applied sciences

MDPI

Article

Roadmap to Reasoning in Microservice Systems: A Rapid Review

Amr S. Abdelfattah and Tomas Cerny *

Computer Science, ECS, Baylor University, One Bear Place #97141, Waco, TX 76798-7356, USA
* Correspondence: tomas_cerny@baylor.edu

Abstract: Understanding software systems written by others is often challenging. When we want to assess systems to reason about them, i.e., to understand dependencies, analyze evolution trade-offs, or to verify conformance to the original blueprint, we must invest broad efforts. This becomes difficult when considering decentralized systems. Microservice-based systems are mainstream these days; however, to observe, understand, and manage these systems and their properties, we are missing fundamental tools that would derive various simplified system abstract perspectives. Microservices architecture characteristics yield many advantages to system operation; however, they bring challenges to their development and deployment lifecycles. Microservices urge a system-centric perspective to better reason about the system evolution and its quality attributes. This process review paper considers the current system analysis approaches and their possible alignment with automated system assessment or with human-centered approaches. We outline the necessary steps to accomplish holistic reasoning in decentralized microservice systems. As a contribution, we provide a roadmap for analysis and reasoning in microservice-based systems and suggest that various process phases can be decoupled through the introduction of system intermediate representation as the trajectory to provide various system-centered perspectives to analyze various system aspects. Furthermore, we cover different technical-based reasoning strategies and metrics in addition to the human-centered reasoning addressed through alternative visualization approaches. Finally, a system evolution is discussed from the perspective of such a reasoning process to illustrate the impact analysis evaluation over system changes.

Keywords: architecture reconstruction; microservices visualization; architecture degradation; evolution

Citation: Abdelfattah, A.S.; Cerny, T. Roadmap to Reasoning in Microservice Systems: A Rapid Review. *Appl. Sci.* **2023**, *13*, 1838. https://doi.org/10.3390/app13031838

Academic Editors: Sanjay Misra, Robertas Damaševičius and Bharti Suri

Received: 22 December 2022
Revised: 19 January 2023
Accepted: 24 January 2023
Published: 31 January 2023

1. Introduction

Microservices architecture is a specific style of service-oriented architecture. It is built of microservices that are relatively small and loosely coupled, running in their processes and communicating through lightweight communication protocols [1,2]. This architecture promotes dynamic and independent development, scaling, and deployment [1,3]. Therefore, several organizations, such as Netflix, Amazon, and Spotify, have adopted microservice architecture to build scalable, maintainable software systems [4,5]. This architecture style influences the structure of organizations, such that individual components are typically developed by a specific team; dependencies between components lead to dependencies between teams [1]. Moreover, multiple surveys (https://martinfowler.com/articles/microservices.html accessed on 1 January 2023) [6,7] identified and highlighted the following microservices characteristics,

1. Autonomous: Separately and independently developed and deployed within their own lifecycle.
2. Scalability: Horizontally scaling up and down per the need, while they are small and stateless components.
3. Heterogeneity: Various programming languages and technologies can involve the same system.

4. Specialization: Each microservice handles specific domain logic and communicates with other microservices through communication protocols.
5. Fault Tolerance: They tolerate failures by applying different resiliency techniques, such as a failover mechanism whenever a service fails.

These characteristics provide many advantages; however, they bring challenges to the system's development and evolution at the same time. These challenges become apparent when we aim to reason about the system as a whole to understand the dependencies, impact analysis, departures from the original architectural blueprint, etc.

In general, a reasoning system [8] is a program that generates conclusions from the available knowledge about a particular system. Reasoning systems offer two modes human-centered and technical reasoning. The human-centered focus is on interaction with human experts who guide the processing or lead to conclusions. Technical reasoning is batch processing that considers all the available information at once to generate the answer.

When making development decisions, we do not use any reasoning system and base our decisions on our limited or approximated knowledge of the system. In microservices, developers often understand their microservice needs within a bounded domain context but typically do not consider the holistic system perspective as it evolves quickly and transparently while being managed by distinct development teams.

In microservices, the reasoning challenges are the non-existence of the holistic system perspective given the decentralized system nature. This is further exacerbated when dealing with the autonomous and heterogeneity characteristics of the architecture. This limits the capabilities to understand the system completely and reason about it. Furthermore, the traditional software visualization approaches serving human experts are limited when we attempt to fit them into the enterprise and scalability characteristics of microservices [1,6,9]. Finally, the ever-changing evolution aspect with many moving parts changing continuously makes it difficult when assuring various quality attributes [6].

Facing these challenges requires constructing different representations and views of the microservices architecture. These different system viewpoints can help in understanding and reasoning about the system and its properties [1]. This paper gives an overview of the microservice-based reasoning process and its different perspectives. We review some of the highlighted literature and illustrate and discuss the reasoning process stages in more detail. We start with addressing architecture reconstruction techniques and how this helps in deriving different reasoning approaches. Moreover, we present various evaluation perspectives of the system and the different strategies for achieving them. After that, we illustrate the microservice-based evolution process and its related challenges. Furthermore, we highlight the visualization challenges and approaches to present the microservice-based representations and evaluation for human-centered reasoning.

The rest of the paper is organized as follows: Section 2 discusses background information. Section 3 outlines the research questions for this research and a roadmap for addressing them. Then, Section 4 shows the different analytic techniques and the current approaches for extracting and reconstructing microservice architecture. This is followed by Section 5 with the technical reasoning and the microservices visualization techniques. After that, Section 6 illustrates the microservice-based evolution process and its challenges. Finally, the paper is concluded in Section 8.

2. Background

Microservices use unified platform-independent communication protocols for the exchange of information [6]. These communication protocols can be synchronous or asynchronous. The REpresentational State Transfer (RESTful) (https://www.w3.org/2001/sw/wiki/REST accessed on 1 January 2023) architectural communication style commonly supports synchronous communication through the HTTP protocol (https://www.w3.org/Protocols accessed on 1 January 2023). In some cases, RESTful can also be used as an asynchronous or what is called "deferred synchronous" [1]. Using it asynchronously as

follows, the service immediately answers a request with an *Accepted* response code and a link to obtain actual results once the operation has finished.

Other approaches use user-defined callback methods, such that the client specifies a link that the server should call with the results of the request once ready [10]. Furthermore, asynchronous message communication is the common asynchronous way of interaction in microservice-based systems. It is achieved as an indirect communication through message queues or some other kind of message-based broker [1].

OpenAPI (https://github.com/OAI/OpenAPI-Specification/blob/main/versions/2.0.md accessed on 1 January 2023) specification (formally known as: Swagger (https://swagger.io accessed on 1 January 2023) RESTful API Documentation Specification) is the most widely-used specification for describing RESTful web services [11]. The content of the specification includes many attributes, such as the API title, version, contact information, path, and HTTP method (such as GET, POST, and DELETE.). It can also include input parameters and response descriptions for services. A variety of tools can generate code, documentation, and test cases based on this specification. Moreover, Swagger is available for a broad spectrum of technologies, and the provided API information is based on the OpenAPI specification [1].

3. A Rapid Review Design and Research Questions

Microservices architecture supports independent development, deployment, and scalability. This architecture emerges by dynamically assembling services to systems [1]. Managing and reasoning about the different perspectives of the system raise multiple challenges, particularly with the large number of microservices. In other words, for addressing the dependency perspective, what is the process of managing the complex dependency relationships between microservices in a system? However, it is even more challenging to consider the continuous changes and their impact on the system architecture during the system evolution [6].

The above challenges befit being addressed through the Rapid Review (RR) design method [12] in this paper. The RR method emerges from a practical problem that is more appealing to practitioners. It is a lightweight secondary study that focuses on delivering evidence promptly [12,13]. Therefore, the RR is very applicable, particularly to software factories and institutes that conduct applied research in collaboration with industry [12]. Although systematic reviews produce objective evidence, multiple studies [14–16] demonstrated a lack of connection between that evidence and the needs of software engineering practice. The RR method omits and simplifies certain steps of systematic review methods to deliver evidence in a timely manner with lower costs [12]. Therefore, we limited our scope of search as detailed below and broadened our snowballing and collaboration phase. However, this should be considered a complementary method and not a substitute for systematic reviews.

3.1. Research Questions

This paper involves two main factors for applying the RR design as follows: the practical problem and expert collaboration in forming a solution. Combining those factors, we expressed the addressed problem through the following research questions:

RQ$_1$ *What are the different perspectives of microservice-based architecture, what are the challenges to reconstructing these perspectives holistically for a system, and what are the current techniques that elaborate on solutions?*

RQ$_2$ *What are the reasoning aspects of microservice-based architecture, what are the challenges to visualizing the perspectives of microservice-based architecture to reason about, and what are the current approaches that contribute to overcoming these challenges?*

RQ$_3$ *What are the challenges in cloud-native systems evolution, and how to efficiently assure the quality of continuously changing systems?*

Answering these questions is emphasized as an illustration of the automatic reasoning process for microservice-based systems. The process phases are shown in Figure 1. We start by reconstructing a holistic intermediate representation of decentralized system codebases and artifacts. After that, we construct two reasoning approaches. Human-centered reasoning visualizes the different perspectives to facilitate reasoning about the system to the system stakeholders. Technical reasoning evaluates the system performance using different techniques and metrics to detect, report, and visualize the system health indicators.

The following sub-sections provide a more in-depth look into the details of the search process. The sections that follow are based on this process, resulting in a comprehensive overview of the microservices reasoning roadmap. This includes a detailed explanation of each phase of the process in addition to providing reviewed techniques for applying them.

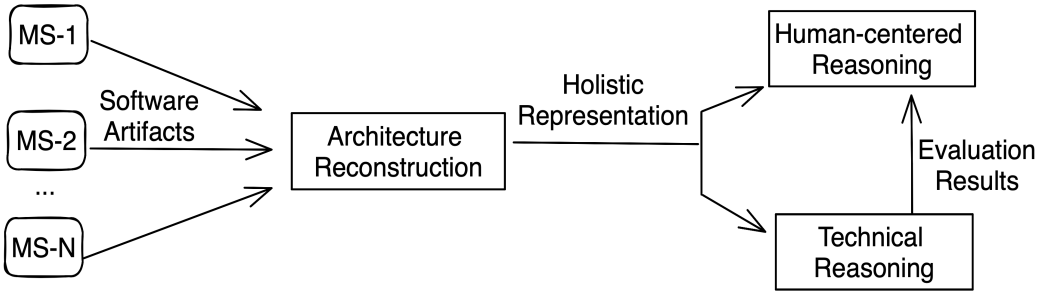

Figure 1. Microservices reasoning process overview.

