer forums, and technical documentation, which often prioritize practical guidance and anecdotal insights over rigorous, reproducible methodologies. Unlike white literature, which relies on well-defined benchmarks to validate tools and approaches, grey literature focuses more on addressing immediate challenges encountered in real-world implementations. Additionally, the lack of standardization in grey literature contributions limits their ability to establish widely accepted benchmarks, as the content often varies significantly in scope, quality, and depth. This observation highlights the disconnect between academic rigor and industrial practices, emphasizing the need for more collaboration to bridge the gap and develop benchmarks that are both scientifically robust and practically applicable.

### 5.5. RQ3: Does the existing literature consider dependencies in addressing software architecture quality attributes?

We have mapped the literature and considered dependencies to various architectural qualities. The most obvious division of the literature approaches is based on whether it considers the dynamic or static system perspective or both, which is rare.

Software architecture quality attributes also align around the static or structural perspective (i.e., maintainability, evolvability, integrability) or the runtime system perspective (i.e., performance, availability, reliability). Oftentimes, the analysis type considered by the approach in literature (static or dynamic) is an excellent indicator of the work focus toward the architecture quality attribute types, but not exclusively.

A very interesting observation comes from approaches that focus on dynamic analysis using logs, traces, and metrics. The dependencies they extract are either composite symptoms with unclear causes, such as correlation between time series, or limited to control dependencies. Works using dynamic analysis consider root cause analysis and anomaly detection and typically aim to improve performance, availability, or occasionally reliability (Zuo et al., 2021; Song et al., 2023; Zhou et al., 2023; Ding et al., 2023; Ma et al., 2019b; Yang et al., 2023; Al Maruf et al., 2022).

**Table 20**
Literature mapping of dependency types to specific software architecture quality attributes.

| Arch. Quality | Dependency category | | | | | |
|---|---|---|---|---|---|---|
| | D1. Data | D2. Control | D3. Resource | D4. Semantic | D5. QA | D6. requirements |
| Performance | Grohmann (2022) | Zuo et al. (2021), Song et al. (2023), Zhou et al. (2023), Ding et al. (2023), Ma et al. (2019b), Yang et al. (2023) and Al Maruf et al. (2022) | | | | |
| Availability | Manish Kumar (2018) | Zuo et al. (2021), Ding et al. (2023), Yang et al. (2023), Al Maruf et al. (2022) and Tsonev (2017) | Chugh and Ahire (2022) | | | |
| Reliability | | Lv et al. (2024) | | | | |
| Deployment | | Zhou et al. (2023) and Lv et al. (2024) | Pai (2023) | | Pai (2023) | Pai (2023), Flater (2021) and Walpita (2020) |
| Maintainability | Daoud et al. (2021), Abdelfattah and Cerny (2023a), Wang et al. (2023), Gortney et al. (2022), Nitin et al. (2023), Abdelfattah and Cerny (2023b) and Engel et al. (2018) | Daoud et al. (2021), Abdelfattah and Cerny (2023a), Silva et al. (2021), Wang et al. (2023), Gortney et al. (2022), Zhong et al. (2023), Genfer and Zdun (2021), Nitin et al. (2023), Abdelfattah and Cerny (2023b), Chen et al. (2023), Engel et al. (2018), Raj and Sadam (2021), Saidi et al. (2023), Wilta and Kistijantoro (2023), Tsonev (2017) and Mangano (2022) | Pai (2023) | Daoud et al. (2021) and Saidi et al. (2023) | Daoud et al. (2021) and Pai (2023) | Speth et al. (2023), Saidi et al. (2023) and Pai (2023) |
| Design (decomposition, modularity) | Daoud et al. (2021), Nitin et al. (2023), Saidi et al. (2023) and Manish Kumar (2018) | Daoud et al. (2021), Nitin et al. (2023), Chen et al. (2023), Raj and Sadam (2021), Saidi et al. (2023), Wilta and Kistijantoro (2023) and Link (2018) | Wang et al. (2023), Link (2018) and Chugh and Ahire (2022) | Daoud et al. (2021) and Saidi et al. (2023) | Daoud et al. (2021) | Flater (2021) and Walpita (2020) |
| Evolution | | Apolinário and de França (2021) | | | | |
| Post-Analysis | Abdelfattah and Cerny (2023a), Wang et al. (2023) and Gortney et al. (2022) | Abdelfattah and Cerny (2023a), Silva et al. (2021), Wang et al. (2023), Gortney et al. (2022), Fekete et al. (2023) and Rahman et al. (2019) | Wang et al. (2023) | | | |

The limited perspective of dynamic analysis on control dependency or composite symptoms is a bit obvious since logs and traces contain limited information with a shared focus on remote calls, events, timing, messages, and possibly other details that indicate symptoms. Control dependency is the most commonly mentioned dependency across the works we identified. It is essential to mention that many works consider the Service Dependency Graph (SDG) to be solely composed of control dependencies. Many works consider SDG as the central artifact to perform root cause analysis and anomaly detection, which should be noted.

Among other architecture quality attributes in the dynamic perspective, works focus on deployment dependencies to optimized deployment (Zhou et al., 2023); however, we classify deployment dependency as a symptom of underlying causes. Moreover, there are cases using control dependencies derived from dynamic analysis to improve evolution, maintenance, and decomposition (Silva et al., 2021; Apolinário and de França, 2021), engage in modeling the system for observability or system post-analysis (Silva et al., 2021). Few works consider visualization aspects (Silva et al., 2021; Ma et al., 2019b; Zuo et al., 2021), which relatively overlaps works considering the temporal perspective.

The static analysis perspective is richer when it comes to dependency type utilization and architectural qualities. There is a great focus on the decomposition of systems and inherently also design (Daoud et al., 2021; Zhong et al., 2023; Nitin et al., 2023; Chen et al., 2023; Saidi et al., 2023), which typically looks at mono-to-micro migration. We can see the control, data, resources, semantics, and requirement dependencies involved here.

System maintainability is perhaps the most significant category (Daoud et al., 2021; Abdelfattah and Cerny, 2023a; Wang et al., 2023; Gortney et al., 2022; Zhong et al., 2023; Genfer and Zdun, 2021; Nitin et al., 2023; Abdelfattah and Cerny, 2023b; Chen et al., 2023; Speth et al., 2023; Engel et al., 2018; Saidi et al., 2023; Wilta and Kistijantoro, 2023). The full spectrum of dependency types can be seen here. While source code is typical input for static analysis, it is also shown to consume policies, YAML files, or change records (Lv et al., 2024; Fekete et al., 2023).

System post-analysis to reason about its qualities or extension is very common, mostly using control and data dependencies (Engel et al., 2018; Abdelfattah and Cerny, 2023a; Wang et al., 2023; Gortney et al., 2022; Fekete et al., 2023; Rahman et al., 2019). Visualization has

also been approached in the static (or hybrid) perspective (Gortney et al., 2022; Abdelfattah and Cerny, 2023b; Ma et al., 2019b; Fekete et al., 2023). These visualizations are typically limited to control dependencies and rarely include data dependencies. The most common intent is to present the user with a holistic system overview to improve observability or aid decision-making.

We plot the complete synthesis in Table 20 that allows tracking dependency type to identify software architecture quality attributes approached by identified literature.

## 6. Discussion

This section provides a detailed overview of the relationships between the outcomes of each stage in the study. It also discusses the challenges highlighted in the identified literature and elaborates on future directions proposed, noting occasional overlaps. Additionally, it summarizes the different perspectives identified in white and grey literature.

### 6.1. Interconnections among dependencies, tools, and benchmarks

This study focuses on dependencies in microservices-based systems, addressing gaps and connections in the existing literature. While the literature offers fragmented information on dependencies, we developed a taxonomy encompassing six dependency categories and four symptom categories, integrating insights from diverse sources. Additionally, we identified tools from the literature designed to detect specific dependencies, many of which are validated using benchmarks for measuring and analyzing these dependencies.

These developed aspects – dependency categories taxonomy, symptoms, tools, and benchmarks – are interconnected and represented cohesively in Fig. 6. The figure provides an overview of these relationships, emphasizing areas with extensive research and investigation, as well as those that remain overlooked in both academic and industrial contexts.

Diving deeper into this figure connections, it reveals that *D2. Control Dependency* and *D3. Resource Dependency* are the most addressed dependency categories in the literature, including multiple dependency types and aspects from the literature. In contrast, the remaining categories in the introduced taxonomy: *D1. Data Dependency*, *D4. Semantic Dependency*, *D5. Quality Attributes Dependency*, and *D6. Requirement Dependency*, they receive limited attention diverse representations in the literature. Moreover, dependencies classified under *S4. Symptoms (Various)* are frequently mentioned in various contexts within the literature, highlighting their prevalence in both academic and industrial environements.

From the tools perspective, control dependencies and their flow, and the synchronous calls dependencies are the most commonly analyzed by various tools. Following this, data dependencies, particularly model and schema-related dependencies, are represented a few times by the various tools.

On the other hand, some tools, such as MicroDepGraph and MAFM, are thoroughly validated using multiple benchmarks. Others, like MicroQA, Ortelius, MCI Tool, and Triple, have relatively moderate validation using a few number of benchmarks. Regarding the benchmarks, the analysis shows that Train-Ticket and jPetStore stand out as the most frequently used benchmarks for validating tools in dependencies, surpassing other identified benchmarks in usage.

### 6.2. Open challenges

Most works can be divided by their focus and perspectives on static and structural system aspects, while others consider the dynamic system perspective, such as performance, root cause analysis, or deployment. However, from both perspectives, they operate with microservice dependencies that overlap.

