applied sciences

Article

Assessing Evolution of Microservices Using Static Analysis

Amr S. Abdelfattah [1,*], Tomas Cerny [1,*], Jorge Yero Salazar [2], Xiaozhou Li [3], Davide Taibi [3] and Eunjee Song [2]

[1] Systems and Industrial Engineering, University of Arizona, Tucson, AZ 85721, USA
[2] Computer Science, Baylor University, Waco, TX 76798, USA; jorge_yero1@baylor.edu (J.Y.S.); eunjee_song@baylor.edu (E.S.)
[3] Computer Science, University of Oulu, 90570 Oulu, Finland; xiaozhou.li@oulu.fi (X.L.); davide.taibi@oulu.fi (D.T.)
* Correspondence: amrelsayed@arizona.edu (A.S.A.); tcerny@arizona.edu (T.C.)

Abstract: Microservices have gained widespread adoption in enterprise software systems because they encapsulate the expertise of specific organizational subunits. This approach offers valuable insights into internal processes and communication channels. The advantage of microservices lies in their self-contained nature, streamlining management and deployment. However, this decentralized approach scatters knowledge across microservices, making it challenging to grasp the holistic system. As these systems continually evolve, substantial changes may affect not only individual microservices but the entire system. This dynamic environment increases the complexity of system maintenance, emphasizing the need for centralized assessment methods to analyze these changes. This paper derives and introduces quantification metrics to serve as indicators for investigating system architecture evolution across different system versions. It focuses on two holistic viewpoints of inter-service interaction and data perspectives derived through static analysis of the system's source code. The approach is demonstrated with a case study using established microservice system benchmarks.

Keywords: software evolution; static analysis; metrics; software architecture

Citation: Abdelfattah, A.S.; Cerny, T.; Yero Salazar, J.; Li, X.; Taibi, D.; Song, E. Assessing Evolution of Microservices Using Static Analysis. *Appl. Sci.* **2024**, *14*, 10725. https://doi.org/10.3390/app142210725

Academic Editor: Young-Gab Kim

Received: 11 September 2024
Revised: 24 October 2024
Accepted: 13 November 2024
Published: 20 November 2024

1. Introduction

Microservice architecture is commonly employed in complex systems, facilitating scalability and decomposing complicated organizational structures into smaller, self-contained, independently managed units. These units are then governed by separate development teams [1]. It becomes critical to understand the architecture of such systems to effectively evolve them. System evolution is inevitable, driven by various factors such as new market demands, technological shifts, and optimization efforts. This evolutionary process for the system source code often involves new features, resolving bugs, and potentially introducing new services with their respective data models and system dependencies.

However, due to the decentralized nature of this evolution, which spreads across semi-autonomous teams, the holistic system-centric architectural perspective is often lost [2]. Moreover, individual teams may inadvertently make suboptimal design choices, leading to inefficiencies that affect the system. Consequently, system architecture may degrade during changes as system complexity increases.

To address these challenges, specifically the absence of a system-centered view, signified by a lack of assessment and reasoning mechanisms in system evolution available before the system deployment, we propose deriving a centralized perspective of the system's architecture while tracing quantifiable structural metrics across microservice system versions throughout the evolution to illustrate the impact of changes to individual microservices.

Software architecture can be described by various viewpoints [3]. A viewpoint represents a set of conventions for creating, interpreting, and utilizing an architectural view to address specific concerns. For microservices, the literature has proposed service and data viewpoints to describe the system holistic perspective [4–6]. Service viewpoint constructs service models specifying microservices, endpoints, and interactions, which should be an essential view in the system's holistic view [7]. It aids in assessing potential ripple effects during their evolution [2]. On the other hand, the data viewpoint addresses data models and microservice-bounded contexts. While these viewpoints are used to describe the overall system architecture [3], they have not been used to assess system evolution.

Objective. The main objective of this paper is to utilize statically derived approximations of service and data viewpoints and translate them into quantifiable metrics to track the evolution of a microservice system and the impact of individual changes across microservices. These metrics serve as indicators to comprehend the system's evolution in response to changes. Regularly calculating these metrics enables architects to make informed decisions to optimize the architecture, ensuring improved performance, scalability, and maintainability. This involves a methodology using Software Architecture Reconstruction (SAR)) [8,9], which applies static analysis of microservices source code to create viewpoints for calculating various centric metrics.

This work showcases a comprehensive case study with a practical application of the proposed metrics in the context of system evolution analysis. The case study employs a variety of approaches to demonstrate and evaluate the underlying reasons driving the evolution of systems across various releases.

Research Questions. Identifying and calculating centric metrics provides indicators of changes. Even if they do not capture every detail perfectly, architects can better understand the system's evolution and make informed decisions to optimize the architecture. For instance, if the number of microservices increases over time, it suggests the system is evolving to meet new requirements. Splitting a service into multiple specialized microservices can enhance scalability and performance, but adding extra services might impact system modularity and introduce more connections. Similarly, if the number of data entities grows due to new features, it can affect data storage and retrieval efficiency. Tracking these numbers and their changes helps ensure that management practices evolve to maintain performance and scalability.

These indicators provide practitioners and architects with a clearer and more focused perspective when exploring human-centric approaches such as architecture visualization or source code debugging and investigation. The paper addresses the following two research questions:

RQ$_1$ How do we quantify the service viewpoint and assess its evolution?

To address this research question, we utilized two metrics and their changes: the number of microservices and the number of microservice connections. The number of microservices provides a quantitative measure of the system's scale or complexity, while the number of microservice connections measures the communication intensity between services.

When calculated from the centric service viewpoint, these metrics offer insights into the system's modularity evolution by highlighting expansions or contractions in high-level components between subsequent versions and releases. Additionally, they reveal changes in the dependency view across system releases.

RQ$_2$ How to quantify the data viewpoint and assess its evolution?

To address this research question, we utilized five different metrics from a data perspective. These metrics include changes in the number of entities, both persistent and transient, and the number of relationships between these entities. These metrics provide insights into the scale of data entities within the system and measure the complexity and interconnectedness of data entities within a data model. Additionally, they measure the scaling of bounded contexts by the number of matching entities and relationships between different bounded contexts of microservices.

These numbers offer insights into the data perspective and its modularity changes throughout system releases. They also reveal changes in bounded contexts over the system's evolution.

Academic Contributions. This paper makes the following academic contributions to the field of microservice-based systems:

- Theoretical Contribution: We introduced and defined seven novel system-centric metrics tailored specifically for microservice architectures. Unlike traditional metrics, which often focus on monolithic systems or lack granularity for microservices, these metrics provide a new way to quantify key architectural properties such as service granularity. This advances the current understanding of microservice system behavior, helping to bridge a gap in the existing literature.
- Methodological Contribution: We developed and implemented automated methods for extracting the proposed metrics directly from the system's source code. This methodological advancement allows for the efficient, repeatable, and scalable analysis of microservice-based architectures, enabling researchers and practitioners to assess systems without extensive manual intervention. The use of automation also opens up new pathways for continuous monitoring and checking of system health.
- Practical Contribution: A Java-based prototype was developed as a practical tool for calculating and reporting these metrics. This prototype can be integrated into development pipelines, providing indicators about the system architecture that can influence architectural decisions and system improvements.
- Data Contribution: We published a dataset containing the extracted metrics data, capturing insights from both service and domain viewpoints of a benchmark system. This dataset provides a valuable resource for further research, enabling other scholars to replicate and extend our study or apply the metrics to different microservice systems.
- Empirical Contribution: A comprehensive case study was conducted to evaluate and interpret the extracted metrics. This empirical investigation not only validates the usefulness of the metrics but also provides evidence-based insights into how these metrics can be applied in realistic scenarios, contributing to the body of knowledge on microservice architecture assessment.
- Visualization Contribution: We adapted a visualization approach to display architectural views from both service and data viewpoints. This visualization is essential for understanding and reasoning about the relationship between different microservices in the system, offering a new way to analyze and optimize microservice architectures.