3.2. Searching for Relevant Literature

We conducted a literature search for relevant publications on the topic of microservice reasoning roadmaps. We used multiple search queries in Scopus, IEEE, and ACM, with a focus on logical combinations of keywords, such as "microservices", "reasoning", and "roadmap". However, the search yielded few or no relevant results. To overcome this, we employed our expertise in microservices to combine different approaches to construct and illustrate a process for achieving microservice-based reasoning. We limited the search scope to Scopus, and we searched in the metadata of articles (Title, Abstract, and Indexing terms). We used the search query to identify techniques used in answering the mentioned research questions and drawing the reasoning process roadmap.

To refine the search and focus on the core scope of the review, we constructed a search query that included the keywords "microservice" and "architecture". To support each research question, we also included the following keywords: RQ1: any of "reconstruction" or "intermediate representation", RQ2: "reasoning", and RQ3: both of "evolution" and "changes" to indicate the microservices evolution techniques during the changes of the architecture. To ensure that the articles included a described technique or model that can be used, we included the keywords "technique", "solution", and "model" in the search query. The search was conducted until the end of 2022.

Scopus-based Search Query *TITLE-ABS-KEY ((microservices OR microservice OR microservice) AND architecture AND ((reconstruction OR "Intermediate Representation") OR reasoning OR (evolution AND changes)) AND (technique OR solution OR model)) AND PUBYEAR > 1959 AND PUBYEAR < 2023 AND PUBYEAR > 1959 AND PUBYEAR < 2023*

After the snowballing process and ad hoc search process, we identified any additional relevant literature to find more options to build the roadmap for the practitioners and

researchers. Therefore, we searched into the white and grey literature for each related individual part of the process and included the eligible articles in the study.

3.3. Screening and Eligibility Criteria

We conducted a search using the specified search query above, which resulted in a total of 70 papers being gathered. To ensure that only relevant papers were included in the review, a three-step filtration process was employed. The first step involved narrowing down the papers to 50 that met the following exclusion criteria or did not meet the inclusion criteria.

Inclusion criteria:

- *Papers including applied techniques of microservices reconstruction/reasoning/evolution.*
- *Papers showing a use-case for validating their results.*
- *Papers supporting a categorization methodology for reconstruction/reasoning/evolution.*
- *Papers providing a discussion of microservices reconstruction/reasoning/evolution.*

Exclusion criteria:

- *Not in microservices scope.*
- *Literature reviews.*
- *Not an article/paper.*
- *Not in English.*
- *Duplicate.*
- *Opinion paper.*

The second step involved eliminating papers based on their title and abstract, which resulted in 17 papers. Finally, the third step involved reading through the full text of the remaining papers to further filter out any irrelevant papers, resulting in a final pool of 10 papers. The papers are filtered based on the steps of the filtration process, and these steps are outlined in the link (Search Papers: https://zenodo.org/record/7549228#.Y8iJhy2B3RY accessed on 20 January 2023) to the filtered papers.

After filtering the papers, we analyzed them to extract the information needed to answer the research questions. We also used a snowballing process to find any additional relevant papers or articles during the analysis. The snowballing process was broad and included more extensive searching and discussion. This process was conducted in close collaboration with experts in the field, who participated in providing feedback and discussing the various components and workflow of the process.

3.4. Extracting Data from Papers

In the data extraction phase, we analyzed the filtered papers to extract the necessary information for answering the research questions. This involved inspecting the papers for any data related to categories such as architecture reconstruction, reasoning (technical-based or human-centered), and evolution processes. It is important to note that a paper may contain information that pertains to multiple categories, such as both architecture reconstruction and reasoning, and therefore may be included in multiple sections of the review.

The information obtained from the papers can be classified into two categories. The first category includes techniques and solutions, where a summary of the technique, its impact on the reasoning process, and any challenges or discussions related to it are extracted. Additionally, any potential future developments or extensions to the method are also noted. The second category includes additional classifications or information that can be used to expand the process and add more details to its components. This information helps to organize the paper, identify key points that need to be addressed, and provide more detailed explanations of the process and its various stages.

4. Architecture Reconstruction

Software architecture reconstruction is the process of extracting different representations from software artifacts. These software artifacts are categorized into static and dynamic data as shown in Figure 2 corresponding to the nature of the data and the stage of the data generation [17,18].

Source code repositories hold extensive static information such as source code, configuration files, documentation, code reviews, and code analysis through the progression of commits and commit messages. The dynamic data is produced during the runtime, such as logs, traces, and telemetry data. Moreover, some artifacts share both static and dynamic behaviors, such as tickets, which can be static (requirements or change requests) or dynamic (production issues).

The different representations give a better understanding of the system from multiple perspectives. These perspectives support the reasoning about the system to the three main roles (architect, software developer, and DevOps engineer), each with a different point of view [18,19]. The architect is concerned with the overall view of the applications components and their interactions. The software developer implements the requirements functionalities and their changes. The DevOps engineer manages the deployment plans and monitors the running system's health.

Moreover, the reconstruction process is more challenging in microservice-based systems, such that each microservice is self-contained, codebase independent, and can be implemented using different languages and conventions [20]. Reconstructing these representations for the microservice-based systems implies deriving centralized views and concerns of the overall decentralized system. Analysis techniques are employed to generate these different intermediate representations as illustrated in Figure 2. Furthermore, they can be post-processed to fuel multiple advances in facing the complexity of these systems. These representations are the cornerstones of the human-centered and technical reasoning approaches. Moreover, they facilitate the tracking of these complex systems' performance and their architecture improvements and degradation during the evolution process [17,18].

4.1. Analysis Techniques

The analysis process needs to consider and understand the nature of the data to provide the required tailored representations [21]. Therefore, the analysis techniques are recognized based on the data required to process whether static or dynamic. Although microservices allow the use of different languages and technologies, combining and analyzing heterogeneous artifacts to reason about the whole system is challenging [2].

Microservices architectural information is important at three different levels, service, interaction, and infrastructure [1]. This paints a clear image of the different perspectives of the system. Thus, the analysis process produces intermediate representations to reflect these various aspects of the system as depicted in Figure 2. Some consider the service level of process flow, API descriptions, domain model, etc. Others represent the communication relationships between services and the deployment infrastructure information [1]. Therefore, to retrieve an overall view of the system, the static and dynamic information must be combined and analyzed.

Some common representations that cover these levels are defined and summarized in Table 1. Their classification listed that static analysis fosters service-level extraction. The dynamic analysis contributes more to the infrastructure level. Nevertheless, both static and dynamic techniques support the interaction level.

4.1.1. Static Analysis

The static analysis techniques inspect the static data to construct a system perspective representation [22,23]. It extracts the service description and API specifications. Furthermore, it helps in converting from one programming language to another [1]. It generates several system representations considering the service and interaction level information as highlighted in Table 1.

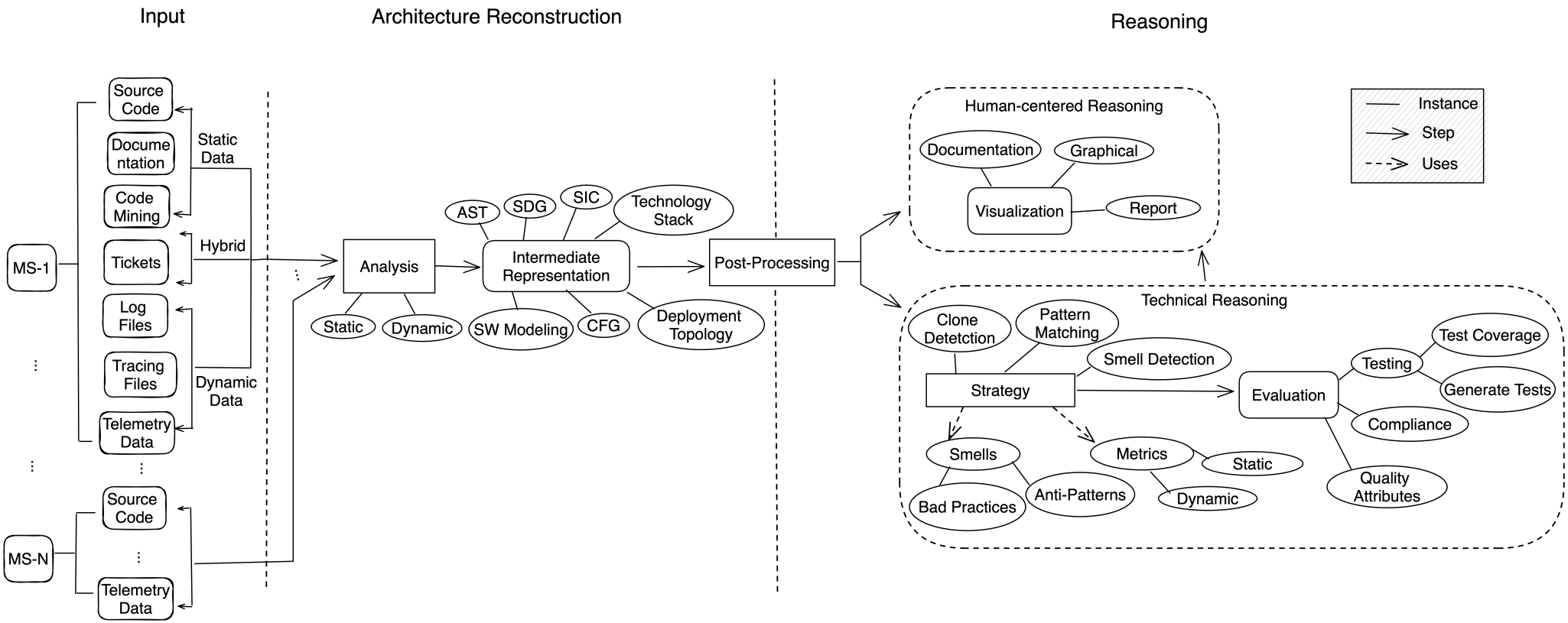

Figure 2. Microservices reasoning process.

The static analysis techniques produce an Abstract Syntax Tree (AST). This represents the code structure in a hierarchical data structure. Although it differs from one language to the other [21], it is the base for extracting most other representations. AST drops all the syntactic clutter that appears in the Concrete Syntax Tree (CST).

The static analysis generates a Class Collaboration Network (CCN) and Package Collaboration Network (PCN) to model the internal structure of services. These representations are constructed through multiple artifacts. Savić et al. [24] and Wen et al. [25] generated the CNN by parsing the Java source code. Wheeldon et al. [26] used Java Doclet capabilities to inspect the source code structure, while Puppin et al. [27] parsed JavaDoc HTML pages.

Furthermore, CNN and PCN collaborate to generate a system domain model. The domain model is an important element of microservice design. These domain models are valid within a specific context (called a bounded context) [28]. On the other hand, the technology stack shows the different technologies used in each service, which impacts the decision of moving functionality from one service to the other [28].

Nevertheless, these representations' structures are tightly coupled with source code languages. For example, Pascal-like syntax `repeat-until` and Java-like syntax `do-while` are represented differently, although they are semantically equivalent. Challenging this heterogeneity urges the need for universal representation. Universal representation describes elements of language semantics independently of language and its syntax. As an instance, the universal node `LOOP_STATEMENT` captures the equivalence in semantics for the above example of repeat-until and do-while [28,29]. Therefore, extracting the architectural information in a universal representation requires a suite of extraction parsers [22].