In the first category, the input is typically a source code or deployment descriptors. Such works consider microservices dependencies when addressing proper modularity of the system whether in mono-to-micro approach (Daoud et al., 2021; Qian et al., 2023; Bajaj et al., 2024; Nitin et al., 2023; Chen et al., 2023; Saidi et al., 2023), SOA-to-micro approach (Raj and Sadam, 2021) or domain-driven decomposition (Driessen et al., 2023). In this context, various works point to a limitation of working with a single language (Nitin et al., 2023; Ma et al., 2019a).

A somewhat reverse perspective of system decomposition is software architecture reconstruction, where microservice dependencies are used to build the system holistic perspective (Abdelfattah and Cerny, 2023a; Engel et al., 2018; Wang et al., 2023). Some works consider dependency extraction as the core goal to help system observability and practitioner understanding (Rahman et al., 2019; Abdelfattah and Cerny, 2023b).

Software architecture reconstruction is often related to visualization of the microservice-based systems (Fekete et al., 2023; Gortney et al., 2022; Ma et al., 2019a; Silva et al., 2021; Abdelfattah and Cerny, 2023a). Visualization approaches use dependencies to infer the connection between microservices.

There are explicit mentions in literature suggesting to identify more dependency types (Zhong et al., 2023) or to differentiate between weak and strong types (Apolinário and de França, 2021) with weights in intermediate models. Various works directly mention plans for future works on more dependency types (Engel et al., 2018; Ma et al., 2019b). A wider variety of dependency types used in the architecture reconstruction can strengthen the architecture model (Wang et al., 2023).

Apart from the single instance at the time, literature considered evolving systems. Change point analysis is mentioned (Apolinário and de França, 2021) aiming to signal alerts indicating architecture degradation. This relates to the identification of anti-patterns (Genfer and Zdun, 2021) and management of dependencies across the evolution (Abdelfattah and Cerny, 2023a).

There is an overlap with the dynamic system perspective category when looking at system evolution considering traces for management of system evolution (Al Maruf et al., 2022). However, in this category, traces or monitoring are the main input where deployment descriptors or code can augment the analysis. For instance, the evolution of traces is the direct interest of anomaly detection (Tang et al., 2023).

However, anomaly detection and root cause analysis is a big problem just for a single version of the system (Zhang et al., 2022; Tang et al., 2023; Yang et al., 2023; Ding et al., 2023; Sakai et al., 2021; Ye et al., 2021). Root cause analysis is used for localization (Zhang et al., 2022). It is the interest of many to look for diverse anomalies (Tang et al., 2023) or to explicitly look for anti-patterns on dependency graphs (Al Maruf et al., 2022). Anomaly detection can identify overload (Song et al., 2022) and help optimize resources given to microservices (Song et al., 2023). Invocation frequency over certain dependencies is something that static analysis cannot identify (Yang et al., 2023; Ye et al., 2021), and it can lead to performance bottlenecks. This analysis is used to verify service layer agreements (Meng et al., 2023), and explicit emphasis is highlighted for dependencies that are not obvious but impact performance.

Apart from root cause analysis, deployment optimization is a challenge for deploying microservices with complex dependencies on the resource-constrained edge servers (Lv et al., 2024). Deployment is another interesting dynamic perspective where changes cause incidents. Studying issues in issue tracking reveals that only a small number of incidents are caused by change with explicit Refs. Batta et al. (2021). Moreover, even if a change is identified as potentially problematic, that information alone has limited value, to make it actionable more reasonable explanation must be given and dependency management can facilitate this Batta et al. (2021).

Some works in the dynamic category explicitly mention the need for better visualization (Zuo et al., 2021; Genfer and Zdun, 2021), ideally those showing dependencies or root cause of the problem (Silva et al., 2021).

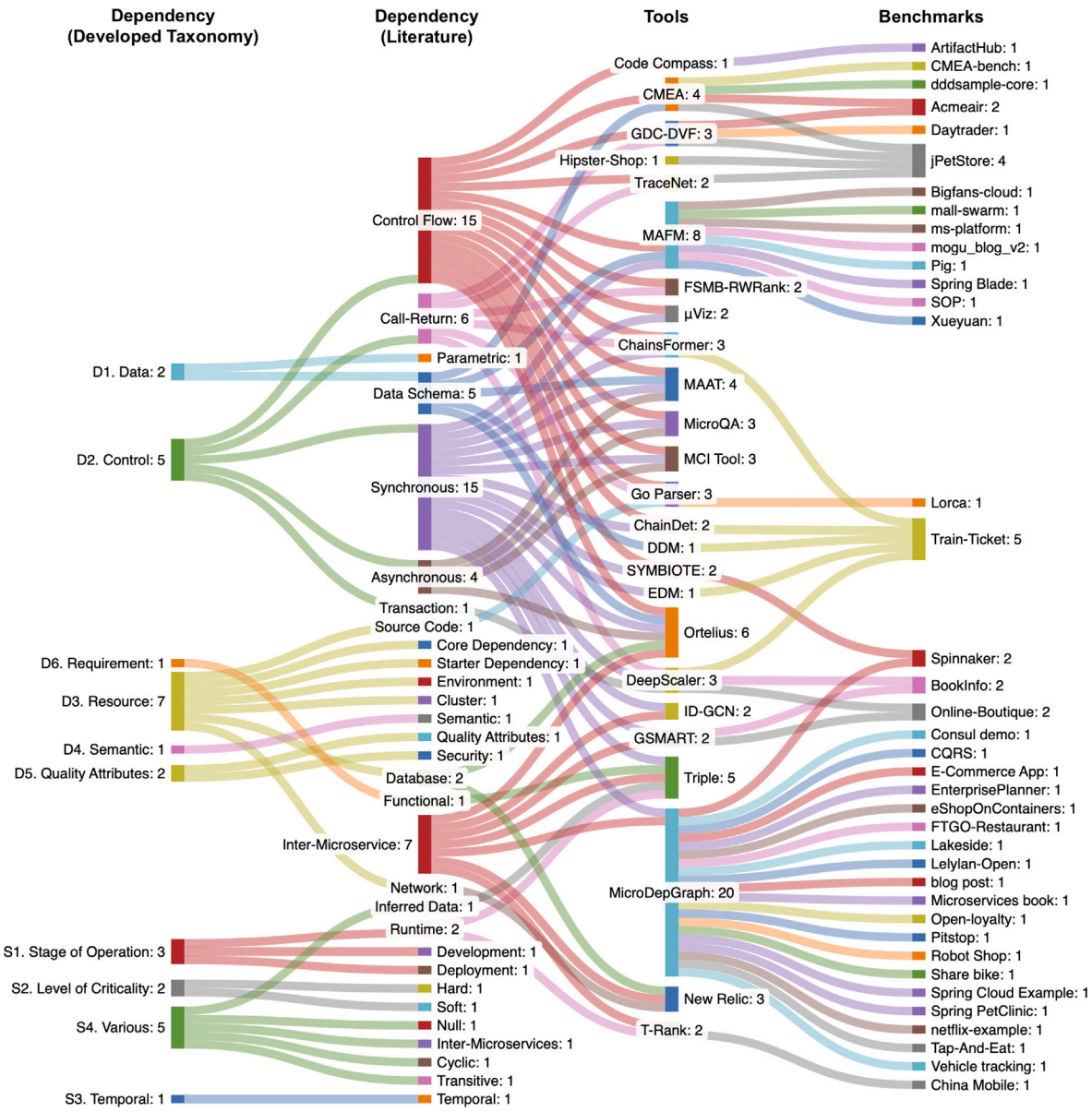


**Fig. 6.** Overview of the dependency taxonomy, dependencies from the literature, tools, and benchmarks relationships. *Note:* The numbers beside each name indicate the count of connections to items on its right column. The original version of the diagram is available at https://zenodo.org/records/14208294.

### 6.3. Implications and future directions

Recognizing different types of microservice dependencies and their aspects should serve as a catalyst for scientific advancements. The wider the spectrum of dependency types we can consider, the smarter the model becomes (Wang et al., 2023; Zhong et al., 2023; Ma et al., 2019b; Engel et al., 2018), with more connection points (Fekete et al., 2023) and the better analysis we can perform for root causes, anti-patterns, architectural degradation, or other tasks. However, the dependencies must be first detected and then managed across the evolution of the system.

Despite the straight focus, we have to deal with different systems developed with different languages and frameworks (Abdelfattah and Cerny, 2023b; Nitin et al., 2023; Genfer and Zdun, 2021; Rahman et al., 2019; Chen et al., 2023) and asynchronous perspectives to microservice interaction (Abdelfattah and Cerny, 2023b; Genfer and Zdun, 2021; Ma et al., 2019b), which is often time ignored in research.

We likely cannot rely on regular expressions (Wilta and Kistijantoro, 2023), and AST parsers need to be used while assuming invocation frequency (Yang et al., 2023) is the factor we can only get from dynamic analysis leading toward hybrid approaches. We do have evidence that explicit dependency tracking is not sufficient (Batta et al., 2021; Meng et al., 2023) in the analysis.

The greater challenge to take on is the direction of system evolution when dependencies act as change propagation avenues and can serve to change impact analysis advancements.

The advancement of predictive analytics and machine learning might be a promising direction to discover other dependency types and help distinguish between symptoms and causes. The integration of machine learning offers a promising path toward proactive dependency management. By developing predictive models tailored to microservice environments, researchers and practitioners could anticipate and mitigate dependency challenges before they affect system performance. Future research should focus on leveraging these models to forecast evolving dependencies and avoid performance bottlenecks, ultimately improving system resilience and operational efficiency.