The rest of this paper is organized as follows: Section 2 explains the related works. In Section 3, the paper details the methodology used for metric extraction and viewpoints representation. Section 4 focuses on evaluating the methodology and presenting a case study to demonstrate the results. Section 5 delves into the answers to the research questions and discusses the implications and potential extensions of the methodology. Section 6 highlights potential threats to the validity of the research. Section 7 concludes the study.

2. Related Work

The evolution of microservices brings up multifaceted challenges across various system dimensions. To name a few, Bogner et al. [2] classified many challenges that practitioners face with microservices and their evolution, including lack of centralization, ripple effects, and lack of guidance. We must be aware that it is a common setting that microservices are managed by distinct teams [10,11], and when there are hundreds of microservices interconnected, management is really complex. This can be underlined by Lercher et al. [12] describing problems with API change propagation. Such change propagation brings significant overhead on communication across teams when aiming to evolve microservices. Despite the initial assumptions for many that microservices can be managed independently, there are microservices dependencies that could lead to difficult maintenance needing multiple team attention [13]. Unfortunately, there is a lack of change impact analysis tools for microservices [14].

To proactively identify and mitigate potential issues, a comprehensive understanding of how the system evolves with each change becomes imperative [2,15]. Consequently, multiple studies [16–18] introduced diverse approaches and metrics aimed at comprehending the system and its complex perspectives.

Various studies have introduced SAR techniques that enable the extraction of viewpoints from the system [19]. Maffort et al. [20] employed static analysis coupled with reflection modeling to create a high-level model that depicts architectural components and data/control flows. They described component granularity as a higher-level organizational unit that encapsulates related functionalities or groups of classes within the system. They provided a lightweight approach to identify architecture conformance issues but focus on specific absences and divergences. Soldani et al. [21] analyzed Docker configurations to grasp the microservice deployment topology, providing insights into component interactions. Additionally, various studies [22–24] have combined static and dynamic analysis to furnish both static and dynamic viewpoints, facilitating comprehensive architecture reconstruction.

However, while these studies have proposed approaches to reconstruct different system perspectives, they often overlook the critical aspect of system evolution. For instance, Mayer et al. [25] addressed system evolution by considering snapshots of the current architecture, yet they do not consider the dynamic evolution of microservices over time. Moreover, Sampaio et al. [26] proposed an approach that leverages both static and dynamic information to generate a representation of evolving microservice systems based on service evolution modeling.

The evolution of microservice architecture significantly impacts the cohesion and coupling of system components. Several studies have introduced metrics designed to assess these critical features of the system. For instance, de Freitas et al. [27] emphasized metrics at both the individual service level and in aggregate. These metrics delve into the frequency of interactions between microservices, patterns of relationships formed through these interactions, the equilibrium of dependencies across services, and the significance of each service within the architecture. Meanwhile, Moreira et al. [28] proposed three metrics that focus on the internal perspective of services, assessing aspects such as the consistency in parameters and return types across exposed interfaces, internal cohesion based on inputs and outputs, and the overall cohesion within a service reflecting unity in its operational implementations.

Moreover, Genfer et al. [29] considered mining software repositories to assess miroservice evolutions. The focus is on the API and presents five high-level architectural metrics. Also, their approach only applied regular expressions on the source code to connect microservices over endpoints, which is limiting and lacks deeper details about what is below the API, which might be the core problem.

On the other hand, visualization is crucial for conveying architectural viewpoints and their evolution to practitioners, allowing them to analyze and understand the properties of architecture. The literature highlights various visualization approaches, such as those in [30–33]. These studies present different approaches and rendering mediums for visualizing the same microservice viewpoint, mostly the service viewpoint, but each approach has its own drawbacks, posing additional challenges for practitioners while investigating the system and its evolution. This emphasizes the need for preliminary indicators to guide practitioners as they navigate these visualizations.

The comparison in Table 1 outlines how our proposed method differs from the existing approaches, emphasizing the distinct focus, extracted metrics, and analysis methods each one utilizes. The methods in the existing literature have limitations when it comes to reconstructing abstract system-centric views directly from the source code. In contrast, this paper presents the construction and utilization of these views, which are subsequently mined to extract comprehensive metrics that provide an insightful indicator and perspective on the entire system. This methodology utilizes static analysis through source code parsers tailored for microservice development frameworks. Rather than relying on regular expressions, our method emphasizes the component-based architecture of microservices, typically structured around key components like controllers, repositories, and services. These concepts are largely language-agnostic, enabling a more general and flexible model [34]. As a result, the metrics in our methodology target the component layers, delving deeper to capture the relationships between them from both the service and data viewpoints. This approach enables a more profound understanding of microservice architecture evolution and enhances the capacity for reasoning about it.

Table 1. Comparison of proposed method with existing approaches.

Study	Focus of Analysis	Extracted Metrics	Analysis Methods
Bogner et al. [2]	Challenges in microservice evolution and system dimensions	No specific metrics introduced	High-level analysis, surveys
Lercher et al. [12]	API change propagation and team communication	No metrics, analysis of ripple effects	Surverys
Maffort et al. [20]	Architecture conformance through detection of absences and divergences from the high-level architectural model	No metrics, four heuristics for absences and divergences	Static and historical source code analysis
de Freitas et al. [27]	Frequency of interactions between microservices and their significance	Service-level interaction metrics	Static analysis, empirical case studies
Soldani et al. [21]	Microservice deployment topology	Deployment topology metrics (container interactions)	Docker configuration analysis
Genfer et al. [29]	Microservice evolution based on API and architectural metrics	Five high-level architectural metrics (API endpoints)	Source code analysis using regular expressions
Proposed Method (This Paper)	System-centric analysis of microservice evolution	Seven system-centric metrics focusing on the component-based architectural service and data viewpoints.	Static analysis, automated extraction

3. The Proposed Centric Metrics

The evolution of the system's architecture is a direct consequence of the modifications made to the source code. Source code modifications predominantly influence both the service and data views. The two viewpoints we consider in this work are the service and data viewpoints, which are focused on relevant concerns of services and the data. The addition or removal of microservices, endpoints, and inter-service calls significantly impacts the service view. Furthermore, an examination of the data model highlights the duplication of data items within different bounded contexts of microservices. Notably, the presence of multiple data item duplications in various microservices signifies a coupling between those services.

3.1. Centric Metrics Methodology Overview

An architecture view is a collection of models representing the system architecture relative to a set of architectural concerns. In particular, the service view constructs service models specifying microservices, endpoints, and interactions. This essential view provides the system's holistic perspective [7] is used to explain the system dependencies and the overall microservice ecosystem and becomes useful when assessing potential ripple effects during microservice evolution [2]. Similarly, the data view uses data models to explain which data are used by which microservice and their interconnection. We can highlight the presence of data entity overlaps, which signifies a coupling between particular services.