Rakić et al. [28] proposed an enriched Concrete Syntax Tree (eCST) [30] that enriches a mixture of AST and CST with universal nodes. Therefore, it contains an abstraction of the program structure needed for most program analyses, apart from all concrete program elements. The eCST is generated using a suite of language-specific parsers to achieve language independency—nevertheless, it is not built for microservices architecture. Furthermore, Schiewe et al. [29] focused on the microservices architecture. They presented a universal intermediate representation called Language-Agnostic Abstract-Syntax Tree (LAAST). This represents a low-level structure of the source code, however, it is assessed by generalized component parsers that extract relevant high-level information, such as components.

4.1.2. Dynamic Analysis

The dynamic analysis techniques examine the running system outcomes. They monitor and extract the services interactions, infrastructure, and runtime deployment information [1]. Microservices architecture guides services to be individually deployed and scaled. The running services should not influence each other to avoid cascading failures. Moreover, microservices may require different runtime environments because they may use different implementation technologies [1,31].

Representing the infrastructure as part of the architecture is important. As shown in Table 1, the deployment topology indicates the infrastructure and the deployed system component health [1]. The dynamic analysis technique detects multiple deployment-related behaviors as follows [1],

- Services Distribution: Checking if service is available at least twice, if a service is available in all data centers, or if specific services are running on the same physical machine.
- Call Balance: Measuring the number of requests between the running service instances. It considers the never called service instances.
- Failing Host Prediction: Predicting the impact of failing hosts to improve the distribution of microservices.
- Crosses Data Center Boundaries: Determining whether or not service communication crosses data center boundaries.

The dynamic analysis entails inspecting the possible execution paths and scenarios of the system; however, some tasks require the collaboration of the static analysis to be achieved.

Table 1. Intermediate representations summary.

Representation	Definition	Technique	Level
AST	The Abstract Syntax Tree is a tree representation that conveys the structure of the source code. It is used by compilers to read code and generate the target binaries [21].	Static	Service
CFG	The Control Flow Graph is also called Static Call Graph and, for object-oriented software systems, is known as a Method Collaboration Network [32,33]. Each CFG is associated with its semantic meaning in the system, where nodes are methods and edges are calls between these methods [34].	Hybrid	Service
Models	Class Collaboration Network (CCN): It represents the collaboration of classes and interfaces to reference each other.	Static	Service
	Package Collaboration Networks (PCN): It models the package collaboration, such that two packages are connected when a package contains a class that references at least one class belonging to the other package.		
	Domain Model: It is a conceptual model of the domain that incorporates both behavior and data relations. Both domain models and bounded context models are also part of a service description [28].		
Technology Stack	It presents the programming language and technologies that are used in each microservice [1].	Static	Service
SDG	The Service Dependency Graph describes dependencies between microservices. It shows where microservices call each other [1,18].	Hybrid	Interaction
SIC	The Service Invocation Chain indicates a sequence of service invocations between multiple microservices. It provides supplementary information to SDG to identify the system request paths [6].	Hybrid	Interaction
Deployment Topology	It represents the deployment distributions of microservices in a system. It shows information about physical infrastructure and which service instances are deployed onto which machine [1].	Dynamic	Infrastructure

4.1.3. Hybrid Analysis

Some representations can be constructed either statically or dynamically as shown in Table 1. A Service Dependency Graph (SDG) shows the dependencies between microservices. Static analysis techniques can extract full dependencies from the codebase artifacts, while the dynamic analysis techniques process the logs and tracing data to construct the executed dependency graph [1,6]. It is the same for the Control Flow Graph (CFG), which is also known as the static call graph, and Method Collaboration Network (MCN) [32,33].

The static analysis uses the source code to depict the internal process flows of the service, while the dynamic analysis uses runtime instrumental information to record all calls and returns that occur in a service's execution [35]. The constructed runtime paths group together the sequence of calls that correspond to a given request. Thus, the static analysis can extract the complete call network of the system.

Service Invocation Chain (SIC) inspects request-related endpoints. The static analysis techniques can construct the SIC by assessing semantic control flows in the source code. On the other hand, it is straightforward to extract these dynamically through tracing data. However, the dynamic analysis can provide information about the strength of service interactions, which is essential to identify those services that must be scaled together [1,3].

The collaboration between static and dynamic analysis techniques is necessary for extracting information as unused endpoints. Comparing the extracted full API description with the actual endpoint calls from the dynamic traces data [1]. It is also possible to identify the barely used or unused microservices. This could enhance the microservice system performance and size by removing these unused microservices instead of keeping them maintainable [1].

4.2. Approaches

The scientific interest increasing in extracting the architecture of microservice-based systems; however, little investigation has been performed in this direction [1,36].

Mayer et al. [1] applied the hybrid analysis technique to extract the architecture of REST-based microservice systems. They generated API descriptions and captured the communication relationships along with infrastructure-related information at runtime. The static information extraction depended on Swagger to generate the API descriptions of services. The dynamic data was analyzed of outgoing and incoming requests of each service instance from log files. Each log entry represents a specific request, comprising a timestamp, response time, response code, a unique ID of the source service instance, the URL of the target service instance, and the requested method.

An aggregation service periodically combines these all log files into a holistic one. This approach only supports the Spring Framework (https://spring.io accessed on 1 January 2023) and the Restful APIs connections. Furthermore, supporting additional technology stacks requires specific support for implementing interceptors for service calls. Finally, they applied the approach in a system containing a few nodes. Then, they conducted an interview study [37], including 15 architects, developers, and operations experts. The study concluded that information about service APIs, service interactions and dependencies, service version, the number and distribution of service instances, and system metrics as most important.

Rakić et al. [28] presented an approach that is concerned with the quality metrics of heterogeneous software code, such as Cyclomatic Complexity (CC) and Lines of Code (LOC). They implemented a static analysis technique to construct the eCST universal representation for the source code. This approach supported six representative programming languages, Java, C#, Delphi, Modula-2, Pascal, and COBOL. They implemented a parser generator to generate language-specific parsers to automate the process of adding support for an additional language.

Moreover, they developed a filter mechanism that executes a sequence of parameterized *"select - connected by"* queries. It extracts specific representations from the eCST, for example, CFG is obtained by selecting all FUNCTIONS connected by CALLS links. This approach was validated by extracting isomorphic software networks for two small but structurally and semantically equivalent programs written in the specified programming languages. They built an eCST that contains more than one million nodes in less than a minute.

Nevertheless, this approach cannot differentiate semantics on the paring level; for example, in Java, it cannot decide if some class is abstract or concrete because of the same syntax of declaration. That could require post-processing on the tree to provide this information. Moreover, it does not consider the distributed systems and most of their representations, such as SDG and SIC.

Ma et al. [6] proposed an approach for analyzing microservice-based systems. It uses reflection and Swagger to statically extract API descriptions and service call information. It depends on Swagger for the implementation languages that do not support the feature of reflection. It constructs and visualizes the SDG and SIC to detect anomalies and enable reasoning about the system dependencies. To validate the approach, they generated 11 simulated microservice-based applications with various numbers of microservices ranging from 10 to 1000, and they allocated service calls between them randomly. The results indicated that this approach yielded acceptable performance for the creation of the representations. It also demonstrated the feasibility of being applied in both small-scale and large-scale systems despite the used visualization rendering space being very limited. Although they focused on microservice-based system reconstruction, they did not include message queue communications in the analysis process.

Ishida et al. [38] presented a technique for creating SDG using only logged data. The proposed approach is composed of four main steps. The first step involves analyzing the structure of each logger to identify any synonym attributes that may be present. The second step requires reconciling the schemas by applying Formal Concept Analysis. The third step involves developing a set of rules, based on domain knowledge, to associate log entries and form chains of related events. They recommended the use of Markov Logic to define these rules and Markov Logic Networks [39] to deduce whether two events are related in the context of a transaction. In the final step, the chains of associated events are analyzed to create a Microservice Dependency Graph.

Singh et al. [40] introduced the ARCHI4MOM method, a framework for describing and analyzing asynchronous communication and implementation for specific frameworks. The method extracts the communication components involved in asynchronous messaging and reconstructs the architecture through dynamic analysis of tracing information. The authors demonstrated the method using a system based on Kafka messaging middleware and the Spring messaging framework, where they also adapted the OpenTracing (https://opentracing.io accessed on 1 January 2023) structure by incorporating new tags and log pairs. ARCHI4MOM adds new logs that not only help identify messaging spans but also adjust them before and after the trace reconstruction process.

Bushong et al. [41,42] implemented Prophet, a tool specifically designed for analyzing Java projects using the Spring framework. It provides a technique for automatically generating context maps (as part of the domain model) and SDG for microservice projects using static code analysis. The context map is created in two parts, first by creating the bounded context of each microservice project and then combining them into a map for the entire system. The SDG is generated by analyzing the code again to identify HTTP calls among microservices. This includes identifying the API endpoints of each service and where they are called from other services. This information is then used to create a graph showing the communication paths between the services.

Schiewe et al. [29] proposed a novel static analysis technique that provides a unified interface. It extracts a broad range of component types that can scale across various platforms. It constructed the LAAST representation from the source code. After that, a set of generalized parsers was introduced to detect specific component types. These parsers can be system-specific parsers to better cope with platform differences. This approach was evaluated in a case study involving two large, heterogeneous, cloud-native system benchmarks. The study demonstrated a unified identification approach to determine system data entities and endpoints. It shows strong potential to transform static code analysis practices by operating with component types rather than low-level programming constructs.

There are various software network extractors [28]; however, in most cases, they are tied to a particular programming language and extract only one type of software network. For instance, Murphy et al. [43] presented extractors for CFG from the C programming language.

4.3. Conclusion and RQ1 Answer

The analysis process aims to extract a holistic architecture from the distributed microservices to allow reasoning about the system as a whole. The microservice-based systems emphasize the importance of considering the multiple perspectives of service, interaction, and infrastructure. Therefore, static and dynamic analysis are complementary techniques, the dynamic analysis reveals subtle defects or vulnerabilities whose cause is complex to be discovered by static analysis; therefore, it requires the system to be deployed first.

The static analysis examines all possible execution paths—not only those invoked during execution. Although microservices allow the use of different languages and technologies, the distribution and heterogeneity are the most challenging characteristics of microservice-based analysis techniques. Facing these challenges requires an approach that comprises a generic way to retrieve the necessary static and dynamic data from different distributed microservices in addition to combining the data into a centric perspective [1].

In other words, a universal intermediate representation is needed across different languages and technologies. Although the static construction of this universal representation is more challenging, utilizing the techniques of component extraction can advance the current static analysis techniques to better face the development practices in a higher level perspective [28,29].