Visualization needs advancement for real-time analytics, root cause analysis, and change impact analysis. One of the most pressing needs in dependency management is the ability to visualize dependencies.

Existing tools are often limited in their capacity to reflect real-time changes, particularly as systems evolve. By developing adaptive visualization tools incorporating live or changing data feeds, we can enable dependency tracking that provides developers with actionable insights. Such tools could empower teams to respond swiftly to changes, facilitating better-informed decisions and helping to prevent cascading failures in complex, distributed environments.

Proper detection and management of dependencies could facilitate scalability challenges and facilitate root cause identification. For systems undergoing rapid expansion or frequent updates, this seems to be a fundamental need. Many current tools fail to scale effectively within large, evolving microservice ecosystems, necessitating adaptive systems that can handle extensive dependencies while maintaining high performance. Future research should prioritize scalable architectures capable of handling both the depth and breadth of dependencies, with a focus on minimizing manual intervention. Developing systems that dynamically adapt to architectural changes can significantly reduce maintenance overhead and increase system resilience.

Many works we assessed used toy applications for assessment, and empirical validation through industry collaboration is needed in the field. Our review reveals that many dependency management strategies lack broader empirical validation in real-world settings. Practical testing and case studies are essential to determine the real-world viability of proposed solutions. Engaging with industry partners for empirical testing would provide a nuanced understanding of the strengths and limitations of these strategies in operational contexts. Additionally, empirical studies can reveal hidden challenges, offering insights that are crucial for refining theoretical models and enhancing practical applications.

Many researchers in microservices foster interest (or obsession) with anti-patterns and smells; however, could proper management of dependencies turn such research direction obsolete? Many researchers engage in technical debt, but it seems too late to cope with the problem and change impact analysis using dependencies might be the right properly scoped replacement for established practices.

New tools that manage dependencies will influence established development practices, and microservice dependency management calls for robust educational resources and interdisciplinary approaches. By developing training materials that cover dependency management concepts and best practices, we can enhance the skills of practitioners, leading to better-designed, maintainable systems. Moreover, fostering cross-disciplinary collaboration that draws on insights from software engineering, data science, and systems theory could inspire novel solutions to dependency challenges. Uncovering methodologies from grey literature, industry reports, and developer forums may introduce innovative, uncharted approaches to dependency management.

Ultimately, these implications and future directions converge on the need for a cohesive, adaptive ecosystem for microservice dependency management. This ecosystem would combine the predictive power of machine learning with real-time visualization, scalable architectures, and empirically validated strategies to create a comprehensive framework. By pursuing these pathways, we could transform dependency management into an integrated discipline that proactively addresses the complexities of microservice systems. Such a framework would not only align with the dynamic demands of modern software architectures but also provide a foundation for continued innovation in the field. This synthesis of implications and future directions serves as a roadmap for advancing the state of microservice dependency management, encouraging researchers and practitioners alike to push the boundaries of what is currently achievable.

### 6.4. Differences between white and grey literature findings

This study covered both white and grey literature to explore microservice dependencies, systematically combining findings and insights from academic and industrial sources to create a standardized and comprehensive catalog.

The findings revealed notable differences between these two sources. While most dependency categories in the taxonomy appear in both white and grey literature, Semantic Dependency is unique to academic studies. Certain dependency types, such as Parametric Dependency (in Data Dependency) and Call-Return and Transaction Dependencies (in Control Dependency), are absent in grey literature. Conversely, Asynchronous Dependency, Source Code Dependency, and Database Dependency are more common in industrial contexts. Industry-focused grey literature also emphasizes multiple symptoms as causes of dependencies, whereas academic papers contribute minimally to this aspect. This difference likely stems from the academic focus on precise terminology to describe theories and events, whereas industrial work prioritizes practical solutions. In industry, an issue may simply be labeled as a "dependency", while academic studies take a more cautious and deliberate approach in naming it as such. That said, some symptoms are also identified as causes of dependencies in academic studies. For instance, Inter-Microservice Dependency is consistently recognized in both white and grey contexts, often with implications that align closely with Control Dependency; However, we classified it as a symptom since it could represent data, resource, or other types of dependencies between microservices.

Furthermore, academic literature leads in introducing tools for analyzing microservice dependencies, with almost all tools stemming from white literature, except for two from grey literature. These industry-driven tools mainly address Control and Data Dependencies but highlight the lack of open-source industrial tools. Academic tools, while innovative, are often prototypes and not production-ready. Benchmarks also show a significant academic focus, with a notable absence of real-life systems from industry, most probably that is due to proprietary concerns from indutrial organizations.

Regarding quality attributes, grey literature focuses on practical qualities such as availability, maintainability, deployment, and system modularity, reflecting the immediate priorities of industry practitioners. Nevertheless, academic studies address a broader range including these attributes, and also connect dependencies to other quality attributes like performance, reliability, and post-analysis.

In summary, while academic and industrial sources address overlapping themes, their focus areas diverge. Academia emphasizes breadth, theoretical insights, and tool development, whereas industry highlights practicality, symptoms, and specific subset of quality attributes. These differences provide a balanced perspective, enhancing the depth and value of the study and its findings.

## 7. Threats to validity

This section delineates potential threats to the validity of our systematic literature review, as per the classification suggested by Ampatzoglou et al. (2019). We focus on three primary areas: study selection validity, data validity, and research validity, addressing the measures taken to mitigate these risks.

### 7.1. Study selection validity

The validity of our study selection process is critical to ensure that no relevant studies are overlooked. To mitigate the risk of exclusion, we designed a comprehensive search strategy that included multiple databases to cover a broad spectrum of literature on microservice dependencies. We further employed backward snowballing techniques to capture additional studies from the references of selected articles. Despite these precautions, there is an inherent risk of missing studies that may use non-standard terminology or are published outside the indexed journals. To counterbalance this, the initial pool of studies was screened independently by multiple team members, followed by a consensus meeting to resolve any disagreements, ensuring a robust selection process.

**Table 21**
White literature articles — Part (1/2).

| Title | Year | Authors | Reference |
|---|---|---|---|
| Graph-based and scenario-driven microservice analysis, retrieval, and testing | 2019 | Shang-Pin Ma, Chen-Yuan Fan, Yen Chuang, I-Hsiu Liu, Ci-Wei Lan | Ma et al. (2019a) |
| Microservice Architectures for Scalability, Agility and Reliability in E-Commerce | 2017 | Hasselbring, Wilhelm, Steinacker, Guido | Hasselbring and Steinacker (2017) |
| Graph-Reinforcement-Learning-Based Dependency-Aware Microservice Deployment in Edge Computing | 2024 | Lv, Wenkai, Yang, Pengfei, Zheng, Tianyang, Lin, Chengmin, Wang, Zhenyi, Deng, Minwen, Wang, Quan | Lv et al. (2024) |
| Dependency-aware Microservice Deployment and Resource Allocation in Distributed Edge Networks | 2023 | Zhou, Jizhe, Wang, Guangchao, Zhou, Wei | Zhou et al. (2023) |
| The Microservice Dependency Matrix | 2023 | Abdelfattah, Amr S. and Cerny, Tomas | Abdelfattah and Cerny (2023b) |
| An Automatic Scaling System for Online Application with Microservice Architecture | 2022 | Song, Youmei, Li, Chaoran, Zhuang, Kuoran, Ma, Tianjiao, Wo, Tianyu | Song et al. (2022) |
| ChainsFormer: A Chain Latency-Aware Resource Provisioning Approach for and Microservices Cluster | 2023 | Song, Chenghao, Xu, Minxian, Ye, Kejiang, Wu, Huaming, Gill, Sukhpal Singh, Buyya, Rajkumar, Xu, Chengzhong | Song et al. (2023) |
| DeepScaler: Holistic Autoscaling for Microservices Based on Spatiotemporal GNN with Adaptive Graph Learning | 2023 | C. Meng, S. Song, H. Tong, M. Pan, Y. Yu | Meng et al. (2023) |
| Microservice extraction using graph deep clustering based on dual view fusion | 2023 | Lifeng Qian, Jing Li, Xudong He, Rongbin Gu, Jiawei Shao, Yuqi Lu | Qian et al. (2023) |
| TraceNet: Operation Aware Root Cause Localization of Microservice System Anomalies | 2023 | Yang, Jingjing, Guo, Yuchun, Chen, Yishuai, Zhao, Yongxiang | Yang et al. (2023) |
| A Comprehensive Microservice Extraction Approach Integrating Business Functions and Database Entities | 2024 | Deepali Bajaj, Anita Goel, Suresh Gupta | Bajaj et al. (2024) |
| CARGO: AI-Guided Dependency Analysis for Migrating Monolithic Applications to Microservice Architecture | 2023 | Nitin, Vikram, Asthana, Shubhi, Ray, Baishakhi, Krishna, Rahul | Nitin et al. (2023) |
| From a Monolith to a Microservice Architecture Based Dependencies | 2023 | Saidi, Malak and Tissaoui, Anis and Faiz, Sami | Saidi et al. (2023) |
| Evaluation of microservice architectures: A metric and tool-based approach | 2018 | Engel, Thomas, Langermeier, Melanie, Bauer, Bernhard, Hofmann, Alexander | Engel et al. (2018) |

### 7.2. Data validity

Ensuring the accuracy of data extracted from the studies is paramount. To this end, each study was subjected to a detailed examination by two independent reviewers using a predefined data extraction template aligned with our research questions. This dual-review approach helps to minimize biases and errors in data collection. Any discrepancies between reviewers were resolved through discussion, involving a third reviewer when necessary to reach a consensus. This methodological rigor enhances the reliability of our data, although subjective judgments in interpreting study findings could introduce variability.