While architecture viewpoints serve as a frame of reference for addressing concerns pertinent to the architecture description's purpose [3], this approach utilizes SAR approach [8,9] using static analysis for microservices to generate service and data views corresponding to particular viewpoints. Consequently, analyzing these views yields a timeline of the system's evolution, facilitating an understanding of the motivations behind modifications and the broader impact of these changes.

The methodology is outlined in Figure 1 as follows:

1. **Intermediate Representation Construction (Section 3.2)**: This phase receives the source code of the microservices projects. It extracts the architectural components and constructs an intermediate representation for each of these two service and data central viewpoints, as illustrated in Figure 2.
2. **Metrics Definition (Section 3.3):** It employs the constructed intermediate representations to introduce seven convenient and automated metrics derived and adapted from both the service *(S1, S2)* and data *(D1–D5)* views. These metrics are designed to quantify changes within both aspects of the architecture to serve as indicators of the system's main development stream and offer early insights into potential architectural degradation.
3. **Demonstrating Example (Section 3.4):** It provides an example of how the methodology operates to extract and calculate the proposed metrics.

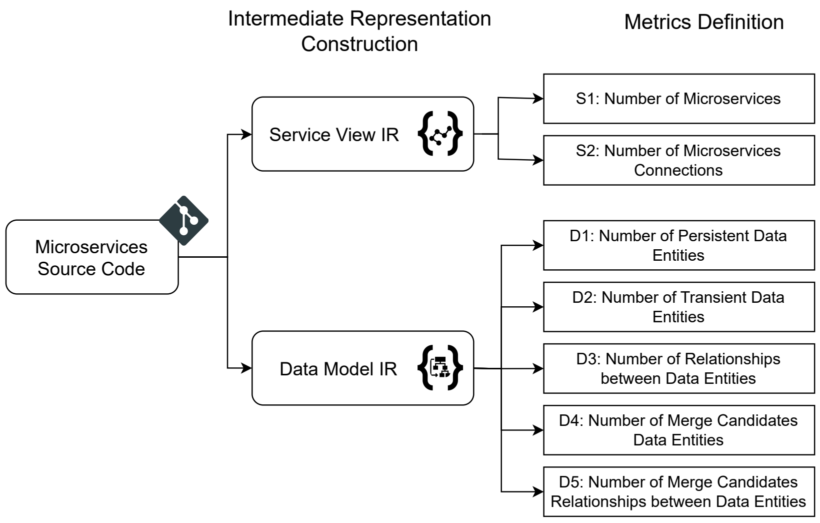

Intermediate Representation Construction
Metrics Definition
Microservices Source Code
Service View IR
Data Model IR
S1: Number of Microservices
S2: Number of Microservices Connections
D1: Number of Persistent Data Entities
D2: Number of Transient Data Entities
D3: Number of Relationships between Data Entities
D4: Number of Merge Candidates Data Entities
D5: Number of Merge Candidates Relationships between Data Entities

Figure 1. Centric Metrics Methodology Overview.

The objective of employing this methodology is to regularly calculate these metrics to provide practitioners and architects with a comprehensive understanding of the system and its evolution, enabling informed decisions to optimize the architecture. This approach aims to help practitioners to balance the integration of new features to the system with ongoing maintenance. Additionally, it offers them indicators to help them while they visually investigate the system and delve into the causes behind the metric fluctuations.

Moreover, these metrics provide a centralized explanation of system properties. They can be tracked across different versions of the system to ensure that system properties align with the changes made in each version. Practitioners can create a timeline of these metrics, illustrating how system attributes evolve across various versions.

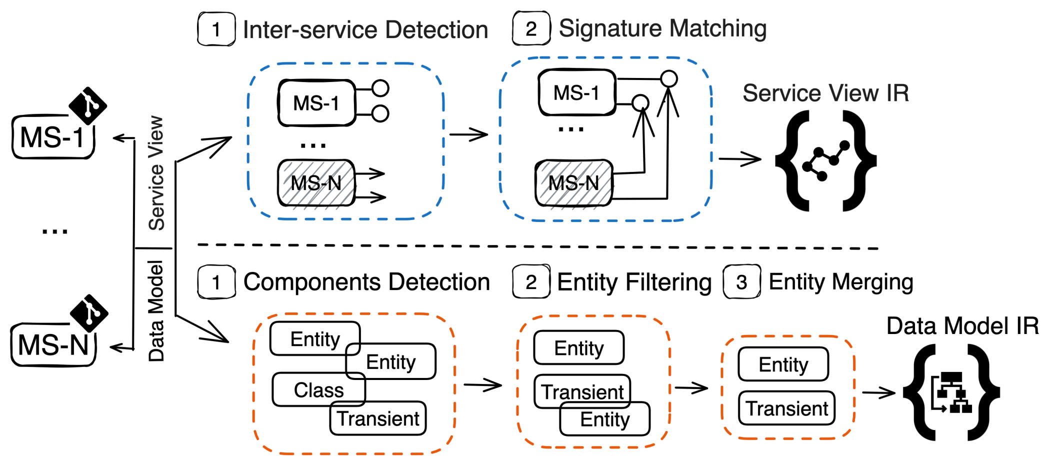


Figure 2. Service view and data model intermediate representation construction process.

3.2. Intermediate Representation Construction

This methodology applies SAR by leveraging static code analysis techniques to construct the service view and data model of the data view from the source code.

The methodology commences by iteratively examining project folders within the system to identify standalone projects as microservices. This can be achieved through parsing specific deployment or configuration scripts found in the project folders, such as Docker Compose deployment files. Once identified, we harness the source code of each microservice to create the corresponding views.

3.2.1. Service View Construction

The construction of the service view follows a two-phase process, as depicted in Figure 2 (upper part). In the first phase, known as *Inter-service Detection*, we employ static analysis to extract endpoint declarations and endpoint requests initiated within the source code. In the second phase, referred to as *Signature Matching*, we employ a regex-like approach to match requests to their corresponding endpoints across different microservices. This matching process considers attributes of endpoints and requests, including endpoint paths, parameters, and HTTP method types, ultimately identifying connections between the source and destination microservices. The extracted information is represented as an intermediate representation (IR) of the service view.

3.2.2. Data Model View Construction

The construction of the data model view comprises three phases, as illustrated in Figure 2 (lower part). The first phase, termed *Components Detection*, focuses on identifying components, which are individual units or classes, along with their attributes within the source code. In the second phase, known as *Entity Filtering*, we select the data entities among the extracted class components. Then, we distinguish data entities, both persistent and transient. Transient entities are identified based on attributes such as setters and getters, which are used in the responses of endpoint methods. Additionally, entity relationships are identified from attribute fields that reference other entities within the system. The third phase, termed *Entity Merging*, identifies entities that are candidates for merging from different microservices based on similar names and matching data types between their fields. The resulting information is represented as an intermediate representation (IR) of the data model view.