There are highlighted barriers in the proposed approaches that limit the applicability of the approaches as summarized in Table 2. The dependency on specific language features, such as the reflections in [6] and the interceptors for service calls in [1]. Furthermore, the dependency on external frameworks as Swagger integration in [6] and the data collection development library in [1].

On the other hand, the evaluation process for the proposed techniques lacked several details, such as the unavailability of enough testbenchs that share similar characteristics to real-life systems. This makes the approaches limited, such as the approach in [1] that was validated over a small number of microservices, which is not common in real-life systems. In [6], the approach was validated over randomly generated testbenches, which impacts the evaluation efficiency. In addition there were unclear and missing derived studies details, such as the study design type, setup details, participants, and tasks. Furthermore, the characteristics of the representation format affect the usability and interoperability of the produced intermediate views.

Utilizing this information leads to designing an approach to face the main mentioned challenges. It aims to construct a universal intermediate representation across different languages and technologies; and this is concluded in three steps as depicted in Figure 3 starting with extracting the AST per each microservice codebase. The second step is applying a mechanism to convert each language-related AST to a unified structure and then combining these distributed representations to represent the holistic system. This unified structure follows readable standard formats, such as JSON, XML, and SQL. This structure takes advantage of being human-interpreted and computer interpreted at the same time. The third step introduces extraction and filtration techniques; they extract components and different intermediate representations through this universal structure.

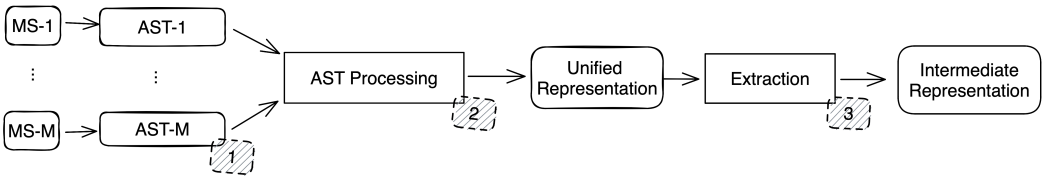

Figure 3. Architecture reconstruction designed approach.

Table 2. Architecture reconstruction challenges and solutions.

Challenges	Solutions
The microservices components and codebases distribution.	Integrating mechanisms for combining the distributed representation into a holistic one. For example, constructing the holistic domain model of the system from the bounded models as discussed in [18,44].
The development heterogeneity from one microservice to another.	Providing specific parsers for multiple languages and technologies specifications to analyze this diversity between services. Some research [28,29] address techniques for achieving that.
The dependency on language-specific features and external libraries. Such approaches depend on the interceptors and the Swagger library.	Finding a unified approach that works without that dependency, such as the CST and AST analysis approaches as mentioned in [29]. Another solution is to provide multiple techniques to support each of these operating dependencies.
The limited number of testbenches matches the microservices real-life characteristics.	Enriching the existing few testbenches to support the missing components and features, such as heterogeneity systems and different message queues mechanisms. In addition to generating testbenches to match real-life systems criteria.
The lack of understandability in the representation's format. A tool-specific format that is neither standard nor human interpreted.	Producing the representations in a standard format that is both human and computer interpreted; to be readable and easy to debug, apart from its applicability to integrate with different technologies.

5. Reasoning

The architecture reconstruction phase prepared overall representations of the system and its behavior. They can be used to understand the system architecture to reason about its properties [1] as shown in Figure 2. The reasoning phase processes and adapts these different representations to visualize the system perspectives for human-centered reasoning. Moreover, several strategies are applied to produce technical reasoning about the system performance aspects. The following subsections depict more details about these types of reasoning.

5.1. Technical Reasoning

Technical reasoning provides the capabilities to understand and evaluate the system's properties. Cross-service debugging and monitoring are challenging tasks. Thus, providing an ability to assist in debugging and reasoning about different system attributes could improve the efficiency and maintainability of these microservice-based systems [6].

As shown in Figure 2, the architecture reconstruction phase converts the artifacts of the system into proper intermediate representations. Then, the technical reasoning phase employs different strategies and techniques over the processed representations. These strategies use multiple metrics and practice catalogs (such as the Smells catalog) to evaluate the system's performance, such as quality attributes, testing, verification, and compliance.

5.1.1. Testing and Compliance

A system that lacks complete microservice testing is very fragile. Therefore, multiple testing levels are evaluated, including unit tests, service tests, end-to-end tests, and behavior-driven tests [6]. The unit testing is executed during the development phase; it ensures the functionality of a microservice works as required. The service level testing ensures successful service invocation as a single unit, while the end-to-end level testing considers the microservices integration. It validates whether the system acts as the expected

use cases. Finally, the behavior-driven testing examines the microservice system behavior in all anticipated situations [6].

Some contributions provide help in reasoning about the testing attributes; Ma et al. [6] proposed an approach that calculates the coverage of the service tests, apart from generating the missing tests using Pact (https://pact.io accessed on 1 January 2023). They introduced an approach to find incomplete test coverage of calls across an entire microservice mesh. When a service test case fails, the SIC that involves the failed invocation will be highlighted to help users to identify the error root. Moreover, Rakić et al. [28] suggested an extension to their approach to be used in an automatic test case generator. That could happen by generating a language-independent enriched Control Flow Graph (eCFG) by inserting branches that represent possible execution paths.

Das et al. [45] proposed a method that analyzes a set of microservice artifacts that communicate with each other through REST calls. It identifies RBAC violations for the whole microservice system by matching the security metadata of individual microservices with their REST communication flow. There are three modules included in the analyzer: a discovery module, a flow-matcher module, and an analysis module. The discovery module implements the extraction phase of SAR. It collects endpoint specification and security metadata. Next, the flow-matcher module performs the construction and manipulation phases of SAR by resolving the interaction among microservices. Finally, the analysis module completes the analysis phase of SAR and detects potential RBAC violations based on the other two modules' output.

There are other aspects, including verification, validation, and regulatory compliance. However, there are no adequate contributions for microservice-based systems. For example, evaluating regulatory compliance is challenging, such that considering the appropriate level of regulation considered necessary, when it should be applied, and how it is then enforced. Mashaly et al. [46] proposed a microservices architecture-based solution to comply with the GDPR (https://gdpr.eu accessed on 1 January 2023) principles.

5.1.2. Quality Attributes

Reasoning about the system quality attributes is concerned with several attributes of system behavior. Each quality attribute (such as performance, security, fault-tolerance, reusability, and maintainability) requires different strategies and techniques to evaluate the system accordingly. These strategies use multiple data sources (such as metrics and bad smell catalogs) to feed their decisions.

The metrics could be produced through static analysis-based representations, such as cyclomatic complexity (CC) and Lines of Code (LOC) [28], while dynamic analysis-based metrics, such as the response time, error rate, and workload; identify potential performance indicators, for example, it identifies scaled up or down for the used services [1]. Bad smells are indicators of multiple and different incidences, such as undesired patterns, antipatterns, and bad practices. They negatively affect the quality attributes, such as understandability, testability, extensibility, reusability, and maintainability of the system under development [2].

In addition to traditional smells, there are microservice-specific smells that are also problematic for the development and maintenance of microservice-based systems. Azadi et al. [47] proposed a catalog of microservices architectural smells, and they classified them based on modularity, hierarchy, and healthy dependency structure. These principles are defined as follows [47],

- *Modularity:* The system property complies with cohesive and loosely coupled building components.
- *Hierarchy:* The ranking of boundaries between the different layers of abstraction.
- *Healthy Dependency Structure:* Indication of the chain of changes that happen in the system each time it is modified.

Moreover, Taibi et al. [2] collected evidence of microservices bad practices by interviewing developers experienced with microservice-based systems. They provided a

catalog of rated bad smells together with possible solutions to overcome them. Five smells are summarized as the most mentioned during the interviews, as follows: Wrong Cuts, Hard Coded Endpoints, Shared Persistency, Cyclic Dependency, and Not Having an API Gateway. They are described and summarized in Table 3.

Table 3. Microservices smells summary.

Smells	Definition
Cyclic Dependency [2]	A cyclic chain of calls between microservices exists, e.g., A calls B, B calls C, and C calls back A.
Strong Coupling [1]	Strong communication relationships between microservices, such as services that are always deployed together.
Wrong Cuts [2]	Microservices are separated on a technical basis (such as presentation, logic, and data layers) instead of a business domain.
Hard Coded Endpoints [2]	Hardcoded IP addresses and ports are used for connecting microservices. Furthermore, referred to as Hardcoded IPs and Ports.
Shared Persistency [2]	Different microservices access the same relational database; however, it is worse when they share access to the same entities of the same relational database. Furthermore, this is proposed as Data Ownership.
Not Having an API Gateway [2]	Microservices communicate directly with each other. In the worst case, the client consumers communicate directly with microservices. This increases the complexity of the system and decreases its ease of maintenance, particularly when the number of microservices increases.
Code Clone [34,48]	A code fragment that has other code fragments identical or similar to it in the source code. The code clone can be syntactic or semantic.
Microservice Size [1,3]	A large microservice that cannot be further implemented by a single team. It should be split into multiple services.
Number of Interfaces [1]	A microservice that manages a large number of interfaces in the defined data model. It needs to be split into several services.

Some code smells are technically detectable; however, that requires employing the appropriate strategy and technique based on the smell type. Mayer et al. [1] utilized the reconstructed representation to automatically detect four smells, Strong Coupling, Cyclic Dependency, Microservice Size, and Number of Interfaces, see Table 3. Ma et al. [6] detected the cyclic dependency by applying Tarjan's Strongly Connected Component algorithm [49] to the extracted representation.

Several studies [2,6,47] considered that Cycle Dependency is the one more critical, such that it deeply affects software maintenance, and it causes unlimited service calls to make the system function improperly. However, Code Clone smell has a serious impact on the system maintainability and its code consistency. It appears by copying and pasting source code into different components, even with minor modifications [34].

Code clones are considered more critical in microservice-based and enterprise systems; as these systems tend to be repetitive in their distributed structure [34]. Several code clone detection techniques are proposed; however, there is a lack of them targeting enterprise and microservice-based systems. Svacina et al. [34] proposed a semantic clone detection technique for distributed systems. They use Java Reflection and Javassist to produce CFG.

Then, they applied a global similarity function on two arbitrary CFGs to detect the clone percentage between them.

On the other hand, Arcelli et al. [50] proposed an approach that aims to improve the performance of a software system by matching the system behavior at runtime with its architectural design. The idea is to connect performance-critical situations at runtime with their corresponding potential causes in design. This is achieved by creating traceability models that define the relationships between system design and runtime information in a consistent and exploitable way.

The authors generated traceability links between the UML of the system and the logs by using JTL (Janus Transformation Language) [51]. The JTL traceability engine can execute these transformations and automatically generate the corresponding traceability links between elements of the UML design model and the log model. The analysis of these traceability models can help to identify system property deviations and affected components.