### 7.3. Data synthesis validity

The trustworthiness of the data synthesis in this study was established by adhering to the Credibility, Confirmability, Dependability, and Transferability criteria (Cruzes and Dyba, 2011).

*Credibility* is addressed through validating the accuracy of the data extraction and analysis that was ensured by involving multiple authors who reviewed and validated the process individually. This included using representative text segments like dependency types and dependency artifacts from the literature to support coding decisions and achieving consensus on the taxonomy categories. The *Confirmability* is maintained by employing systematic approaches of thematic analysis, such as open-coding and axial coding, minimizing personal biases. The coding outcomes were cross-verified with the findings of primary studies to ensure alignment and consistency. Regarding *Dependability*, the stability of the synthesis process was reinforced by documenting all changes during the iterative stages of data categorization and taxonomy construction. An audit trail was maintained to provide transparency and reproducibility of the analysis steps. The process details are included in the reproducibility package. For *Transferability*, the findings were structured to be applicable in broader contexts by providing detailed descriptions of the selected studies, the coding techniques employed, and the resulting taxonomy. With the publicly available reproducibility package, this enables other researchers to understand, replicate, or adapt the approach for their own domains.

### 7.4. Research validity

Research validity concerns the extent to which the findings of the review are generalizable and reproducible. Our systematic approach, characterized by clear inclusion and exclusion criteria and a transparent, detailed documentation of the review process, supports the generalizability of our findings within the scope of microservice dependencies. The reproducibility of our study is facilitated by the comprehensive description of our methods in Section 3, allowing other researchers to replicate our work. Nevertheless, the dynamic nature of software engineering practices, particularly in an evolving field like microservices, may affect the long-term applicability of our results.

## 8. Conclusions and future work

This systematic multivocal literature review has analyzed the dependencies within microservices, revealing the intricacies of managing these dependencies effectively to maintain scalable and resilient systems. Our review started with an initial selection of 1733 papers, narrowed down through a meticulous filtering process to include 45 significant publications. The study synthesized information from both white and grey literature sources, providing a comprehensive overview of dependency types, challenges, and management strategies.

We uncovered an array of dependency types and their potential impacts on microservice architectures. These dependencies play a crucial role in system qualities like performance and scalability, influencing architectural decisions and operational strategies. Despite the diverse strategies and tools available, managing these dependencies remains a challenge due to their dynamic and complex nature. This review has highlighted the need for more adaptive and intelligent management systems that can anticipate and mitigate the risks associated with these dependencies.

The study also underscored the absence of a unified framework for categorizing and managing these dependencies effectively. This gap in the literature has been used as an opportunity to develop a comprehensive taxonomy of microservice dependencies in this manuscript to aid in better understanding and managing these complex relationships.

For future work, we aim to explore the development of advanced tools that manage dependencies in real-time. Such tools would not

**Table 22**
White literature articles — Part (2/2).

| Title | Year | Authors | Reference |
|---|---|---|---|
| Microservice architecture recovery based on intra-service and inter-service features | 2023 | Wang, Lulu, Hu, Peng, Kong, Xianglong, Ouyang, Wenjie, Li, Bixin, Xu, Haixin, Shao, Tao | Wang et al. (2023) |
| A multi-model based microservices identification approach | 2021 | Daoud, Mohamed, El Mezouari, Asmae, Faci, Noura, Benslimane, Djamal, Maamar, Zakaria, El Fazziki, Aziz | Daoud et al. (2021) |
| ChainDet: A Traffic-Based Detection Method of Microservice Chains | 2023 | Zheng, Chunyang, Wang, Jinfa, Si, Shuaizong, Zhang, Weidong, Sun, Limin | Zheng et al. (2023) |
| On measuring coupling between microservices | 2023 | Chenxing Zhong, He Zhang, Chao Li, Huang Huang, Daniel Feitosa | Zhong et al. (2023) |
| Automatic Dependency Tracking in Microservice-based Systems Using Static Analysis in Helm Charts | 2023 | Fekete, Anett, Kovács, Benedek, Porkoláb, Zoltán | Fekete et al. (2023) |
| MAT: Automating Go monolithic applications transform into microservices through dependency analysis and AST | 2023 | Chen, Yi-Yuan, Hsu, Kuo-Hsun, Hou, Andrew Weian | Chen et al. (2023) |
| Version-based microservice analysis, monitoring, and visualization | 2019 | Ma, Shang-Pin, Liu, I-Hsiu, Chen, Chun-Yu, Lin, Jiun-Ting, Hsueh, Nien-Lin | Ma et al. (2019b) |
| $\mu$Viz: Visualization of Microservices | 2021 | Silva, Sara, Correia, Jaime, Bento, Andre, Araujo, Filipe, Barbosa, Raul | Silva et al. (2021) |
| Automatic Measurement of Microservice Architecture Quality with Cohesion, Coupling, and Complexity Metrics | 2023 | Wilta, Alvin, Kistijantoro, Achmad Imam | Wilta and Kistijantoro (2023) |
| Fault Root Rank Algorithm Based on Random Walk Mechanism in Fault Knowledge Graph | 2021 | Sun, Yindong, Zhao, Longjun, Wang, Zhen, Cui, Dandan, Yang, Yang, Gao, Zhipeng | Sun et al. (2021) |
| Integrating issue management systems of independently developed software components | 2023 | Speth, Sandro, Breitenbücher, Uwe, Krieger, Niklas, Wippermann, Pia, Becker, Steffen | Speth et al. (2023) |
| Temporal relations extraction and analysis of log events for micro-service framework | 2021 | Zuo, Yuan, Zhu, Xiaozhou, Qin, Jiangyi, Yao, Wen | Zuo et al. (2021) |
| Triple: The Interpretable Deep Learning Anomaly Detection Framework based on Trace-Metric-Log of Microservice | 2023 | Ren, Rui, Wang, Yang, Liu, Fengrui, Li, Zhenyu, Xie, Gaogang | Ren et al. (2023) |
| T-rank: A lightweight spectrum-based fault localization approach for microservice systems | 2021 | Ye, Zihao, Chen, Pengfei, Yu, Guangba | Ye et al. (2021) |
| Roadmap to reasoning in microservice systems: a rapid review | 2023 | Abdelfattah, Amr S, Cerny, Tomas | Abdelfattah and Cerny (2023a) |
| Method of constructing Petri net service model using distributed trace data of microservices | 2021 | Sakai, Masaru, Takahashi, Kensuke, Kondoh, Satoshi | Sakai et al. (2021) |
| Visualizing microservice architecture in the dynamic perspective: A systematic mapping study | 2022 | Gortney, Mia E, Harris, Patrick E, Cerny, Tomas, Al Maruf, Abdullah, Bures, Miroslav, Taibi, Davide, Tisnovsky, Pavel | Gortney et al. (2022) |
| Identifying domain-based cyclic dependencies in microservice apis using source code detectors | 2021 | Genfer, Patric, Zdun, Uwe | Genfer and Zdun (2021) |
| TraceDiag: Adaptive, Interpretable, and Efficient Root Cause Analysis on Large-Scale Microservice Systems | 2023 | Ding, Ruomeng, Zhang, Chaoyun, Wang, Lu, Xu, Yong, Ma, Minghua, Wu, Xiaomin, Zhang, Meng, Chen, Qingjun, Gao, Xin, Gao, Xuedong, others | Ding et al. (2023) |
| A system for proactive risk assessment of application changes in cloud operations | 2021 | Batta, Raghav, Shwartz, Larisa, Nidd, Michael, Azad, Amar Prakash, Kumar, Harshit | Batta et al. (2021) |
| Benchmarks for end-to-end microservices testing | 2023 | Smith, Sheldon, Robinson, Ethan, Frederiksen, Timmy, Stevens, Trae, Cerny, Tomas, Bures, Miroslav, Taibi, Davide | Smith et al. (2023) |
| Empirical Evaluation of Microservice Architecture | 2022 | Kaushik, Neha, Kumar, Harish, Raj, Vinay | Kaushik et al. (2022) |
| Patterns for migration of SOA based applications to microservice architecture | 2021 | Raj, Vinay, Sadam, Ravichandra | Raj and Sadam (2021) |
| A curated dataset of microservices-based systems | 2019 | Rahman, Mohammad Imranur, Panichella, Sebastiano, Taibi, Davide | Rahman et al. (2019) |
| Using microservice telemetry data for system dynamic analysis | 2022 | Al Maruf, Abdullah, Bakhtin, Alexander, Cerny, Tomas, Taibi, Davide | Al Maruf et al. (2022) |
| Microservice Anomaly Diagnosis with Graph Convolution Network Based on Implicit Microservice Dependency | 2023 | Tang, Hao, Guo, Yuchun, Yang, Jingjing, Chen, Yishuai | Tang et al. (2023) |
| Fault localization for microservice applications with system logs and monitoring metrics | 2022 | Zhang, Qixun, Jia, Tong, Wu, Zhonghai, Wu, Qingxin, Jia, Lichun, Li, Donglei, Tao, Yuqing, Xiao, Yutong | Zhang et al. (2022) |
| A quantitative assessment method for microservices granularity to improve maintainability | 2023 | Driessen, Famke, Ferreira Pires, Luís, Moreira, Joao Luiz Rebelo, Verhoeven, Paul, van den Bosch, Sander | Driessen et al. (2023) |
| Model Learning for Performance Prediction of Cloud-native Microservice Applications | 2022 | Grohmann, Johannes Sebastian | Grohmann (2022) |
| Analytically-Driven Resource Management for Cloud-Native Microservices | 2024 | Zhang, Yanqi, Zhou, Zhuangzhuang, Elnikety, Sameh, Delimitrou, Christina | Zhang et al. (2024) |
| A method for monitoring the coupling evolution of microservice-based architectures | 2021 | Daniel R.F. Apolinário and Breno B.N. de França | Apolinário and de França (2021) |

only address the current limitations of existing dependency management approaches but also enhance the adaptability and efficiency of microservice architectures. Additionally, there is a need for empirical studies that can validate the proposed strategies and tools in real-world scenarios, ensuring their effectiveness and robustness.