3.2.3. Intermediate Representation Formalization

The two IRs constructed from both views encompass vital attributes for depicting the system's inter-connections and data dependencies. These attributes can be expressed using the following mathematical notations, which provide a precise and formal representation of the relationships within the system, along with the formulas for metrics calculation accordingly:

$$S = \{ms_1, ms_2, \cdots, ms_n\};\ ms_i = \langle E_i, C_i, D_{p_i}, D_{t_i} \rangle$$
$$E_i = \{\langle u_j, r_j, P_j \rangle, \cdots\};\ C_i = \{\langle ms_j, u_k, r_k, P_k \rangle, \cdots\};$$
$$D_{p_i} = \{\langle F_j, R_{ij} \rangle, \cdots\};\ D_{t_i} = \{\langle F_j, R_{ij} \rangle, \cdots\}$$
$$P = \{\langle t, n \rangle, \cdots\};\ F_i = \{\langle t, n \rangle, \cdots\}$$

S	- Microservice system
ms_i	- Single microservice.
E_i	- Endpoints defined in the microservice ms_i
C_i	- Calls made from the microservice ms_i to all other microservices.
D_{p_i}	- Persistent data entities in the microservice ms_i.
D_{t_i}	- Transient data entities in the microservice ms_i.
R_j	- Relationships starting from a data entity j to other entities.
F_j	- Fields in a data entity j.
P	- Parameters in an endpoint or in a call.
u	- URL path.
r	- Return type.
t	- Field data type.
n	- Field name.

The constructed views and their properties can be elucidated by categorizing the five main sets of extracted attributes. The process begins with a microservice system as input and subsequently analyzes the system to yield the following sets of attributes per each microservice. Firstly, a set of endpoints (E) is introduced within each microservice. Secondly, a set of request calls (C) is initiated from each microservice towards endpoints in other microservices. Lastly, the system yields two sets of data entities: one for the persistent data entities (D_p) and the other for the transient data entities (D_t).

3.3. Metrics Definition

The seven metrics utilize centralized attributes extracted from the system. Metrics S1 and S2 pertain to the service view, as detailed in Table 2. Metrics D1 through D5 are associated with the data view, as shown in Table 3. Each metric quantifies certain attributes to describe a centric perspective of the microservice system and to assist in the reasoning about its evolution.

Table 2. Service view metric definitions.

S1: Number of Microservices (#μ_s)
Description: The count of microservices in the system. **Value:** $\#\mu_s = \lvert S \rvert$. **Goal:** Provides a quantitative measure of the system's scale or complexity in terms of microservices. **Motivation:** Provides insights into the system's modularity evolution by showing expansions or contractions in high-level components between subsequent releases.
S2: Number of Microservice Connections (#$C\mu_s$)
Description: Number of endpoint calls between microservices in the system. **Value:** $\sum_{i=1}^{n} \lvert C_i \rvert$; where $n = \lvert S \rvert$. **Goal:** Provides a measure of the communication intensity between microservices in the system. **Motivation:** Reveals changes in the dependency view across system releases and identifies potential bottlenecks, such as excessive connections to a single service.

Table 3. Data view metric definitions.

D1: Number of Persistent Data Entities (#PDE_s)
Description: The count of relational and non-relational persistent data entities in the system. **Value:** $\sum_{i=1}^{n} \lvert D_{p_i} \rvert$; where $n = \lvert S \rvert$. **Goal:** Provides insight into the scale of persistent data entities within the system. **Motivation:** Provides insights into a data perspective, i.e., the storage approach per each microservice. In addition to the data model, modularity changes throughout releases.
D2: Number of Transient Data Entities (#TDE_s)
Description: Number of data entities used as data transfer objects (DTOs) across microservices (not persistent in storage). **Value:** $\sum_{i=1}^{n} \lvert D_{t_i} \rvert$; where $n = \lvert S \rvert$. **Goal:** Provides an indication of the usage scale of data transfer objects. **Motivation:** Illustrates the evolution of additional entities created for customizing responses and data transfer purposes but not involved in domain/business logic. It indicates whether each microservice consumes or delivers more data models to others.
D3: Number of Relationships between Data Entities (#RDE_s)
Description: The count of relationships between data entities in each microservice's data model. **Value:** $\lvert MR \rvert/2$; such that $MR = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} \{r, f(r) \mid r \in \bigcup_l R_{ijl}\}$; where $n = \lvert S \rvert$, $m = \lvert D_{p_i} \rvert + \lvert D_{t_i} \rvert$, $f(r)$ is a function that reverses relationships to prevent duplicate counting of both the relationship and its reverse counterpart. **Goal:** Provides a measure of the complexity and interconnectedness of data entities within each microservice's data model. **Motivation:** Demonstrates the evolution of data model complexity and the presence of cohesive models within each microservice over different releases.
D4: Number of Merge Candidate Data Entities (#MDE_s)
Description: The count of entities duplicated in multiple bounded contexts. These entities may retain different fields based on the purpose of each bounded context. **Value:** $(\sum_{i=1}^{n} \lvert Dp_i \rvert + \lvert Dt_i \rvert) - \lvert DE \rvert$; where $DE = \{f_d(d) \mid d \in \bigcup_{i=1}^{n} Dp_i + Dt_i\}$, $n = \lvert S \rvert$, $f_d(d)$ is a function that receives an entity and returns the corresponding entity after the Entity Merging phase; it is defined as $$f_d(d) = \begin{cases} d & \text{if } d \text{ not merged} \\ d' & \text{if } d \text{ was merged into } d' \end{cases}$$ **Goal:** Provides insight into the level of duplication across bounded contexts, highlighting potential opportunities for consolidation or optimization. **Motivation:** Shows bounded context changes over system evolution; it reveals data dependencies among microservices.
D5: Number of Merge Candidate Relationships between Data Entities (#$MRDE_s$)
Description: The count of relationships that are candidates for merging based on the merge candidates between entities (#MDE_s). **Value:** $(\sum_{i=1}^{n} \sum_{j=1}^{m} \lvert R_{ij} \rvert) - \lvert RDE \rvert$; where $RDE = \{f_r(r) \mid r \in \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} R_{ij}\}$, $n = \lvert S \rvert$, $m = \lvert D_{p_i} \rvert + \lvert D_{t_i} \rvert$, $f_r(r)$ is a function that receive a relationship and returns the corresponding relationship after the Entity Merging phase, it is defined as $$f_r(r) = \begin{cases} r & \text{if } r \text{ not merged} \\ r' & \text{if } r \text{ was merged into } r' \end{cases}$$ **Goal:** Indicates potential opportunities for merging relationships between entities based on identified merge candidates. **Motivation:** Provides a holistic data model view of the microservice system and showcases changes in data fragments across different system releases.

3.4. Demonstrating Example

This section illustrates the proposed metrics examples. Consider four microservices MS-1 $\cdots$ MS-4, where each microservice contains data entities, either persistent (P) or transient (T). These entities have relationships with each other, and the connections between microservices are depicted with arrows linking two transient entities that facilitate data transfer through these connections. Refer to Figure 3 for a visual representation of this example.

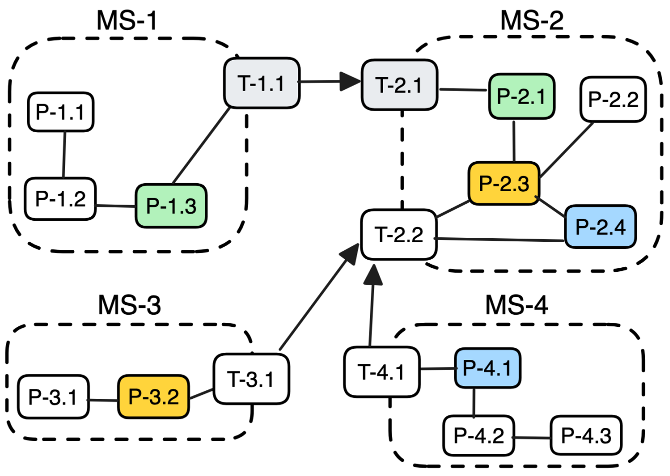


Figure 3. Demonstrating example for metrics. (P: Persistent Entity, T: Transient Entity).