Many more evaluation results are produced in the technical reasoning phase. However, these results need to be reported and visualized properly. The appropriate visualization can speed up comprehension of the reconstructed representation and results. The below human-centered reasoning subsection discusses this visualization-related perspective and its challenges.

5.2. Human-Centered Reasoning (Visualization)

The automatic documentation of a microservice architecture is one of the main purposes of the architecture reconstruction [1]. The architecture reconstruction through static and dynamic analysis produces detailed information on individual services and the overall system architecture. Both the architecture representations and the technical reasoning outcomes can be documented, reported, and visualized, as illustrated in Figure 2. Many means can be used to articulate the system architecture and its behavior to stakeholders, such as architects, developers, and DevOps [44].

Many investigations have been conducted in the direction of utilizing the current visualization methodologies to render the microservice-based systems [9]. These approaches include notation-based UML modeling (such as Class diagram, Package diagram, Interaction diagram, and Activity diagram), ArchiMate (https://pubs.opengroup.org/architecture/archimate3-doc accessed on 1 January 2023) that visualizes enterprise business processes, and Metaphors (such as the 3D city metaphor [52] and island metaphor [53] as shown in Figure 4). Furthermore, the graph representation is strongly involved in rendering the dependency and interaction perspectives.

The authors in [1,6,54] proposed graph-based visualization tools that render the SDG for microservice-based systems. These representations lack vital features for microservice-based systems, such as service interactions and dependency, and runtime behavior. The missing information in these approaches alongside the limitations of the rendering space scalability promoted researchers to propose new approaches for visualizing microservice-based architectures. Virtual Reality (VR) and Augmented Reality (AR) techniques are employed to overcome the mentioned limitations, particularly the rendering space and the navigation into large systems.

Oberhauser et al. [55] employed VR capability to demonstrate enterprise architecture models. Cerny et al. [44] proposed a tailored AR-based visualization for microservices SDG representations. They analyzed microservice-based systems using a static analysis technique, and then they implemented a Microvision tool that rendered the constructed SDG in the AR medium as shown in Figure 5.

Furthermore, microservices still have limited scientific publications covering this area of providing the appropriate visualization to reason about the system perspectives. The gray literature may hold valuable insights [9,56]. Netflix (https://simianviz.surge.sh/netflix accessed on 1 January 2023) demonstrates an interactive visualization for the SDG of their system as shown in Figure 6. Amazon (X-Ray: https://docs.aws.amazon.com/xray/latest/

devguide/xray-console.html accessed on 1 January 2023) provides a solution called an X-Ray console. It is a visual map that depicts the lifecycle of the request.

Figure 4. Island metaphore [53].

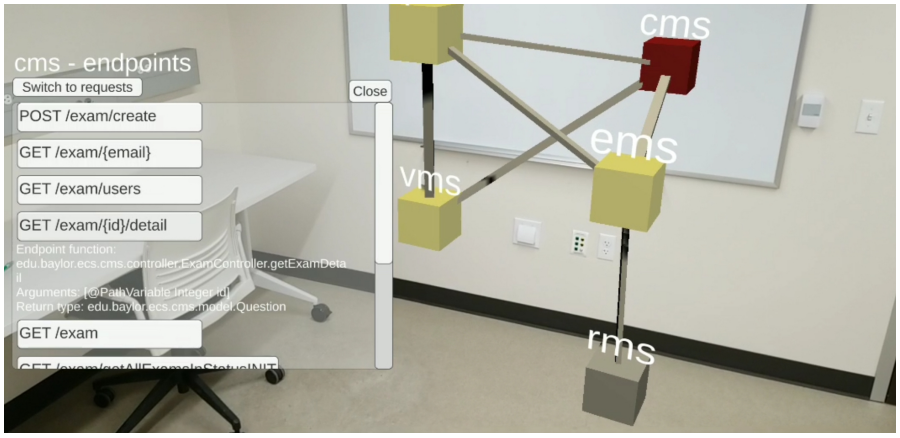

Figure 5. Microvision [44].

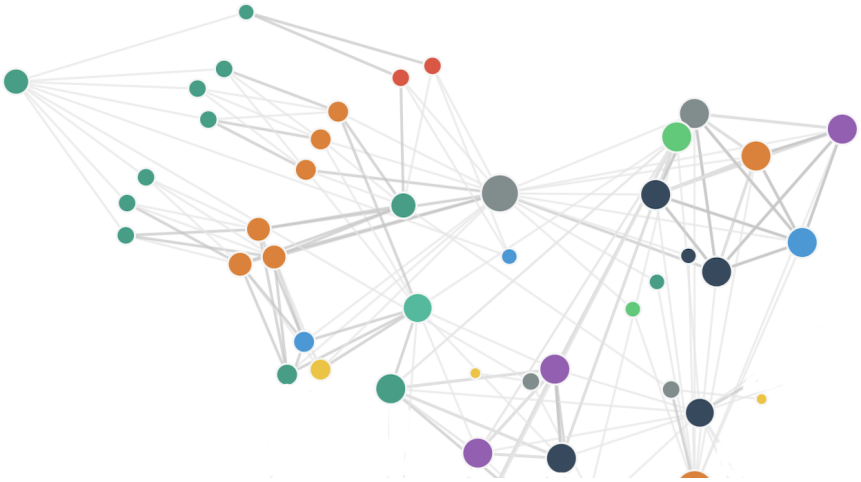

Figure 6. Netflix SDG visualization (https://simianviz.surge.sh/netflix accessed on 1 January 2023).

5.3. Conclusion and RQ2 Answer

Reasoning about the system perspectives involves technical and human-centered approaches. The technical reasoning is concerned with inspecting the architecture to evaluate it and identify design problems [1]. Evaluating the system corresponding to a specific perspective needs a tailored technique and an appropriate data source containing the guidelines of this perspective. Despite the mentioned catalogs for microservices bad smells, there is a lack of standardization [47], and different names are often used to indicate the same problem.

Plugging in the appropriate visualization to the process shown in Figure 2 fuels more advances in the system features. Visualizing the evaluation results is a vital task; however, it is complex, particularly for the perspectives that require overall system visualization, such as tracking the change impact, and debugging a runtime error. Moreover, human-centered reasoning is established on visualization methodologies to investigate the different aspects of the system.

For example, experiencing a multiple responsibility function raises the question in turn of whether the identified functionality should be moved to another service or implemented as a new service. The visualization of SDG and technology stack could help the architect to decide, such that it is difficult to move functions between services implemented with different technologies [1].

The visual space is a critical challenge for visualizing microservices architectures [9]. In addition to the lack of consideration to the microservices characteristics and perspectives, such as visualizing services interactions and dependency, and the runtime behavior. [44]. The challenges and their emerging solutions are concluded in Table 4.

Sketching an overall approach for addressing these challenges decomposes into three parts as highlighted in Figure 7: standard data catalogs, tailored algorithms, and visualization. First, the data catalogs contain the following: aspect standards that are used in the industry, the mechanisms used for detecting their related vulnerabilities, and the intermediate representations that are employed for extracting these aspects. Secondly, implementing a tailored algorithm that follows the catalog specifications to detect specific aspects of the system. Finally, the tailored visualization technique satisfies the following:

- Integrating the visualization with common software tools, such as integrated development environment (IDE), source control, and task management system.
- Completeness to visualize the different microservices representations. In addition to including the information required for each of these representations.
- Scalable to enable rendering large microservice-based systems.
- Supporting runtime perspectives, such as debugging and monitoring request interactions.
- Interactive and allows the navigation between different views of the system.
- Providing multiple modes, such as the design mode helps in demonstrating specific changes during the analysis and design phases.
- Aggregating multiple layers to blend the technical evaluation results with the several representations.

Thus, the reasoning-based approaches are still open to supporting the missing features and considering the different perspectives of the microservice-based systems.

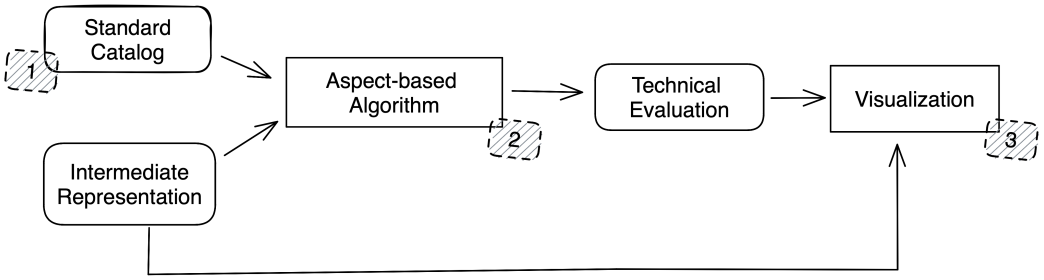

Figure 7. Reasoning designed approach.

Table 4. Reasoning challenges and solutions.

Challenges	Solutions
Technical Reasoning	
The lack of microservice technical aspects standardization. In addition to using different names for describing the same aspect. That makes it challenging to identify the violation specifications.	Creating a combined catalog contains the smells related to microservices in a unified naming convention. This approach should be generalized to include the different reasoning aspects, such as security and compliance. The mentioned catalogs in [2,47] show promising approaches for building a standard one.
The scarcity of detection techniques to cover microservice-based system aspects.	Integrating the technical aspects and the recommended detection mechanisms of each smell in a standard catalog. That also requires deriving industrial surveys; it categorizes the technical aspects with the industrial tools used to detect them as shown in [57,58].
Human-Centered (Visualization)	
The limited 2D rendering space to visualize and reason about large systems.	Employing 3D render space to visualize scalable systems. It facilitates the integration with different technologies (such as AR and VR) to demonstrate more advanced features as shown in [44]. Furthermore, utilizing the 2D space for showing specific representations that do not involve whole system representation.
The incompleteness to visualize microservices-related aspects. It lacks a tailored and complete visualization tool for microservice-based systems.	Designing a visualization approach to support multiple modes. It should be capable of embracing static and dynamic representations of the system, including their perspectives of service, interaction, and infrastructure. In addition to prioritizing the information based on the three roles (architect, software developer, and DevOps engineer) point-of-view. It should involve several research collaborations and user studies; it could build on top of existing promising approaches, such as those mentioned in [44,55].
The poor format to visualize the technical assessment outcome.	Visualizing the technical evaluation results on top of suitable intermediate representations. This utilizes the graphical representation for the system to reason about the technical evaluation results. In addition to supporting suitable reporting formats (such as JSON and XML) to enable the user to interpret, analyze, and integrate them with different tools. As an instance, the authors in [59] propose graphical representations of some architectural anti-patterns.

6. System Evolution

The agility of microservice-based systems, which results from the independent development and integration of new services, leads to continuous architectural changes [3]. The evolution of microservices architecture happens during development and migration lifecycles. The migration process emphasizes the early appearance of the evolutionary perspective in microservice-based systems. It starts with splitting a monolith system into microservices by identifying independent business processes that can be isolated [2]. Moreover, microservice-based systems have both static and dynamic aspects of evolution.