Another important direction for future research is the development of improved change impact analysis methodologies. Changes made to one service often have unintended consequences on other dependent services, leading to unpredictable behavior or failures. Enhanced change impact analysis tools would help in identifying and mitigating such risks before deploying updates, reducing the fragility of interconnected microservices. Developing automated mechanisms for assessing change impact and enforcing compatibility during updates is crucial for maintaining the stability of dynamic service-oriented systems.

**Table 23**
Grey literature articles.

| Title | Year | Authors | Reference |
|---|---|---|---|
| Context mapping — relations | 2016 | theDmi | theDmi (2016) |
| How to handle dependencies between microservices all called within one large service | 2023 | Greg Burghardt | Burghardt (2023) |
| Autonomous Microservices, event queues and service discovery | 2016 | Joeri Sebrechts | Sebrechts (2016) |
| What are the potential problems with operational circular dependency between microservices | 2017 | gsf | gsf (2017) |
| Data from one microservice to another | 2020 | Martin K | Martin (2020) |
| How to manage event dependencies in event-driven architectures? | 2019 | Robert Bräutigam | Bräutigam (2019) |
| Designing java project for monoliths and microservices at same time | 2017 | Tom Van Rossom | Rossom (2017) |
| Microservices inter-communication when response is needed | 2021 | JonasH | JonasH (2021) |
| Dependency hell when using same npm module for both sdk and service itself | 2023 | Yaniv Shiloah | Shiloah (2023) |
| 5 Tips for Managing Service Dependencies in a Microservice Architecture | 2023 | Razorops, Inc. | Razorops, Inc. (2023) |
| Microservices: Tracking Dependencies | 2022 | Akshay Chugh, Jayesh Bapu Ahire | Chugh and Ahire (2022) |
| Circular Dependencies Between Microservices | 2022 | Leonardo Mangano | Mangano (2022) |
| Change Coupling: Visualize the Cost of Change | 2023 | CodeScene Engineering Team | CodeScene Engineering Team (2023) |
| Microservice Architecture Essentials: Loose Coupling | 2023 | Chris Richardson | Richardson (2023) |
| Coupling and Cohesion in Microservices | 2020 | Priyal Walpita | Walpita (2020) |
| Microservices: Design-Time Coupling | 2021 | Chris Richardson | Richardson (2021) |
| Cyclic dependencies in microservices | 2019 | James Youngman | MicroMaster (2019) |
| Microservice should be an independent software unit — Up to which level? | 2020 | Euphoric | Euphoric (2020) |
| Data Dependency Among Microservices | 2021 | KitKarson | KitKarson (2021) |
| Clean Architecture Design Pattern | 2018 | René Link | Link (2018) |
| What is the Specific difference between Java Modules (In Java 9), OSGi bundles and Microservices? | 2018 | Peter Kriens | Kriens (2018) |
| Microservices Why Use RabbitMQ? | 2017 | Dimitar Tsonev | Tsonev (2017) |
| Adding new service into existing micro-service system | 2021 | Flater | Flater (2021) |
| Springboot: difference between responsibilities of dependencies grouped as STARTER vs CORE | 2020 | Aditya Rewari | Rewari (2020) |
| Resolving design dependency between microservices | 2018 | Manish Kumar | Manish Kumar (2018) |
| How to send notification from microservice to asp.net core's SignalRHub | 2019 | Syed Mohammad Fahim Abrar | Abrar (2019) |
| Dependency Hell in Microservices and How to Avoid It | 2016 | Nabil Hijazi | Hijazi (2016) |
| Services dependencies and relations documentation | 2022 | Majid Abdolhosseini | Abdolhosseini (2022) |
| .NET Microservices with CQRS handling dependencies | 2018 | down_town_the | downtownthe (2018) |
| Isolating and Managing Dependencies in 12-Factor Microservice Applications with Kubernetes | 2023 | Santosh Pai | Pai (2023) |
| In a loosely coupled microservice architecture, how do you keep track of your dependencies? | 2017 | David says Reinstate Monica | says Reinstate Monica (2017) |
| microservices bounded context in distributed analytics | 2015 | Adam Siemion | Siemion (2015) |
| How to slice down a monolith where my domain depends heavily from other domains? | 2019 | Vitor Villar | Villar (2019) |
| Dockering a nodejs application with external dependencies | 2017 | Vaibhav, Galkin | vaibhav (2017) |
| How to decompose Monolithic applications into Microservices with interdependent functional modules? | 2015 | Naros | Naros (2015) |
| How to manage dependencies for microservices? | 2017 | Sutty1000 | Sutty1000 (2017) |
| How do you determine what should be a microservice? | 2018 | Dan Chase | Chase (2018) |
| Sharing DTO objects between microservices | 2020 | Daniel Frost | Frost (2020) |
| Best practice for loading multiple dynamic services and them dependencies services | 2016 | Thiago Custodio | Custodio (2016) |
| How to decouple data between microservices? | 2019 | SebS | SebS (2019) |
| How to avoid microservice dependency without slowing down your release process | 2021 | Amit Tayade | Tayade (2021) |
| Microservice dependency manager tools | 2020 | Tracy Ragan | Ragan (2020) |
| Microservice dependency manager tools | 2018 | Aarish Ramesh | Ramesh (2018) |
| Parent POM and Microservices | 2015 | Jigar Shah | Shah (2015) |
| Data Dependency Among Microservices | 2015 | KitKarson | KitKarson (2021) |

The complexities surrounding the maintainability of microservices, especially in highly dynamic environments, also present a significant avenue for future exploration. Effective maintainability is often hindered by the intricate web of dependencies, leading to challenges when scaling services, integrating new features, or refactoring components. Research into frameworks and strategies that provide a clearer view of dependencies and facilitate manageable service evolution is essential for reducing technical debt and long-term costs.

By extending our understanding of microservice dependencies and improving the tools available to manage them, this research contributes to the broader field of software engineering, particularly in enhancing the design and maintenance of distributed systems. Through continued exploration and technological innovation, we can better harness the potential of microservices, ensuring they deliver on their promise of scalability and resilience in the face of evolving software demands.

## CRediT authorship contribution statement

**Amr S. Abdelfattah:** Writing – original draft, Supervision, Methodology, Investigation, Data curation. **Tomas Cerny:** Writing – original draft, Validation, Supervision, Resources, Project administration, Methodology, Investigation, Funding acquisition, Data curation, Conceptualization. **Md Showkat Hossain Chy:** Writing – original draft, Visualization, Validation, Supervision, Resources, Project administration,

Methodology, Investigation, Data curation. **Md Arfan Uddin:** Writing – original draft, Methodology, Investigation, Data curation, Conceptualization. **Samantha Perry:** Writing – original draft, Methodology, Investigation, Data curation, Conceptualization. **Cameron Brown:** Writing – original draft, Methodology, Investigation, Data curation, Conceptualization. **Lauren Goodrich:** Writing – original draft, Methodology, Investigation, Data curation, Conceptualization. **Miguel Hurtado:** Writing – original draft, Methodology, Investigation, Data curation, Conceptualization. **Muhid Hassan:** Writing – original draft, Methodology, Investigation, Data curation, Conceptualization. **Yuanfang Cai:** Writing – review & editing, Validation, Resources, Funding acquisition, Conceptualization. **Rick Kazman:** Writing – review & editing, Validation, Resources, Funding acquisition.

## Funding

This material is based upon work supported by the National Science Foundation, United States under Grant No. 2409933, Grant No. 2236824, Grant No. 2232720 and Grant No. 2232721.

## Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Tomas Cerny reports financial support was provided by National Science Foundation. Rick Kazman reports financial support was provided by National Science Foundation. Yuanfang Cai reports financial support was provided by National Science Foundation. 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.

## Data availability

The data supporting this study is openly available at https://zenodo.org/records/14208294.

## References

Abgaz, Yalemisew, McCarren, Andrew, Elger, Peter, Solan, David, Lapuz, Neil, Bivol, Marin, Jackson, Glenn, Yilmaz, Murat, Buckley, Jim, Clarke, Paul, 2023. Decomposition of monolith applications into microservices architectures: A systematic review. IEEE Trans. Softw. Eng..

Ampatzoglou, Apostolos, Bibi, Stamatia, Avgeriou, Paris, Verbeek, Marijn, Chatzigeorgiou, Alexander, 2019. Identifying, categorizing and mitigating threats to validity in software engineering secondary studies. Inf. Softw. Technol. (ISSN: 0950-5849) 106, 201–230. http://dx.doi.org/10.1016/j.infsof.2018.10.006, URL https://www.sciencedirect.com/science/article/pii/S0950584918302106.

Bushong, Vincent, Abdelfattah, Amr S, Maruf, Abdullah A, Das, Dipta, Lehman, Austin, Jaroszewski, Eric, Coffey, Michael, Cerny, Tomas, Frajtak, Karel, Tisnovsky, Pavel, et al., 2021. On microservice analysis and architecture evolution: A systematic mapping study. Appl. Sci. 11 (17), 7856.