Regarding the metrics related to the service view, we can observe that the number of microservices in the system is four ($\#\mu_s = 4$). Additionally, there are three request calls ($\#C\mu_s = 3$), which occur as follows: MS-1 $\rightarrow$ MS-2, MS-3 $\rightarrow$ MS-2, and MS-4 $\rightarrow$ MS-2.

In the context of data view metrics, there are 12 persistent data entities ($\#PDE_s = 12$). Specifically, MS-1 through MS-4 contain three, four, two, and three persistent data entities, respectively. On the other hand, there are five transient data entities ($\#TDE_s = 5$). MS-2 has two transient data entities, while other microservices have one each. Furthermore, there are 14 relationships between these data entities ($\#RDE_s = 14$): three relationships for each of MS-1 and MS-4, six relationships in MS-2, and two relationships in MS-3.

To illustrate the two metrics related to merge candidates, we demonstrated an entity merging process to identify potential merge candidates between data entities, as depicted in Figure 4. These resulting entities were generated by tracing the connections between transient data entities among microservices and the relationships between these transient and persistent data entities. In these two figures, entities with the same color represent similar entities that exist in multiple microservices' bounded contexts. By merging these similar entities into a single entity, the total number of data entity merge candidates is four ($\#MDE_s = 4$). These candidates are (T-1.1 and T-2.1), (P-1.3 and P-2.1), (P-2.3 and T-3.2), and (P-2.4 and P-4.1). Additionally, there are one relationship merge candidate ($\#MRDE_s = 1$), which is produced by a merge of the relationship between (T-1.1 and P-1.3) and (T-2.1 and P-2.1). These two relationships were merged because the merging process occurred in both of the entities they connected (T-1.1 and T-2.1) and (P-1.3 and P-2.1).

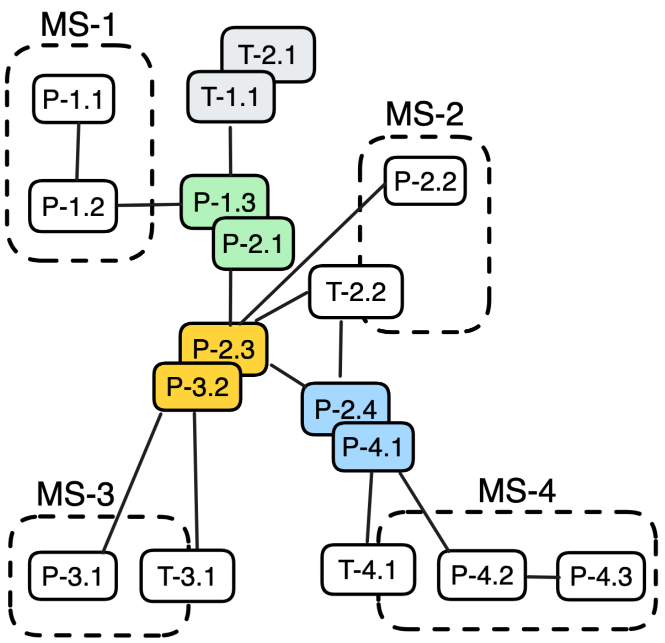


Figure 4. Entity Merging in the demonstrating example. (P: Persistent Entity, T: Transient Entity).

4. Evaluation Case Study

In this section, we showcase the application of our proposed methodology to facilitate reasoning about system evolution. While the proposed metrics provide insights into central system perspectives, they do not inherently reveal the underlying reasons for variations between different system versions. Consequently, these metric values serve as indicators, prompting architects and practitioners to explore the drivers of system changes.

We have implemented the proposed metrics in a prototype and applied them to an open-source testbench to assess and discuss its architectural evolution characteristics. This case study serves the following two objectives:

- Evaluating the feasibility of applying the proposed metrics to quantify a real-life system architecture as part of a proof of concept.
- Emphasizing the significance of the proposed metric measurements in guiding practitioners to investigate specific aspects of the system.

4.1. Experiment Setup

This section details the case study setup, including the software test bench sourced from open-source projects with multiple versions available, as well as the prototype we developed to assess the proposed metrics.

4.1.1. TestBench

We utilized a TrainTicket [35] microservice testbench to demonstrate the case study. It is commonly used in the research community as a representative of real-life microservice systems. Its repository contains seven releases (at the time of this paper written). To demonstrate the evolution impact on the system architecture, this case study selects the following three different releases (versions): v0.0.1 (v0.0.1: https://github.com/FudanSELab/train-ticket/tree/0.0.1, accessed on 20 July 2024), v0.2.0 (0.2.0: https://github.com/FudanSELab/train-ticket/tree/v0.2.0, accessed on 20 July 2024) and v1.0.0 (v1.0.0: https://github.com/FudanSELab/train-ticket/tree/v1.0.0, accessed on 20 July 2024). Those versions show a progressive evolution of the system starting from the first release (v0.0.1) until the current latest release (v1.0.0).

4.1.2. Prototype Implementation

We implemented a proof of concept of the proposed approach in a prototype (Prototype: https://zenodo.org/records/11215210, accessed on 20 July 2024). It is built for analyzing Java-based microservices projects that use the Spring Boot framework [36]. It utilized static source code analysis techniques to extract the data necessary for calculating the metrics and also for constructing the intermediate representations of the service view and data view. The extracted data are published at an online dataset (Dataset: https://zenodo.org/records/10052375, accessed on 20 July 2024).

It accepts as its input a GitHub repository containing microservices-based projects. It scans each microservice to find the Spring Boot REST client (i.e., RestTemplate client) and detect HTTP calls between the services. It matches the detected calls with the endpoints. Moreover, it extracts the bounded context data model of the individual microservices. It scans all local classes in the project using a source code analyzer. To filter this list down to classes serving as persistent and transient data entities, it checks for persistence annotations (i.e., JPA standard entity annotations such as @Entity and @Document), and also, for annotations from Lombok [37] (i.e., @Data), a tool for automatically creating entity objects. To check merge candidates, it uses the WS4J project [38], which uses the WordNet project [39] to detect name and field similarity. This prototype outputs a JSON representation for the service and data views intermediate representations.

4.2. Results

These results quantify the proposed metrics using the TrainTicket testbench. The prototype was applied to the three distinct testbench releases. The seven proposed metrics are listed in Table 4 and plotted in Figure 5.

With these metrics at our disposal, we can interpret the system's evolution across the three versions from both the service and data viewpoints, as outlined below.

Table 4. TrainTicket metric values.