The continuous development of the system could require moving functionality from one service to another. However, different services may use different technologies, thus, it requires the technology stack representation to support the design process [1]. Moreover, adding a new service, a new version, and moving functionality between services impact other services that use this functionality. Therefore, the SDG representation is needed to highlight these dependencies and track the linkage changes between services in an early

stage [6]. This helps to analyze the impact of such changes and also identify any necessary adaptations to the other system components.

Regarding the dynamic evolution in runtime, service instances are deployed and removed frequently and independently from other services [1]. The captured runtime metrics (such as the response time and error rate) help to determine the success of architectural changes and identify potential side effects. Therefore, tracking the dynamic data over time ensures the health of the system. Furthermore, the infrastructure evolution is tracked by collecting information about the status of containers or service instances over time [1]. This helps in the deployment distributions, such that a poor deployment choice can increase cost and can decrease performance, scalability, and fault tolerance. Therefore, these decisions must be re-evaluated as the system evolves [60].

Microservices offer extensive deployment flexibility. For example, two services can be co-located as two containers on the same machine, as two containers in one VM, or as two VMs on the same machine. A poor deployment choice can increase cost and can hurt performance, scalability, and fault tolerance. Furthermore, these decisions must be re-evaluated as the system evolves. Today developers evaluate changing deployment trade-offs through trial and error without a systematic strategy nor much tool support.

That emphasizes the urge to continuously extract and reconstruct the system architecture. The microservices evolution process is depicted in Figure 8, it is triggered through either static or dynamic change in at least a single microservice. Then, it fires an event to re-execute the architecture reconstruction phase and highlight the different representation changes. After that, the reasoning techniques are re-executed to include these highlighted changes. Therefore, they are included in the system visualization for human-centered reasoning.

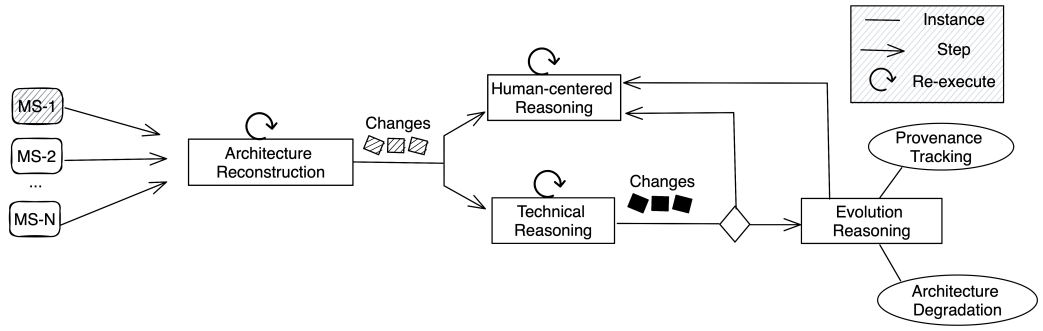

Figure 8. Microservices evolution process.

Furthermore, they are re-evaluated to produce system evaluation changes. Finally, the evolution reasoning applies change impact analysis techniques to evaluate the system performance changes in considerable aspects. For example, provenance tracking [61] records the changes that happen in the architecture over time. In addition, evaluating the degradation highlights the erosion that is indicated in the architecture over changes. These evaluation results are reported to attain human-centered reasoning. Applying this process ensures the ongoing evaluation of the system's performance from its different perspectives.

Existing architecture reconstruction approaches often provide snapshots of the current architecture, without considering the evolution of microservices over time [1,62]. That needs interpretation by the developer for reconciling the deployed version of the system with the historical snapshots [60]. That indicates continuous challenges for developers in several perspectives, such as determining compatibility and consistency between microservice versions, identifying architectural improvements and degradation, and evaluating changing deployment topology [60].

Rakić et al. [28] developed a software tool that tracks and analyzes changes in the structure of a program. It takes eCSTs as its input and produces the set of changes in the program structure. Arbuckle [63] presented an approach that uses source code changes produced from software repositories to analyze software evolution.

For the microservices-related approaches, Apolinário et al. [64] demonstrated a strategy for evaluating a suite of coupling metrics over time to identify signs of architectural degradation in microservice-based architectures. To achieve this, the authors proposed the SYMBIOTE method to monitor coupling in microservice-based architectures. The method uses dynamic analysis techniques to construct a Service Dependency Graph (SDG) from logs and then applies a set of metrics to measure the coupling property in the system.

Sampaio et al. [60] proposed an approach for combining and tracking microservices structural, deployment, and runtime information. It considers three layers, an architectural layer that captures the service level information. The instance layer captures information about the interaction level, including replicas and upgrades, in addition to the flow of calls and messages between services. The infrastructure layer captures the infrastructure specs and deployment attributes. It keeps track of services and endpoint versions. To optimize the deployment topology, it periodically monitors and stores metrics, such as the CPU load, memory utilization, traffic, and latency of requests.

Conclusion and RQ3 Answer

Microservices are developed, deployed, and evolve independently, however, the evolution process targets keeping the system consistent, coherent, and functional over changes [60]. The evolution tracking provides a high-level overview of the system, furthermore, it validates the conformance of the emerging runtime architecture with the originally planned architecture [1].

As implied in Figure 8 and listed in Table 5, the evolution process challenges include the prior mentioned related challenges to architecture reconstruction, visualization, and technical reasoning processes in addition to the lack of microservices-tailored approaches to detect and deal with the architectural changes data. For instance, there are not enough approaches to address and visualize the microservices architecture provenance tracking [61]. Moreover, continuous evaluation of entire system perspectives is challenging; it requires several techniques to handle both. Whatever the considered perspectives would help the system to keep healthy accordingly.

Applying the evolution through all steps as shown in Figure 8, means every stage seeks different techniques for detecting the impact that happened in their components accordingly. Diving deeper into the three steps required to perform an evolution technique as illustrated in Figure 9. It shows a comparison mechanism working over the intermediate representations and the evolution results and then embedding the visualization with features to show different historical snapshots of the representations in addition to highlighting the changes and their impact on the system. On the other hand, the evolution techniques could function in two different modes: detection and prevention. The detection mode creates alerts and shows the impact of the change, such as architecture smell detection, while the second mode could prevent the impact that occurs as a result of the changes, such as security vulnerability prevention.

Finally, applying the illustrated process in Figure 8 would alert practitioners to reason about the system in an early stage during development and while the system evolves. Thus, they can avoid bad practices before their effect is rippled into different parts of the system and affect its quality attributes. Furthermore, recommendation techniques could be embedded in the process to deliver helpful information to users. In addition to supporting the system to make conclusions based on calculated metric values for improving the system quality.

Table 5. Evolution challenges and solutions.

Challenges	Solutions
The lack of microservices-tailored approaches for architecture provenance tracking [61]. It is challenging to detect and represent architectural changes automatically.	Employing algorithms to compare architecture components. That could happen in different levels of granularity—for example, the comparison mentioned for semantic clone detection in [34].
The shortage of visualization to depict and track the microservices architecture changes.	That includes two parts: format of changes and visualization. First, it represents the changes in a standard format for the best interpretation and integration with different tools. Second, the visualization technique enables the rendering of the changes and their impact on the system representations as highlighted in [44,59].
The evolution techniques require the re-execution of multiple techniques during system changes. The evolution process is challenged considering the discussed difficulties related to providing each technique of the following: - Architecture reconstruction. - Visualization. - Technical reasoning process.	Specifying the required scope and changes to track and their impact to analyze in the system. The evolution techniques could perform corresponding actions to some chosen aspects of the system.

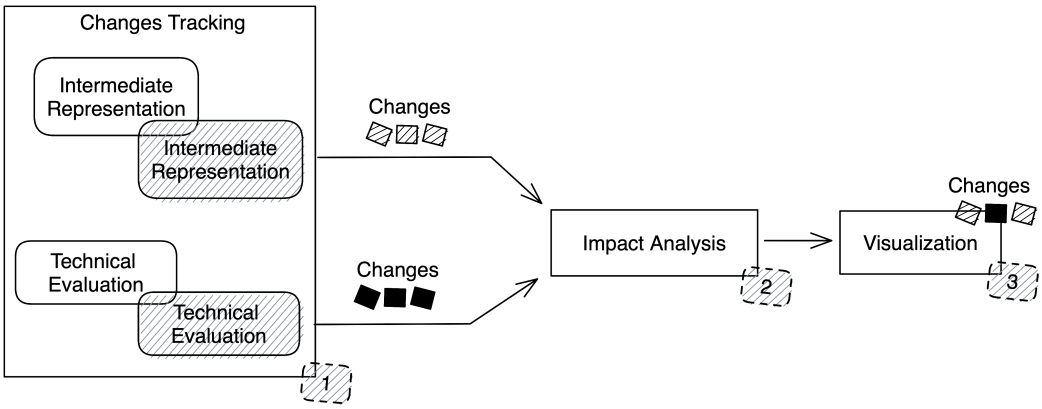

Figure 9. Evolution-designed approach.

7. Threats to Validity

To ensure the credibility and reliability of our study, we addressed several potential validity issues that may impact the quality of our results and conclusions. Using Wohlin's taxonomy [65], we examined four types of validity threats: external validity, concept validity, internal validity, and conclusion validity.

7.1. Construct Validity

Construct validity refers to the degree to which the study's research questions and design accurately measure the intended construct or concept. In this study, we attempted to address potential construct validity threats by carefully selecting search terms and conducting a pilot search to ensure the relevance of the results. Multiple combinations of search queries were attempted before finalizing the search terms in order to include all important keywords related to our research.

The study's limitations, such as using only Scopus for the initial search and the possibility of omitting important research during the filtration process, could potentially compromise the validity of the study. To address this, we aimed to achieve the goal of

outlining the reasoning process, which guided the review rather than solely relying on the search results. This approach helped to fill in any gaps that were missed during the initial search process. Additionally, we expanded our snowballing process to include a variety of sources, including white and grey literature, which may contain unique perspectives and ideas related to the topic of the study.

7.2. Internal Validity

Internal validity assesses whether the methods used to gather and analyze data are sound. To ensure that all relevant articles on the chosen topic were included in the review, we examined a limited set of publications through our initial search query. However, we also relied on snowballing and our expertise to guide the selection of papers. In terms of reliability and repeatability, we clearly defined our search keywords and used synonyms to allow for the replication of the literature search. However, replicating the specific structure of the review may be challenging as it is influenced by our approach and expertise in the field. Nevertheless, by following the provided information on the research design and focusing on answering the research questions parts, it is feasible to replicate the essential elements of the roadmap.

Another potential concern is related to the thoroughness of the papers included and excluded in the study. The inclusion and exclusion criteria for papers in this study were carefully considered during the screening, reading, and discussion process. The criteria for inclusion and exclusion were clearly described, and the dataset used for the filtration process was provided. Additionally, the data that was extracted from the papers was also discussed.