Cerny, Tomas, Chy, Md Showkat Hossain, Abdelfattah, Amr, Soldani, Jacopo, Bogner, Justus, 2024. On Maintainability and Microservice Dependencies: How Do Changes Propagate? pp. 277–286. http://dx.doi.org/10.5220/0012725200003711.

Community, Ortelius, 2024. Ortelius: Open source microservice management platform. https://github.com/ortelius/ortelius. (Accessed 21 November 2024).

Cruzes, Daniela S., Dyba, Tore, 2011. Recommended steps for thematic synthesis in software engineering. In: 2011 International Symposium on Empirical Software Engineering and Measurement. IEEE, pp. 275–284.

Ding, Xiang, Zhang, Cheng, 2022. How can we cope with the impact of microservice architecture smells? In: Proceedings of the 2022 11th International Conference on Software and Computer Applications. pp. 8–14.

Dragoni, Nicola, Giallorenzo, Saverio, Lluch-Lafuente, Alberto, Mazzara, Manuel, Montesi, Fabrizio, Mustafin, Ruslan, Safina, Larisa, 2016. Microservices: yesterday, today, and tomorrow. CoRR abs/1606.04036. arXiv:1606.04036. URL http://arxiv.org/abs/1606.04036.

Haindl, Philipp, Kochberger, Patrick, Sveggen, Markus, 2024. A systematic literature review of inter-service security threats and mitigation strategies in microservice architectures. IEEE Access.

Hasselbring, Wilhelm, Steinacker, Guido, 2017. Microservice architectures for scalability, agility and reliability in E-commerce. In: 2017 IEEE International Conference on Software Architecture Workshops. ICSAW, pp. 243–246. http://dx.doi.org/10.1109/ICSAW.2017.11.

Kendall, Judy, 1999. Axial coding and the grounded theory controversy. West. J. Nurs. Res. 21 (6), 743–757.

Kitchenham, Barbara, Charters, Stuart, 2007. Guidelines for performing systematic literature reviews in software engineering. 2.

Kosińska, Joanna, Baliś, Bartosz, Konieczny, Marek, Malawski, Maciej, Zieliński, Sławomir, 2023. Toward the observability of cloud-native applications: The overview of the state-of-the-art. IEEE Access 11, 73036–73052.

Medium, 2024. Medium: Where good ideas find you. https://medium.com/. (Accessed 21 November 2024).

Meta Stack Exchange, 2008. How does reputation work? URL https://meta.stackexchange.com/questions/7237/how-does-reputation-work. (Accessed 20 November 2024). https://meta.stackexchange.com/questions/7237/how-does-reputation-work.

Mparmpoutis, Antonios, Kakarontzas, George, 2022. Using database schemas of legacy applications for microservices identification: A mapping study. In: Proceedings of the 6th International Conference on Algorithms, Computing and Systems. pp. 1–7.

Newman, Sam, 2015. Building Microservices: Designing Fine-Grained Systems, first ed. O'Reilly Media, ISBN: 978-1491950357, p. 280.

Parker, Garrett, Kim, Samuel, Al Maruf, Abdullah, Cerny, Tomas, Frajtak, Karel, Tisnovsky, Pavel, Taibi, Davide, 2023. Visualizing anti-patterns in microservices at runtime: A systematic mapping study. IEEE Access 11, 4434–4442.

Relic, New, 2024. New relic: Monitor, debug and improve your entire stack. https://newrelic.com/. (Accessed 21 November 2024).

Schmidt, Roger Anderson, Thiry, Marcello, 2020. 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, IEEE, pp. 1–6.

Söylemez, Mehmet, Tekinerdogan, Bedir, Tarhan, Ayça Kolukısa, 2024. Microservice reference architecture design: A multi-case study. Softw. - Pract. Exp. 54 (1), 58–84.

Stack Overflow, 2024. What's reputation? URL https://stackoverflow.com/help/whats-reputation. (Accessed 14 November 2024).

Stack Overflow Meta, 2023. What's the average reputation on stack overflow? URL https://meta.stackoverflow.com/questions/281147/whats-the-average-reputation-on-stack-overflow. (Accessed 14 November 2024).

Strauss, Anselm L., Corbin, Juliet, 2004. Open coding. Soc. Res. Methods: Reader 303–306.

Torkura, Kennedy A, Sukmana, Muhammad IH, Cheng, Feng, Meinel, Christoph, 2018. Cavas: Neutralizing application and container security vulnerabilities in the cloud native era. In: Security and Privacy in Communication Networks: 14th International Conference, SecureComm 2018, Singapore, Singapore, August 8-10, 2018, Proceedings, Part I. Springer, pp. 471–490.

Valdivia, Juan Alejandro, Lora-González, A, Limón, X, Cortes-Verdin, K, Ocharán-Hernández, Jorge Octavio, 2020. Patterns related to microservice architecture: a multivocal literature review. Program. Comput. Softw. 46, 594–608.

Waseem, Muhammad, Liang, Peng, Ahmad, Aakash, Shahin, Mojtaba, Khan, Arif Ali, Márquez, Gastón, 2022. Decision models for selecting patterns and strategies in microservices systems and their evaluation by practitioners. In: Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice. pp. 135–144.

## White Studies

Abdelfattah, Amr S., Cerny, Tomas, 2023a. Roadmap to reasoning in microservice systems: a rapid review. Appl. Sci. 13 (3), 1838.

Abdelfattah, Amr S., Cerny, Tomas, 2023b. The microservice dependency matrix. In: Papadopoulos, George A., Rademacher, Florian, Soldani, Jacopo (Eds.), Service-Oriented and Cloud Computing. Springer Nature Switzerland, Cham, ISBN: 978-3-031-46235-1, pp. 276–288.

Al Maruf, Abdullah, Bakhtin, Alexander, Cerny, Tomas, Taibi, Davide, 2022. Using microservice telemetry data for system dynamic analysis. In: 2022 IEEE International Conference on Service-Oriented System Engineering. SOSE, IEEE, pp. 29–38.

Apolinário, Daniel R.F., de França, Breno B.N., 2021. A method for monitoring the coupling evolution of microservice-based architectures. J. Braz. Comput. Soc. (ISSN: 1678-4804) 27 (1), 17. http://dx.doi.org/10.1186/s13173-021-00120-y.

Bajaj, Deepali, Goel, Anita, Gupta, Suresh, 2024. A comprehensive microservice extraction approach integrating business functions and database entities. Int. Arab J. Inf. Technol. (IAJIT) 21 (01), 32–45. http://dx.doi.org/10.34028/iajit/21/1/3.

Batta, Raghav, Shwartz, Larisa, Nidd, Michael, Azad, Amar Prakash, Kumar, Harshit, 2021. A system for proactive risk assessment of application changes in cloud operations. In: 2021 IEEE 14th International Conference on Cloud Computing. CLOUD, IEEE, pp. 112–123.

Chen, Yi-Yuan, Hsu, Kuo-Hsun, Hou, Andrew Weian, 2023. MAT: Automating go monolithic applications transform into microservices through dependency analysis and AST. In: 2023 9th International Conference on Applied System Innovation. ICASI, pp. 133–135. http://dx.doi.org/10.1109/ICASI57738.2023.10179517.

Daoud, Mohamed, El Mezouari, Asmae, Faci, Noura, Benslimane, Djamal, Maamar, Zakaria, El Fazziki, Aziz, 2021. A multi-model based microservices identification approach. J. Syst. Archit. 118, 102200.

Ding, Ruomeng, Zhang, Chaoyun, Wang, Lu, Xu, Yong, Ma, Minghua, Wu, Xiaomin, Zhang, Meng, Chen, Qingjun, Gao, Xin, Gao, Xuedong, et al., 2023. TraceDiag: Adaptive, interpretable, and efficient root cause analysis on large-scale microservice systems. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1762–1773.

Driessen, Famke, Ferreira Pires, Luís, Moreira, João Luiz Rebelo, Verhoeven, Paul, van den Bosch, Sander, 2023. A quantitative assessment method for microservices granularity to improve maintainability. In: International Conference on Enterprise Design, Operations, and Computing. Springer, pp. 211–226.

Engel, Thomas, Langermeier, Melanie, Bauer, Bernhard, Hofmann, Alexander, 2018. Evaluation of microservice architectures: A metric and tool-based approach. In: Information Systems in the Big Data Era: CAiSE Forum 2018, Tallinn, Estonia, June 11-15, 2018, Proceedings 30. Springer, pp. 74–89.

Fekete, Anett, Kovács, Benedek, Porkoláb, Zoltán, 2023. Automatic dependency tracking in microservice-based systems using static analysis in helm charts. In: 2023 International Conference on Software, Telecommunications and Computer Networks. SoftCOM, pp. 1–7. http://dx.doi.org/10.23919/SoftCOM58365.2023.10271686.

Genfer, Patric, Zdun, Uwe, 2021. Identifying domain-based cyclic dependencies in microservice apis using source code detectors. In: Software Architecture: 15th European Conference, ECSA 2021, Virtual Event, Sweden, September 13-17, 2021, Proceedings. Springer, pp. 207–222.

Gortney, Mia E, Harris, Patrick E, Cerny, Tomas, Al Maruf, Abdullah, Bures, Miroslav, Taibi, Davide, Tisnovsky, Pavel, 2022. Visualizing microservice architecture in the dynamic perspective: A systematic mapping study. IEEE Access 10, 119999–120012.

Grohmann, Johannes Sebastian, 2022. Model Learning for Performance Prediction of Cloud-native Microservice Applications (Ph.D. thesis). Universität Würzburg.