Metric	v0.0.1	v0.2.0	v1.0.0	
S1. $\#\mu_s$	46	40	42	Service View
S2. $\#C\mu_s$	135	91	90	
D1. $\#PDE_s$	31	27	27	Data View
D2. $\#TDE_s$	0	182	81	
D3. $\#RDE_s$	0	41	43	
D4. $\#MDE_s$	11	139	32	
D5. $\#MRDE_s$	0	17	19	

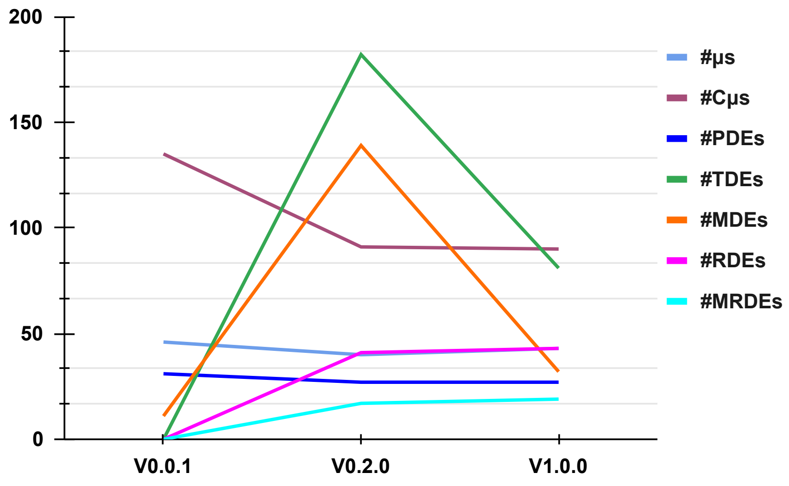


Figure 5. Results of applying the metrics to the evolving TrainTicket system.

4.3. Service View Metric Results

The results of the two service view metrics (S1 and S2) demonstrate the evolution across the three versions in terms of microservice growth and communication intensity of their connections.

S1: Number of Microservices ($\#\mu_s$).

It decreased from 46 in v0.0.1 to 40 in v0.2.0 and then slightly increased to 42 in v1.0.0. This indicates a reduction in the system's high-level components or a consolidation of functionality.

S2: Number of Microservices Connections ($\#C\mu_s$).

It decreased from 135 in v0.0.1 to 91 in v0.2.0 and remained relatively stable at 90 in v1.0.0. This suggests a decrease in the interdependence between microservices or a more optimized communication pattern.

4.4. Data View Metric Results

The results of the five data view metrics (D1–D5) highlight the evolution across the three versions of TrainTicket, with a focus on both persistent and transient data entities and their evolving relationships. Additionally, these metrics emphasize the overlap and potential merging of entities across microservices as the system evolves from one version to the next.

D1: Number of Persistent Data Entities ($\#PDE_s$).

It remained relatively constant across all versions, starting at 31 in v0.0.1 and remaining consistent at 27 in both other versions. This indicates stability in the storage approach and data model for the microservices.

D2: Number of Transient Data Entities ($\#TDE_s$).

It increased significantly from 0 in v0.0.1 to 182 in v0.2.0 and dramatically decreased to 81 in v1.0.0. This suggests the introduction of additional entities for customizing responses and data transfer purposes, which decreased in the latest version to optimize the usage of entities across microservices.

D3: Number of Relationships between Data Entities ($\#RDE_s$).

It started at 0 in v0.0.1, grew to 41 in v0.2.0, and 43 in v1.0.0, suggesting the evolution of the data model complexity and the establishment of relationships between entities.

D4: Number of Merge Candidate Data Entities ($\#MDE_s$).

It increased from 11 in v0.0.1 to 139 in v0.2.0 and decreased to 32 in v1.0.0. This indicates a growing complexity in the bounded contexts and the need for merging similar entities.

D5: Number of Merge Candidate Relationships between Data Entities ($\#MRDE_s$).

It started at 0 in v0.0.1, increased to 17 in v0.2.0, and further increased to 19 in v1.0.0. This indicates the emergence of candidate relationships for merging based on similar entities.

Analysis of these metrics offers valuable insights into the system's evolution, shedding light on shifts in microservice counts, connections, diverse data entities, and their relationships across the entire system. A more profound understanding can be achieved when considering the relationships between these metrics and exploring the cause behind these metrics indicators.

4.5. Case Study Discussion

System architects and practitioners need to consider not only the raw metrics but also the underlying architectural changes and design decisions that influence these metrics. The interconnected nature of microservices means that changes in one area can have ripple effects throughout the system. Further analysis and exploration of the relationships between these metrics can provide a more comprehensive understanding of the system's evolution and its implications for quality and performance.

This section builds upon the findings of the case study to explore three reasoning approaches: question-based reasoning, resolution-based reasoning, and human-centric reasoning.

4.5.1. Question-Based Reasoning

Analyzing metric values raises questions about the system's properties during its evolution. While team members familiar with the system might know the answers, practitioners often need to examine the individual source code of each microservice for investigation. Visualization helps practitioners in exploring and reasoning about these metrics at various levels of detail. Therefore, an intermediate representation is provided in a universal JSON format, ready to be integrated into visualization tools such as those described in [19,32].

The following case study reasoning questions (CQ) have emerged to demonstrate how to follow the indicators provided by metrics to investigate the reasons through the system's source code.

CQ1. What factors contributed to the high number of connections in the v0.0.1 service view compared to the other two versions?
CQ2. What explains the disparity between the number of relationships and entities?
CQ3. Why are there no relationships and transient entities in v0.0.1?
CQ4. What is the reason for the substantial number of merged entity candidates ($\#MDE_s$), particularly in v0.2.0?

In response to CQ1, the reduced number of connections in v0.0.1 can be attributed to the presence of multiple composite microservices that require communication with other services to fulfill their functionalities. This can be exemplified by considering two services, *ts-preserve-service* and *ts-station-service*. The former, *ts-preserve-service*, lacks persistence storage but relies on 12 other microservices to accomplish its functions. Conversely, the latter, *ts-station-service*, has no dependent services but boasts 21 dependencies.

Furthermore, addressing CQ2, the logical relationships between entities rather than direct connections can explain the metrics indication. For instance, the Listing 1 demonstrates that the *FoodStore* entity within *ts-food-service* utilizes the *stationId* field to store the ID value of a *Station* entity, even though no established relationship formally links the two entities. Additionally, the structure of certain microservices, such as *ts-station-service*, featuring persistent storage, while others, like *ts-preserve-service*, lack this feature, can add more justification to this question.

Listing 1. Logical relationship *(ts-food-service: FoodStore entity).*

```
@Data
public class FoodStore {
    private UUID id;
    //logical relationship
    private String stationId;
}
```

Answering CQ3 involves two aspects. First, the absence of transient data entities in v0.0.1, unlike the other two versions, is attributed to the fact that v0.0.1 does not employ Lombok annotations to annotate their transient data classes. Consequently, the prototype could not guarantee other hypotheses for identifying transient entity types. Furthermore, upon investigation, it was revealed that v0.0.1 uses some of the persistent data entities for the purpose of transferring data between microservices. While this may appear as a limitation in the prototype's implementation, after investigation, it highlights an evolutionary perspective on the diverse techniques and strategies employed by v0.0.1 for handling transient data entities, distinct from the approaches taken in the other two versions. The second aspect of this question pertains to the different types of persistent data entities. In v0.0.1, in addition to the absence of transient data entities, no relationships were extracted. This is because v0.0.1 utilizes a non-relational database, MongoDB, for persistence storage. As summarized in Table 5, the system transitioned from non-relational to relational databases in the latest version.