To address concerns about bias, we focused on three key areas: data extraction, categorization, and interpretation of the results. During the data extraction process, we divided the responsibility among two authors to ensure that all relevant information was captured. Additionally, we sought input and feedback from three external experts in the field to further validate the accuracy and completeness of the data.

The categorization of papers and information was also thoroughly reviewed and discussed, with input from the two authors and three external experts. This process involved multiple rounds of comments, modifications, and expansions, with the goal of creating a comprehensive and accurate categorization. Finally, we were mindful of potential bias in the interpretation of the results and conclusions and sought to minimize this by obtaining feedback from a diverse group of experts in the field. By taking these steps, we aimed to minimize the potential for bias and ensure the accuracy of the data throughout the study.

7.3. External Validity

The external validity concerns the ability to extend our findings and conclusions to other contexts and settings. To address this, we made a concerted effort to gather data from a diverse range of sources and databases, particularly during the snowballing process and discussion sessions. Our findings and observations pertain specifically to the microservices reasoning roadmap; however, the method used in this study can be broken down into three distinct parts (architecture reconstruction, reasoning, and evolution) and can be generalized to these specific areas. However, it is important to note that our categorization of the publications was based on the specific problem of microservices reasoning and may not be directly applicable to other contexts. Despite this limitation, the overall roadmap overview can be generalized as it is informed by expert knowledge and not solely dependent on the results of the papers, which could be influenced by missing some related work.

7.4. Conclusion Validity

To address the potential for conclusion validity, we took several steps to reduce bias and to ensure that our conclusions were based on the available evidence. Through multiple brainstorming sessions between the two authors, we aimed to minimize author bias. We

also sought input from other experts in the field through discussion sessions and ensured that the participants were all aware of the conclusions before discussing them in groups. Furthermore, we considered the quality and rating of the references used in the study to minimize the potential for interpretation bias. To imitate the influence of this threat as possible, we emphasized the consensus of the findings and the conclusions regarding the results.

8. Conclusions

Microservice-based systems are usually complex systems with many decentralized parts that are self-contained and evolved by independent teams. When we need to understand various perspectives, aspects, or concerns of these systems, we typically look for information cross-cutting many microservices. This, however, requires significant effort since we need to analyze multiple changing parts and understand the dependencies among them. In system reasoning, we look to find various answers related to system design, evolving architecture, or other often-distributed knowledge about these systems.

In this paper, we reviewed and connected multiple perspectives of the reasoning process in microservice systems to draw a holistic roadmap for the process. As a result, we identified multiple stages of the reasoning process and investigated each stage individually.

The core of the reasoning process centers around microservice system architecture reconstruction. Different techniques and solutions are detailed and discussed. This reconstruction stage results in generating holistic intermediate representations. The architecture reconstruction process lacks mature universal representations that deal with heterogeneous microservices, considering both static and dynamic perspectives.

With system intermediate representation, there are various reasoning approaches to consider. The reasoning strategies utilize the intermediate representations in two ways. *First* is integrating them with catalogs of metrics, patterns, or rules to generate evaluation measurements in the technical-based reasoning. Nevertheless, this lacks tailored techniques and consistently described metrics catalogs and, therefore, empirical validation for the existing catalogs. Furthermore, a deeper investigation is needed to evaluate the harmfulness and the comprehensiveness of the smells catalogs.

Second, visualizing these representations promotes the human-centered perspective. Human-centered reasoning requires microservices-tailored visualization mechanisms that embrace different architectural views and challenge the rendering space limitations.

The perspective of system evolution in the context of the reasoning process adds another dimension to the process. This aims at revealing static and dynamic changes and their impact on system parts. Such a process should produce multiple evaluation measurements on the system to highlight possible architectural degradation. However, it suffers from the rarity of automated techniques for identifying and recording architecture changes. As well, many challenges are open for investigations to ensure the quality of the system over time.

Future Work

Individual microservices form the whole system; however, each microservice typically evolves independently, which introduces vulnerability in the form of a ripple effect impacting other system parts outside of the bounded context of a given microservice. Therefore, future work will target a change impact analysis method to assess the evolution of microservices to prevent ripple effect implications and possible architectural erosion.

Using the intermediate representation of the whole microservices system, it will include a technical perspective indicating code quality changes, as well as a human-centered perspective, enabling developers to visualize the change, its impact, and implications across the system and other system parts. Moreover, the system is becoming more complex over changes, it can be seen as beneficial to understand and track this architectural evolution. Thus, a provenance-tracking method could help enrich the capabilities to control and resolve the microservice architecture during the various development activities.

Author Contributions: Conceptualization, A.S.A.; methodology, A.S.A. and T.C., validation, T.C.; formal analysis, A.S.A.; investigation, A.S.A. and T.C.; resources, A.S.A.; data curation, A.S.A.; writing—original draft preparation, A.S.A. and T.C.; writing—review and editing, A.S.A. and T.C.; visualization, A.S.A. supervision, T.C.; project administration, T.C.; funding acquisition, T.C. All authors have read and agreed to the published version of the manuscript.

Funding: This material is based upon work supported by the National Science Foundation under Grant No. 1854049 and a grant from Red Hat Research (accessed on 30 January 2023).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data available in a publicly accessible repository.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Mayer, B.; Weinreich, R. An approach to extract the architecture of microservice-based software systems. In Proceedings of the 2018 IEEE symposium on service-oriented system engineering (SOSE), Bamberg, Germany, 26–29 March 2018; pp. 21–30.
2. Taibi, D.; Lenarduzzi, V. On the Definition of Microservice Bad Smells. *IEEE Softw.* **2018**, *35*, 56–62. [CrossRef]
3. Wolff, E. *Microservices: Flexible Software Architecture*; Addison-Wesley Professional: San Francisco, CA, USA, 2016.
4. Buchgeher, G.; Winterer, M.; Weinreich, R.; Luger, J.; Wingelhofer, R.; Aistleitner, M. Microservices in a small development organization. In Proceedings of the European Conference on Software Architecture, Canterbury, UK, 11–15 September 2017; Springer: Berlin/Heidelberg, Germany, 2017; pp. 208–215.
5. Singleton, A. The Economics of Microservices. *IEEE Cloud Comput.* **2016**, *3*, 16–20. [CrossRef]
6. Ma, S.P.; Fan, C.Y.; Chuang, Y.; Lee, W.T.; Lee, S.J.; Hsueh, N.L. Using service dependency graph to analyze and test microservices. In Proceedings of the 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), Tokyo, Japan, 23–27 July 2018; Volume 2, pp. 81–86.
7. Familiar, B. What Is a Microservice? *Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions*; Apress: Berkeley, CA, USA, 2015; pp. 9–19. [CrossRef]
8. Wos, L.; Overbeck, R.; Lusk, E.; Boyle, J. *Automated Reasoning: Introduction and Applications*; Prentice-Hall, Inc.: Englewood Cliffs, NJ, USA, 1992.
9. Abdelfattah, A.S. Microservices-Based Systems Visualization: Student Research Abstract. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, 25–29 April 2022; Association for Computing Machinery: New York, NY, USA, 2022; SAC '22, pp. 1460–1464. [CrossRef]
10. Pautasso, C.; Wilde, E. RESTful Web Services: Principles, Patterns, Emerging Technologies. In Proceedings of the 19th International Conference on World Wide Web, Raleigh, NC, USA, 26–30 April 2010; Association for Computing Machinery: New York, NY, USA, 2010; WWW '10, pp. 1359–1360. [CrossRef]
11. Vural, H.; Koyuncu, M.; Guney, S. A systematic literature review on microservices. In Proceedings of the International Conference on Computational Science and Its Applications, Trieste, Italy, 3–6 July 2017; Springer: Berlin/Heidelberg, Germany, 2017; pp. 203–217.
12. Cartaxo, B.; Pinto, G.; Soares, S. The role of rapid reviews in supporting decision-making in software engineering practice. In Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018, Christchurch, New Zealand, 28–29 June 2018; pp. 24–34.
13. Ponce, F.; Márquez, G.; Astudillo, H. Migrating from monolithic architecture to microservices: A Rapid Review. In Proceedings of the 2019 38th International Conference of the Chilean Computer Science Society (SCCC), Concepcion, Chile, 4–9 November 2019; pp. 1–7. [CrossRef]
14. Cartaxo, B.; Pinto, G.; Ribeiro, D.; Kamei, F.; Santos, R.E.; da Silva, F.Q.; Soares, S. Using q&a websites as a method for assessing systematic reviews. In Proceedings of the 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), Buenos Aires, Argentina, 20–28 May 2017; pp. 238–242.
15. Hassler, E.; Carver, J.C.; Kraft, N.A.; Hale, D. Outcomes of a community workshop to identify and rank barriers to the systematic literature review process. In Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, London, UK, 13–14 May 2014; pp. 1–10.
16. Santos, R.E.; Da Silva, F.Q. Motivation to perform systematic reviews and their impact on software engineering practice. In Proceedings of the 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 10–11 October 2013; pp. 292–295.