Kaushik, Neha, Kumar, Harish, Raj, Vinay, 2022. Empirical evaluation of microservices architecture. In: International Conference on Communication and Intelligent Systems. Springer, pp. 241–253.

Lv, Wenkai, Yang, Pengfei, Zheng, Tianyang, Lin, Chengmin, Wang, Zhenyi, Deng, Minwen, Wang, Quan, 2024. Graph-reinforcement-learning-based dependency-aware microservice deployment in edge computing. IEEE Internet Things J. 11 (1), 1604–1615. http://dx.doi.org/10.1109/JIOT.2023.3289228.

Ma, Shang-Pin, Fan, Chen-Yuan, Chuang, Yen, Liu, I-Hsiu, Lan, Ci-Wei, 2019a. Graph-based and scenario-driven microservice analysis, retrieval, and testing. Future Gener. Comput. Syst. (ISSN: 0167-739X) 100, 724–735. http://dx.doi.org/10.1016/j.future.2019.05.048, URL https://www.sciencedirect.com/science/article/pii/S0167739X19302614.

Ma, Shang-Pin, Liu, I-Hsiu, Chen, Chun-Yu, Lin, Jiun-Ting, Hsueh, Nien-Lin, 2019b. Version-based microservice analysis, monitoring, and visualization. In: 2019 26th Asia-Pacific Software Engineering Conference. APSEC, IEEE, pp. 165–172.

Meng, Chunyang, Song, Shijie, Tong, Haogang, Pan, Maolin, Yu, Yang, 2023. DeepScaler: Holistic autoscaling for microservices based on spatiotemporal GNN with adaptive graph learning. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 53–65.

Nitin, Vikram, Asthana, Shubhi, Ray, Baishakhi, Krishna, Rahul, 2023. 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. ASE '22, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450394758, http://dx.doi.org/10.1145/3551349.3556960.

Qian, Lifeng, Li, Jing, He, Xudong, Gu, Rongbin, Shao, Jiawei, Lu, Yuqi, 2023. Microservice extraction using graph deep clustering based on dual view fusion. Inf. Softw. Technol. (ISSN: 0950-5849) 158, 107171. http://dx.doi.org/10.1016/j.infsof.2023.107171, URL https://www.sciencedirect.com/science/article/pii/S0950584923000253.

Raj, Vinay, Sadam, Ravichandra, 2021. Patterns for migration of SOA based applications to microservices architecture. J. Web Eng. 20 (5), 1291–1307.

Rahman, Mohammad Imranur, Panichella, Sebastiano, Taibi, Davide, 2019. A curated dataset of microservices-based systems. SSSME-2019.

Ren, Rui, Wang, Yang, Liu, Fengrui, Li, Zhenyu, Xie, Gaogang, 2023. Triple:The interpretable deep learning anomaly detection framework based on trace-metric-log of microservice. In: 2023 IEEE/ACM 31st International Symposium on Quality of Service. IWQoS, pp. 1–10. http://dx.doi.org/10.1109/IWQoS57198.2023.10188773.

Saidi, Malak, Tissaoui, Anis, Faiz, Sami, 2023. From a monolith to a microservices architecture based dependencies. In: Abraham, Ajith, Pllana, Sabri, Casalino, Gabriella, Ma, Kun, Bajaj, Anu (Eds.), Intelligent Systems Design and Applications. Springer Nature Switzerland, Cham, ISBN: 978-3-031-35501-1, pp. 34–44.

Sakai, Masaru, Takahashi, Kensuke, Kondoh, Satoshi, 2021. Method of constructing Petri net service model using distributed trace data of microservices. In: 2021 22nd Asia-Pacific Network Operations and Management Symposium. APNOMS, IEEE, pp. 214–217.

Silva, Sara, Correia, Jaime, Bento, Andre, Araujo, Filipe, Barbosa, Raul, 2021. $\mu$Viz: Visualization of microservices. In: 2021 25th International Conference Information Visualisation. IV, pp. 120–128. http://dx.doi.org/10.1109/IV53921.2021.00028.

Smith, Sheldon, Robinson, Ethan, Frederiksen, Timmy, Stevens, Trae, Cerny, Tomas, Bures, Miroslav, Taibi, Davide, 2023. Benchmarks for end-to-end microservices testing. In: 2023 IEEE International Conference on Service-Oriented System Engineering. SOSE, IEEE, pp. 60–66.

Song, Youmei, Li, Chaoran, Zhuang, Kuoran, Ma, Tianjiao, Wo, Tianyu, 2022. An automatic scaling system for online application with microservices architecture. In: 2022 IEEE International Conference on Joint Cloud Computing. JCC, pp. 73–78. http://dx.doi.org/10.1109/JCC56315.2022.00018.

Song, Chenghao, Xu, Minxian, Ye, Kejiang, Wu, Huaming, Gill, Sukhpal Singh, Buyya, Rajkumar, Xu, Chengzhong, 2023. ChainsFormer: A chain latency-aware resource provisioning approach for and microservices cluster. In: Service-Oriented Computing: 21st International Conference, ICSOC 2023, Rome, Italy, November 28 – December 1, 2023, Proceedings, Part I. Springer-Verlag, Berlin, Heidelberg, ISBN: 978-3-031-48420-9, pp. 197–211. http://dx.doi.org/10.1007/978-3-031-48421-6_14.

Speth, Sandro, Breitenbücher, Uwe, Krieger, Niklas, Wippermann, Pia, Becker, Steffen, 2023. Integrating issue management systems of independently developed software components. In: International Conference on Agile Software Development. Springer Nature Switzerland Cham, pp. 3–19.

Sun, Yindong, Zhao, Longjun, Wang, Zhen, Cui, Dandan, Yang, Yang, Gao, Zhipeng, 2021. Fault root rank algorithm based on random walk mechanism in fault knowledge graph. In: 2021 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting. BMSB, pp. 1–6. http://dx.doi.org/10.1109/BMSB53066.2021.9547194.

Tang, Hao, Guo, Yuchun, Yang, Jingjing, Chen, Yishuai, 2023. Microservice anomaly diagnosis with graph convolution network based on implicit microservice dependency. In: Proceedings of the 2023 9th International Conference on Computing and Artificial Intelligence. pp. 437–441.

Wang, Lulu, Hu, Peng, Kong, Xianglong, Ouyang, Wenjie, Li, Bixin, Xu, Haixin, Shao, Tao, 2023. Microservice architecture recovery based on intra-service and inter-service features. J. Syst. Softw. 204, 111754.

Wilta, Alvin, Kistijantoro, Achmad Imam, 2023. Automatic measurement of microservice architecture quality with cohesion, coupling, and complexity metrics. In: 2023 10th International Conference on Advanced Informatics: Concept, Theory and Application. ICAICTA, pp. 1–6. http://dx.doi.org/10.1109/ICAICTA59291.2023.10390525.

Yang, Jingjing, Guo, Yuchun, Chen, Yishuai, Zhao, Yongxiang, 2023. TraceNet: Operation aware root cause localization of microservice system anomalies. In: 2023 IEEE International Conference on Communications Workshops. ICC Workshops, pp. 758–763. http://dx.doi.org/10.1109/ICCWorkshops57953.2023.10283749.

Ye, Zihao, Chen, Pengfei, Yu, Guangba, 2021. T-rank: A lightweight spectrum based fault localization approach for microservice systems. In: 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing. CCGrid, IEEE, pp. 416–425.

Zhang, Qixun, Jia, Tong, Wu, Zhonghai, Wu, Qingxin, Jia, Lichun, Li, Donglei, Tao, Yuqing, Xiao, Yutong, 2022. Fault localization for microservice applications with system logs and monitoring metrics. In: 2022 7th International Conference on Cloud Computing and Big Data Analytics. ICCCBDA, IEEE, pp. 149–154.

Zhang, Yanqi, Zhou, Zhuangzhuang, Elnikety, Sameh, Delimitrou, Christina, 2024. Analytically-driven resource management for cloud-native microservices. arXiv preprint arXiv:2401.02920.

Zheng, Chunyang, Wang, Jinfa, Si, Shuaizong, Zhang, Weidong, Sun, Limin, 2023. ChainDet: A traffic-based detection method of microservice chains. In: 2023 IEEE International Performance, Computing, and Communications Conference. IPCCC, IEEE, pp. 194–201.

Zhong, Chenxing, Zhang, He, Li, Chao, Huang, Huang, Feitosa, Daniel, 2023. On measuring coupling between microservices. J. Syst. Softw. (ISSN: 0164-1212) 200, 111670. http://dx.doi.org/10.1016/j.jss.2023.111670, URL https://www.sciencedirect.com/science/article/pii/S0164121223000651.

Zhou, Jizhe, Wang, Guangchao, Zhou, Wei, 2023. Dependency-aware microservice deployment and resource allocation in distributed edge networks. In: 2023 International Wireless Communications and Mobile Computing. IWCMC, pp. 568–573. http://dx.doi.org/10.1109/IWCMC58020.2023.10182768.

Zuo, Yuan, Zhu, Xiaozhou, Qin, Jiangyi, Yao, Wen, 2021. Temporal relations extraction and analysis of log events for micro-service framework. In: 2021 40th Chinese Control Conference. CCC, IEEE, pp. 3391–3396.

## Grey Studies

Abdolhosseini, Majid, 2022. Services dependencies and relations documentation. https://stackoverflow.com/questions/73195886/micro-services-dependencies-and-relations-documentation. (Accessed 16 November 2024).

Abrar, Syed Mohammad Fahim, 2019. How to send notification from microservice to asp.net core's SignalRHub. https://stackoverflow.com/questions/58110187/how-to-send-notification-from-microservice-to-asp-net-cores-signalrhub. (Accessed 16 November 2024).