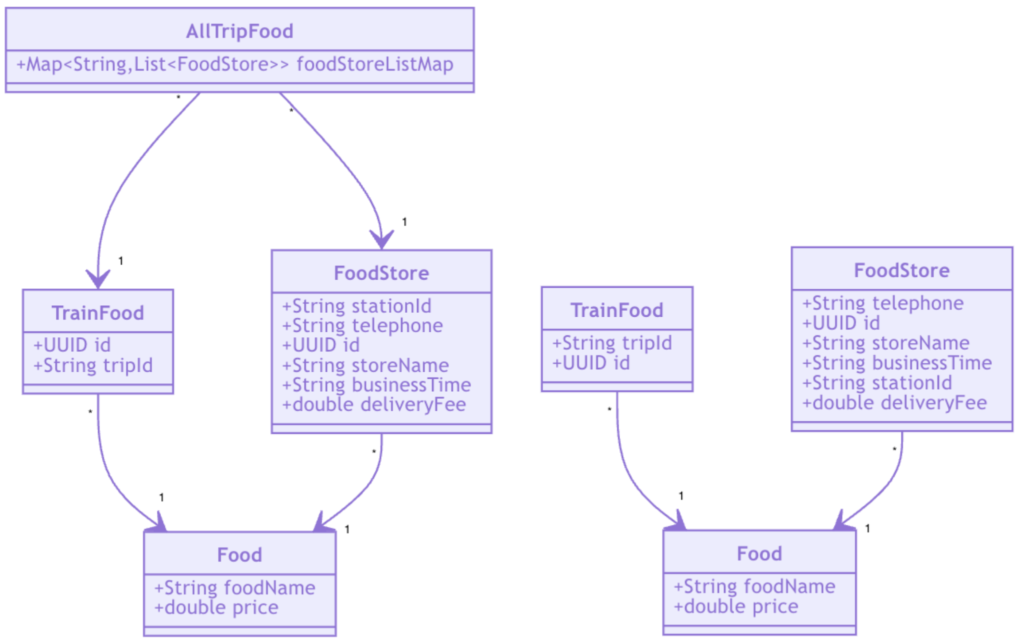
Addressing CQ4, this study provides insights into the process of merging entities across multiple bounded contexts. As highlighted in Table 6, v0.2.0 contains a total of 209 entities ($\#PDE_s + \#TDE_s$) within its bounded context. However, this number of entities is significantly reduced to 70 entities within the context map after merging 139 candidate entities. Upon closer examination of the system's source code, it becomes apparent that 39 entities are duplicated across multiple microservices in v0.2.0. This duplication is particularly evident in v1.0.0, where 39 entities are consolidated into a single shared microservice named *ts-common* to address this concern. This is demonstrated in Figure 6, illustrating three related entities (*Food, TrainFood, FoodStore*) that are duplicated within the bounded contexts of *ts-food-service* and *ts-food-map-service*.

Table 5. Data entity types.

Data Entity	v0.0.1	v0.2.0	v1.0.0	
NoSQL	29	27	0	Sum
SQL	2	0	27	
#PDE $_s$ (Total)	31	27	27	

Table 6. Merging Data Entities.

Data Entity	v0.0.1	v0.2.0	v1.0.0	
$\#PDE_s$	31	27	27	Merge
$\#TDE_s$	0	182	81	
$\#MDE_s$	11	139	32	
Context Map (Merged)	20	70	76	



(**a**) Entities in ts-food-service (**b**) Entities in ts-food-map-service

Figure 6. An example of duplicating entities in microservices (visualized using [40]). Entities are represented in boxes with titles and attributes, connected by directed arrows indicating multiplicity (1 for one and * for many).

4.5.2. Resolution-Based Reasoning

In addition to the case study questions, it is essential to delve into the issues present in v0.0.1 and the resolutions in v1.0.0. The initial system design exhibited some problematic tendencies; there are many small nano-services, which is a recognized anti-pattern for microservices. It can also be described as a microservice greedy anti-pattern, where new microservices were created for each feature, even when they were unnecessary, as evident in later versions when they were merged and consolidated. Furthermore, in v0.0.1, only half of these microservices defined data entities, while the other half appeared to serve as transfer services in the business layer. This approach can be characterized as a wrong cuts anti-pattern, dividing the system into layers based on functionality rather than domain and business considerations, as confirmed by Walker et al. [41] when analyzed the same testbench.

An important observation lies within Table 6. While most dynamic analysis tools consider service calls as the foundation for service connections, the merge entity candidates can indicate another dimension of connectivity when many entities within the system represent the same concept. This presents an efficient mechanism for connecting microservices beyond service calls. v0.2.0 contains 209 scattered entities across the system that reduced to 70 within the context map. Such a process is non-trivial for human-based analysis and would require manual merging with each system change.

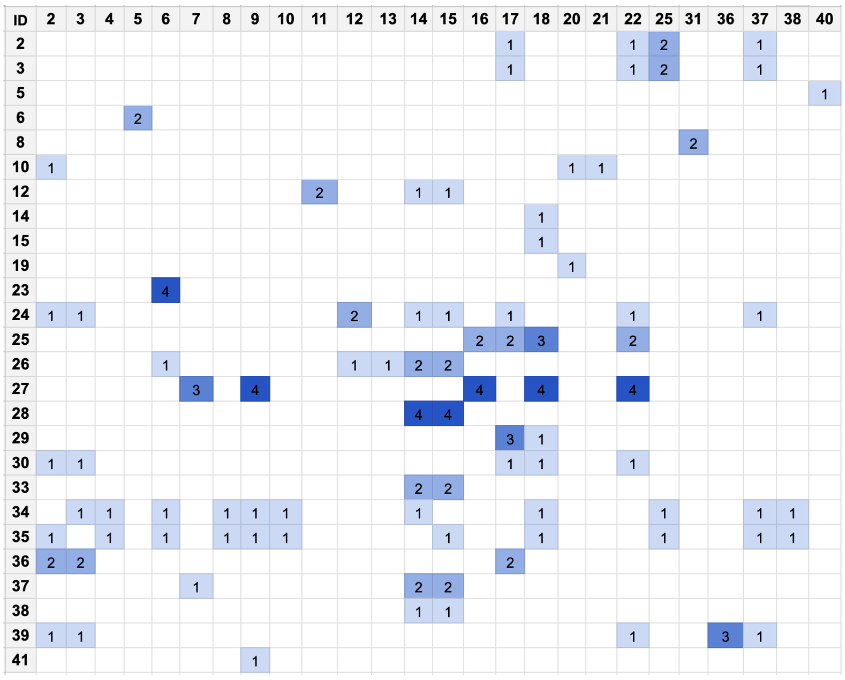
4.5.3. Human-Centric Reasoning

The flexibility of the presented intermediate representation allows it to adapt and be visualized through various visualization tools. These visualizations can significantly reduce the effort required to investigate and understand the source code. The intermediate representation provides holistic viewpoints (service and data) through tailored approaches. We utilized the heat matrix visualization presented in [32] to depict the number of dependency connections between pairs of microservices in TrainTicket v1.0.0, as shown in Figure 7. Moreover, for the same version, we utilized the matrix visualization to display data viewpoints through the relationships between data entities and identify merge candidates among microservices, as depicted in Figure 8. The shown matrices display the number of corresponding dependencies within the cells, using color depth to visually represent them—darker colors indicate more dependencies between each pair. The IDs in the visualization matrices correspond to the microservices listed in Table 7.

Table 7. List of TrainTicket v1.0.0 microservices and their corresponding IDs used for visualization.