17. Rademacher, F.; Sachweh, S.; Zündorf, A. A Modeling Method for Systematic Architecture Reconstruction of Microservice-Based Software Systems. In Proceedings of the Enterprise, Business-Process and Information Systems Modeling, Grenoble, France, 8–9 June 2020; Nurcan, S., Reinhartz-Berger, I., Soffer, P., Zdravkovic, J., Eds.; Springer International Publishing: Cham, Switzerland, 2020; pp. 311–326.
18. Cerny, T.; Abdelfattah, A.S.; Bushong, V.; Al Maruf, A.; Taibi, D. Microservice Architecture Reconstruction and Visualization Techniques: A Review. In Proceedings of the 2022 IEEE International Conference on Service-Oriented System Engineering (SOSE), Newark, CA, USA, 15–18 August 2022; pp. 39–48. [CrossRef]
19. Carnell, J.; Sánchez, I.H. *Spring Microservices in Action*, 2nd ed.; Manning Publications Co.: Shelter Island, NY, USA, 2021; p. 448.
20. Wiggins, A. The Twelve-Factor App. 2017. Available online: https://12factor.net (accessed on 1 January 2023).
21. Neamtiu, I.; Foster, J.S.; Hicks, M. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. *SIGSOFT Softw. Eng. Notes* **2005**, *30*, 1–5. [CrossRef]
22. Weinreich, R.; Miesbauer, C.; Buchgeher, G.; Kriechbaum, T. Extracting and Facilitating Architecture in Service-Oriented Software Systems. In Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, Helsinki, Finland, 20–24 August 2012; pp. 81–90. [CrossRef]
23. Granchelli, G.; Cardarelli, M.; Di Francesco, P.; Malavolta, I.; Iovino, L.; Di Salle, A. Towards Recovering the Software Architecture of Microservice-Based Systems. In Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden, 5–7 April 2017; pp. 46–53. [CrossRef]
24. Savić, M.; Ivanović, M.; Radovanović, M. Characteristics of class collaboration networks in large Java software projects. *Inf. Technol. Control* **2011**, *40*, 48–58. [CrossRef]
25. Wen, L.; Dromey, R.G.; Kirk, D. Software Engineering and Scale-Free Networks*. *IEEE Trans. Syst. Man Cybern. Part B (Cybernetics)* **2009**, *39*, 845–854. [CrossRef]
26. Wheeldon, R.; Counsell, S. Power law distributions in class relationships. In Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation, Amsterdam, The Netherlands, 26–27 September 2003; pp. 45–54. [CrossRef]
27. Puppin, D.; Silvestri, F. The Social Network of Java Classes. In Proceedings of the 2006 ACM Symposium on Applied Computing, Dijon, France, 23–27 April 2006; Association for Computing Machinery: New York, NY, USA, 2006; SAC '06, pp. 1409–1413. [CrossRef]
28. Rakić, G.; Budimac, Z.; Savić, M. Language independent framework for static code analysis. In Proceedings of the sixth Balkan Conference in Informatics, Thessaloniki, Greece, 19–21 September 2013; pp. 236–243.
29. Schiewe, M.; Curtis, J.; Bushong, V.; Cerny, T. Advancing Static Code Analysis With Language-Agnostic Component Identification. *IEEE Access* **2022**, *10*, 30743–30761. [CrossRef]
30. Rakić, G.; Budimac, Z. Introducing enriched concrete syntax trees. *arXiv* **2013**, arXiv:1310.0802.
31. Newman, S. *Building Microservices*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2021.
32. Hyland-Wood, D.; Carrington, D.; Kaplan, S. Scale-free nature of Java software package, class and method collaboration graphs. In Proceedings of the fifth International Symposium on Empirical Software Engineering, Rio de Janeiro, Brazil, 21–22 September 2006.
33. Myers, C.R. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Phys. Rev. E* **2003**, *68*, 046116. [CrossRef] [PubMed]
34. Svacina, J.; Simmons, J.; Cerny, T. Semantic Code Clone Detection for Enterprise Applications. In Proceedings of the 35th Annual ACM Symposium on Applied Computing, Brno, Czech Republic, 30 March–3 April 2020; Association for Computing Machinery: New York, NY, USA, 2020; SAC '20, pp. 129–131. [CrossRef]
35. Parsons, T.; Mos, A.; Trofin, M.; Gschwind, T.; Murphy, J. Extracting Interactions in Component-Based Systems. *IEEE Trans. Softw. Eng.* **2008**, *34*, 783–799. [CrossRef]
36. Francesco, P.D.; Malavolta, I.; Lago, P. Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. In Proceedings of the 2017 IEEE International Conference on Software Architecture (ICSA), Gothenburg, Sweden, 3–7 April 2017; pp. 21–30. [CrossRef]
37. Mayer, B.; Weinreich, R. A Dashboard for Microservice Monitoring and Management. In Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden, 5–7 April 2017; pp. 66–69. [CrossRef]
38. Ishida, A.O.R.; Kontogiannis, K.; Brealey, C. Extracting Micro Service Dependencies Using Log Analysis. In Proceedings of the 2022 IEEE 29th Annual Software Technology Conference (STC), Virtual Event, 3–6 October 2022; pp. 82–92. [CrossRef]
39. Richardson, M.; Domingos, P. Markov logic networks. *Mach. Learn.* **2006**, *62*, 107–136. [CrossRef]
40. Singh, S.; Werle, D.; Koziolek, A. ARCHI4MOM: Using Tracing Information to Extract the Architecture of Microservice-Based Systems from Message-Oriented Middleware. In Proceedings of the European Conference on Software Architecture, Prague, Czech Republic, 19–23 September 2022; Springer: Berlin/Heidelberg, Germany, 2022; pp. 189–204.
41. Bushong, V.; Das, D.; Cernỳ, T. Reconstructing the Holistic Architecture of Microservice Systems using Static Analysis. In Proceedings of the CLOSER, Online Streaming, 27–29 April 2022; pp. 149–157.

42. Bushong, V.; Das, D.; Al Maruf, A.; Cerny, T. Using Static Analysis to Address Microservice Architecture Reconstruction. In Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne, Australia, 15–19 November 2021; pp. 1199–1201.
43. Murphy, G.C.; Notkin, D.; Griswold, W.G.; Lan, E.S. An Empirical Study of Static Call Graph Extractors. *ACM Trans. Softw. Eng. Methodol.* **1998**, *7*, 158–191. [CrossRef]
44. Cerny, T.; Abdelfattah, A.S.; Bushong, V.; Al Maruf, A.; Taibi, D. Microvision: Static analysis-based approach to visualizing microservices in augmented reality. In Proceedings of the 2022 IEEE International Conference on Service-Oriented System Engineering (SOSE), Newark, CA, USA, 15–18 August 2022; pp. 49–58. [CrossRef]
45. Das, D.; Walker, A.; Bushong, V.; Svacina, J.; Cerny, T.; Matyas, V. On automated RBAC assessment by constructing a centralized perspective for microservice mesh. *PeerJ Comput. Sci.* **2021**, *7*, e376. [CrossRef] [PubMed]
46. Mashaly, B.; Selim, S.; Yousef, A.H.; Fouad, K.M. Privacy by Design: A Microservices-Based Software Architecture Approach. In Proceedings of the 2022 second International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC), Cairo, Egypt, 8–9 May 2022; pp. 357–364. [CrossRef]
47. Azadi, U.; Arcelli Fontana, F.; Taibi, D. Architectural Smells Detected by Tools: A Catalogue Proposal. In Proceedings of the 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), Montreal, QC, Canada, 26–27 May 2019; pp. 88–97. [CrossRef]
48. Higo, Y.; Kusumoto, S.; Inoue, K. A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system. *J. Softw. Maint. Evol. Res. Pract.* **2008**, *20*, 435–461. [CrossRef]
49. Tarjan, R. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* **1972**, *1*, 146–160. [CrossRef]
50. Arcelli, D.; Cortellessa, V.; Di Pompeo, D.; Eramo, R.; Tucci, M. Exploiting Architecture/Runtime Model-Driven Traceability for Performance Improvement. In Proceedings of the 2019 IEEE International Conference on Software Architecture (ICSA), Hamburg, Germany, 25–29 March 2019; pp. 81–90. [CrossRef]
51. Cicchetti, A.; Di Ruscio, D.; Eramo, R.; Pierantonio, A. JTL: A bidirectional and change propagating transformation language. In Proceedings of the International Conference on Software Language Engineering, Braga, Portugal, 3–4 July 2011; Springer: Berlin/Heidelberg, Germany, 2011; pp. 183–202.
52. Wettel, R.; Lanza, M. Visually Localizing Design Problems with Disharmony Maps. In Proceedings of the Proceedings of the fourth ACM Symposium on Software Visualization, Ammersee, Germany, 16–17 September 2008; Association for Computing Machinery: New York, NY, USA, 2008; SoftVis '08, pp. 155–164. [CrossRef]
53. Schreiber, A.; Nafeie, L.; Baranowski, A.; Seipel, P.; Misiak, M. Visualization of Software Architectures in Virtual Reality and Augmented Reality. In Proceedings of the 2019 IEEE Aerospace Conference, Big Sky, MT, USA, 2–9 March 2019; pp. 1–12. [CrossRef]
54. Nakazawa, R.; Ueda, T.; Enoki, M.; Horii, H. Visualization Tool for Designing Microservices with the Monolith-First Approach. In Proceedings of the 2018 IEEE Working Conference on Software Visualization (VISSOFT), Madrid, Spain, 24–25 September 2018; pp. 32–42. [CrossRef]
55. Oberhauser, R.; Pogolski, C. VR-EA: Virtual Reality Visualization of Enterprise Architecture Models with ArchiMate and BPMN. In Proceedings of the Business Modeling and Software Design, Lisbon, Portugal, 1–3 July 2019; Shishkov, B., Ed.; Springer International Publishing: Cham, Switzerland, 2019; pp. 170–187.
56. Bogner, J.; Fritzsch, J.; Wagner, S.; Zimmermann, A. Industry practices and challenges for the evolvability assurance of microservices. *Empir. Softw. Eng.* **2021**, *26*, 1–39. [CrossRef]
57. Walker, A.; Cerny, T.; Song, E. Open-Source Tools and Benchmarks for Code-Clone Detection: Past, Present, and Future Trends. *SIGAPP Appl. Comput. Rev.* **2020**, *19*, 28–39. [CrossRef]
58. Bakhtin, A.; Al Maruf, A.; Cerny, T.; Taibi, D. Survey on Tools and Techniques Detecting Microservice API Patterns. In Proceedings of the 2022 IEEE International Conference on Services Computing (SCC), Barcelona, Spain, 11–15 July 2022; pp. 31–38. [CrossRef]
59. Parker, G.; Kim, S.; Maruf, A.A.; Cerny, T.; Frajtak, K.; Tisnovsky, P.; Taibi, D. Visualizing Anti-Patterns in Microservices at Runtime: A Systematic Mapping Study. *IEEE Access* **2023**, *11*, 4434–4442. [CrossRef]
60. Sampaio, A.R.; Kadiyala, H.; Hu, B.; Steinbacher, J.; Erwin, T.; Rosa, N.; Beschastnikh, I.; Rubin, J. Supporting Microservice Evolution. In Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China, 17–22 September 2017; pp. 539–543. [CrossRef]
61. Burgess, K.; Hart, D.; Elsayed, A.; Cerny, T.; Bures, M.; Tisnovsky, P. Visualizing Architectural Evolution via Provenance Tracking: A Systematic Review. In Proceedings of the Conference on Research in Adaptive and Convergent Systems, Virtual, 3–6 October 2022; Association for Computing Machinery: New York, NY, USA, 2022; RACS '22, pp. 83–91. [CrossRef]
62. Granchelli, G.; Cardarelli, M.; Di Francesco, P.; Malavolta, I.; Iovino, L.; Di Salle, A. MicroART: A Software Architecture Recovery Tool for Maintaining Microservice-Based Systems. In Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden, 5–7 April 2017; pp. 298–302. [CrossRef]
63. Arbuckle, T. Measuring Multi-Language Software Evolution: A Case Study. In Proceedings of the 12th International Workshop on Principles of Software Evolution and the seventh Annual ERCIM Workshop on Software Evolution, Szeged, Hungary, 5–6 September 2011; Association for Computing Machinery: New York, NY, USA, 2011; IWPSE-EVOL '11, pp. 91–95. [CrossRef]

64. Apolinário, D.R.; de França, B.B. A method for monitoring the coupling evolution of microservice-based architectures. *J. Braz. Comput. Soc.* **2021**, *27*, 1–35. [CrossRef]
65. Wohlin, C. *Experimentation in Software Engineering: An Introduction*; International Series in Engineering and Computer Science; Kluwer Academic: Philip Drive Norwell, MA, USA, 2000.