Bräutigam, Robert, 2019. How to manage event dependencies in event-driven architectures? https://softwareengineering.stackexchange.com/questions/399308/how-to-manage-event-dependencies-in-event-driven-architectures/399311#399311. (Accessed 16 November 2024).

Burghardt, Greg, 2023. How to handle dependencies between microservices all called within one large service. https://softwareengineering.stackexchange.com/questions/444378/how-to-handle-dependencies-between-microservices-all-called-within-one-large-ser/444381#444381. (Accessed 16 November 2024).

Chase, Dan, 2018. How do you determine what should be a microservice? https://stackoverflow.com/questions/50779254/how-do-you-determine-what-should-be-a-microservice/50779279#50779279. (Accessed 16 November 2024).

Chugh, Akshay, Ahire, Jayesh Bapu, 2022. Microservices: Tracking dependencies. https://last9.io/blog/microservices-tracking-dependencies/. (Accessed 20 November 2024).

CodeScene Engineering Team, 2023. Change coupling: Visualize the cost of change. https://codescene.com/engineering-blog/change-coupling-visualize-the-cost-of-change. (Accessed 20 November 2024).

Custodio, Thiago, 2016. Best practice for loading multiple dynamic services and them dependencies services. https://stackoverflow.com/questions/39561186/best-practice-for-loading-multiple-dynamic-services-and-them-dependencies-servic/39562606#39562606. (Accessed 16 November 2024).

downtownthe, 2018. Net microservices with CQRS handling dependencies. https://stackoverflow.com/questions/50927770/net-microservices-with-cqrs-handling-dependencies. (Accessed 16 November 2024).

Euphoric, 2020. Microservice should be an independent software unit - up to which level? https://softwareengineering.stackexchange.com/questions/406589/microservice-should-be-an-independent-software-unit-up-to-which-level/406590#406590. (Accessed 16 November 2024).

Flater, 2021. Adding new service into existing micro-service system. https://softwareengineering.stackexchange.com/questions/431087/adding-new-service-into-existing-micro-service-system/431129#431129. (Accessed 16 November 2024).

Frost, Daniel, 2020. Sharing DTO objects between microservices. https://softwareengineering.stackexchange.com/questions/337010/sharing-dto-objects-between-microservices/420459#420459. (Accessed 16 November 2024).

gsf, 2017. What are the potential problems with operational circular dependency between microservices. https://softwareengineering.stackexchange.com/questions/350155/what-are-the-potential-problems-with-operational-circular-dependency-between-mic. (Accessed 16 November 2024).

Hijazi, Nabil, 2016. Dependency hell in microservices and how to avoid it. https://www.linkedin.com/pulse/dependency-hell-microservices-how-avoid-nabil-hijazi. (Accessed 16 November 2024).

JonasH, 2021. Microservices inter-communication when response is needed. https://softwareengineering.stackexchange.com/questions/433985/microservices-inter-communication-when-response-is-needed/433992#433992. (Accessed 16 November 2024).

KitKarson, 2021. Data dependency among microservices. https://stackoverflow.com/questions/70509292/data-dependency-among-microservices. (Accessed 16 November 2024).

Kriens, Peter, 2018. What is the specific difference between java modules (in java 9), OSGi bundles and microservices? and what specific purpose is served by each of them? https://stackoverflow.com/questions/67370182/what-is-the-specific-difference-between-java-modules-in-java-9-osgi-bundles-a/67373943#67373943. (Accessed 16 November 2024).

Link, René, 2018. Clean architecture design pattern. https://stackoverflow.com/questions/52352815/clean-architecture-design-pattern/52354886#52354886. (Accessed 16 November 2024).

Mangano, Leonardo, 2022. Circular dependencies between microservices. https://dev.to/cloudx/circular-dependencies-between-microservices-11hn. (Accessed 20 November 2024).

Manish Kumar, 2018. Resolving design dependency between microservices. https://stackoverflow.com/questions/53209975/resolving-design-dependency-between-microservices. (Accessed 16 November 2024).

Martin, K., 2020. Data from one microservice to another. https://softwareengineering.stackexchange.com/questions/403370/data-from-one-microservice-to-another/403521#403521. (Accessed 16 November 2024).

MicroMaster, James Youngman, 2019. Cyclic dependencies in microservices. https://softwareengineering.stackexchange.com/questions/398453/cyclic-dependencies-in-microservices. (Accessed 16 November 2024).

Naros, 2015. How to decompose monolithic applications into microservices with interdependent functional modules? https://softwareengineering.stackexchange.com/questions/286767/how-to-decompose-monolithic-applications-into-microservices-with-interdependent/286769#286769rdependent/286769#286769. (Accessed 16 November 2024).

Next, I.T., 2019. Isolating and managing dependencies in 12-factor microservice applications with kubernetes. https://itnext.io/isolating-and-managing-dependencies-in-12-factor-microservice-applications-with-kubernetes-988638f8bc6d. (Accessed 14 November 2024).

Pai, Santosh, 2023. Isolating and managing dependencies in 12-factor microservice applications with kubernetes. https://itnext.io/isolating-and-managing-dependencies-in-12-factor-microservice-applications-with-kubernetes-988638f8bc6d. (Accessed 20 November 2024).

Ragan, Tracy, 2020. Microservice dependency manager tools. https://stackoverflow.com/questions/53610908/microservice-dependency-manager-tools. (Accessed 16 November 2024).

Ramesh, Aarish, 2018. Microservice dependency manager tools. https://stackoverflow.com/questions/53610908/microservice-dependency-manager-tools/62799384#62799384. (Accessed 16 November 2024).

Razorops, Inc., 2023. 5 tips for managing service dependencies in a microservice architecture. https://www.linkedin.com/pulse/5-tips-managing-service-dependencies-microservice-architecture/. (Accessed 20 November 2024).

Rewari, Aditya, 2020. Springboot: difference between responsibilities of dependencies grouped as STARTER vs CORE. https://stackoverflow.com/questions/61504001/springboot-difference-between-responsibilities-of-dependencies-grouped-as-start. (Accessed 16 November 2024).

Richardson, Chris, 2021. Microservices: Design-time coupling. https://www.infoq.com/presentations/microservices-design-time-coupling/. (Accessed 20 November 2024).

Richardson, Chris, 2023. Microservice architecture essentials: Loose coupling. https://microservices.io/post/architecture/2023/03/28/microservice-architecture-essentials-loose-coupling.html. (Accessed 20 November 2024).

Rossom, Tom Van, 2017. Designing java project for monoliths and microservices at same time. https://stackoverflow.com/questions/47527983/designing-java-project-for-monoliths-and-microservices-at-same-time/47655524#47655524. (Accessed 16 November 2024).

says Reinstate Monica, David, 2017. In a loosely coupled microservices architecture, how do you keep track of your dependencies? https://softwareengineering.stackexchange.com/questions/341904/in-a-loosely-coupled-microservices-architecture-how-do-you-keep-track-of-your-d. (Accessed 16 November 2024).

Sebrechts, Joeri, 2016. Autonomous microservices, event queues and service discovery. https://softwareengineering.stackexchange.com/questions/333409/autonomous-microservices-event-queues-and-service-discovery/333924#333924. (Accessed 16 November 2024).

SebS, 2019. How to decouple data between microservices? https://stackoverflow.com/questions/58194710/how-to-decouple-data-between-microservices. (Accessed 16 November 2024).

Shah, Jigar, 2015. Parent POM and microservices. https://stackoverflow.com/questions/27865238/parent-pom-and-microservices/43570283#43570283. (Accessed 14 November 2024).

Shiloah, Yaniv, 2023. Dependency hell when using same npm module for both sdk and service itself. https://stackoverflow.com/questions/75429757/dependency-hell-when-using-same-npm-module-for-both-sdk-and-service-itself. (Accessed 16 November 2024).

Siemion, Adam, 2015. Microservices bounded context in distributed analytics. https://stackoverflow.com/questions/34450732/microservices-bounded-context-in-distributed-analytics/34470230#34470230. (Accessed 16 November 2024).

Sutty1000, 2017. How to manage dependencies for microservices? https://softwareengineering.stackexchange.com/questions/342281/how-to-manage-dependencies-for-microservices. (Accessed 16 November 2024).

Tayade, Amit, 2021. How to avoid microservice dependency without slowing down your release process. https://stackoverflow.com/questions/70255797/how-to-avoid-microservice-dependency-without-slowing-down-your-release-process. (Accessed 16 November 2024).

theDmi, 2016. Context mapping - relations. https://stackoverflow.com/questions/39791667/context-mapping-relations/39806514?r=SearchResults&s=264%7C39.5482#39806514. (Accessed 16 November 2024).

Tsonev, Dimitar, 2017. Microservices why use rabbitmq? https://stackoverflow.com/questions/45208766/microservices-why-use-rabbitmq/45208903#45208903. (Accessed 16 November 2024).

vaibhav, galkin, 2017. Dockering a nodejs application with external dependencies. https://stackoverflow.com/questions/45648115/dockering-a-nodejs-application-with-external-dependencies. (Accessed 16 November 2024).

Villar, Vitor, 2019. How to slice down a monolith where my domain depends heavily from other domains? https://stackoverflow.com/questions/57065964/how-to-slice-down-a-monolith-where-my-domain-depends-heavily-from-other-domains. (Accessed 16 November 2024).

Walpita, Priyal, 2020. Coupling and cohesion in microservices. https://priyalwalpita.medium.com/coupling-and-cohesion-in-microservices-235ed9203843. (Accessed 20 November 2024).