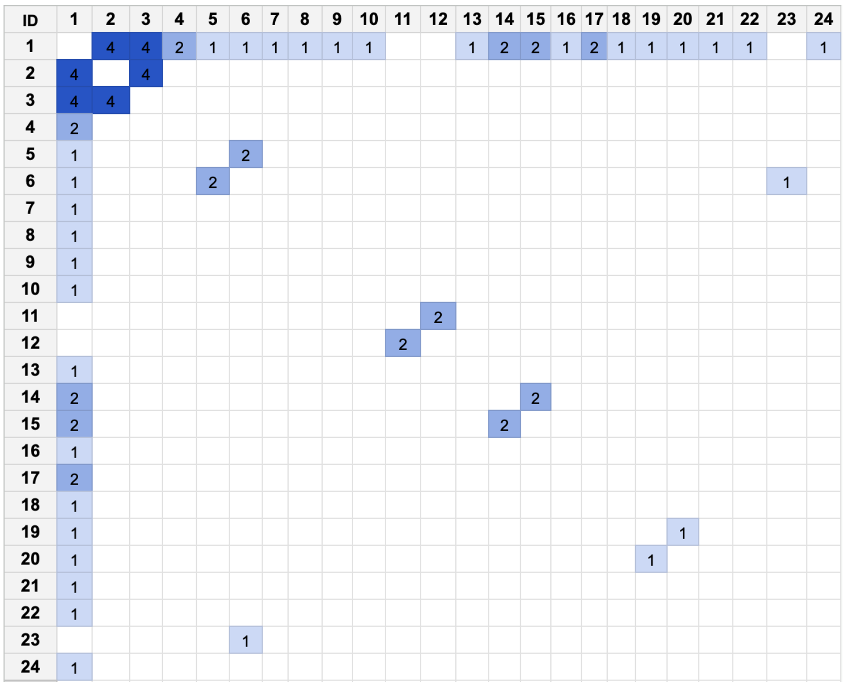
ID	Name	ID	Name	ID	Name
1	ts-common	15	ts-order-service	29	ts-admin-route-service
2	ts-travel-service	16	ts-price-service	30	ts-admin-travel-service
3	ts-travel2-service	17	ts-route-service	31	ts-consign-price-service
4	ts-assurance-service	18	ts-station-service	32	ts-delivery-service
5	ts-auth-service	19	ts-food-delivery-service	33	ts-execute-service
6	ts-user-service	20	ts-station-food-service	34	ts-preserve-other-service
7	ts-config-service	21	ts-train-food-service	35	ts-preserve-service
8	ts-consign-service	22	ts-train-service	36	ts-route-plan-service
9	ts-contacts-service	23	ts-admin-user-service	37	ts-seat-service
10	ts-food-service	24	ts-rebook-service	38	ts-security-service
11	ts-payment-service	25	ts-basic-service	39	ts-travel-plan-service
12	ts-inside-payment-service	26	ts-cancel-service	40	ts-verification-code-service
13	ts-notification-service	27	ts-admin-basic-info-service	41	ts-wait-order-service
14	ts-order-other-service	28	ts-admin-order-service	42	ts-gateway-service

Although visual representation is not the primary focus of this study, our method generates and formalizes a holistic representation to support such tools. We applied the findings from this case study to these visualizations to showcase the results. These visual representations are crucial for analyzing system evolution, offering visual-centric perspectives that aid in understanding and reasoning about architectural changes as the system evolves.



ID	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	20	21	22	25	31	36	37	38	40
2																1				1	2			1		
3																1				1	2			1		
5																										1
6				2																						
8																						2				
10	1																	1	1							
12										2			1	1												
14																	1									
15																	1									
19																		1								
23					4																					
24	1	1									2		1	1		1				1				1		
25															2	2	3			2						
26					1						1	1	2	2												
27						3		4							4		4			4						
28													4	4												
29																3	1									
30	1	1														1	1			1						
33													2	2												
34		1	1		1		1	1	1				1				1				1			1	1	
35	1		1		1		1	1	1					1			1				1			1	1	
36	2	2														2										
37						1							2	2												
38													1	1												
39	1	1																		1			3	1		
41								1																		

Figure 7. Matrix visualization for service view of TrainTicket v1.0.0.



ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1		4	4	2	1	1	1	1	1	1			1	2	2	1	2	1	1	1	1	1		1
2	4		4																					
3	4	4																						
4	2																							
5	1					2																		
6	1				2																		1	
7	1																							
8	1																							
9	1																							
10	1																							
11												2												
12											2													
13	1																							
14	2														2									
15	2													2										
16	1																							
17	2																							
18	1																							
19	1																			1				
20	1																		1					
21	1																							
22	1																							
23						1																		
24	1																							

Figure 8. Matrix visualization for data view of TrainTicket v1.0.0.

5. Discussion and Answers to RQs

This study adopts a centric perspective to examine the architectural evolution of microservices using their service and data viewpoints. In pursuit of this objective, a set of seven metrics is introduced and employed to quantify diverse properties within these two viewpoints. Two metrics focus on aspects of the service view, such as microservice count and inter-service connections (addressing RQ_1), while five metrics pertain to the data view, including data entity types and relationships, including duplicates across different bounded contexts (addressing RQ_2). As a result, these metrics offer valuable insights into elucidating the system's evolution.

Furthermore, the study leverages intermediate representations derived from the system's source code for these two views. These representations can be analyzed to extract metrics and visually presented using customized techniques for deeper exploration and reasoning. The primary aim is to facilitate practitioners' exploration and understanding of the system's central properties.

Metrics as Indicators. The significance of these metrics lies in their ability to serve as indicators of system evolution, offering insights across various dimensions. While the metrics provide indicators about the relative changes when compared across different versions, they also play a vital role in exploring relationships between different metrics within the same version. This enables the reasoning about specific system features and the detection of patterns within the system, as demonstrated in the case study.

Metric Granularity. The proposed metrics present abstract indicators of the holistic perspective of service and data viewpoints, and their granularity can be tailored to different levels within the system. This granularity can be examined at the individual microservice level, allowing the metrics to indicate correlations between microservices within the same version and a single microservice across multiple versions. Furthermore, the granularity can be extended from a technological standpoint, considering the polyglot architecture of microservice systems utilizing various programming languages and technologies. These metrics can offer insight into the evolution of service and data viewpoints in an additional dimension of heterogeneity.

At the same time, granularity can be fine-tuned for each individual metric. For example, the metric of the number of persistent data entities ($\#PDE_s$) can be further divided into relational and non-relational data entities, as demonstrated in the case study. However, this study has chosen the current level of granularity to provide a direct indicator of persistence, with the option to explore different types if the metric values indicate the need for further investigation. A similar granularity approach can be applied to the metric of the number of microservice connections ($\#C\mu$), differentiating between synchronous and asynchronous calls, depending on the specific analysis requirements.

Intermediate Representation Extension. The proposed intermediate representations of the system views contain essential information used for metric extraction and viewpoint visualization. While the proposed methodology primarily employs static source code analysis, dynamic analysis is another valuable technique. Dynamic analysis involves runtime data, which can construct the service viewpoint from the execution perspective. Previous studies have utilized dynamic analysis to extract the service view based on remote procedure calls gathered from logs and traces and to detect anomalies in microservice-based systems [42,43]. Therefore, extending the intermediate representation to include dynamic data analysis alongside the static representation, particularly for the service view, can offer additional metrics to illustrate the system's runtime behavior evolution.

Furthermore, the described methodology can be expanded to include event-based connections. The current representation predominantly focuses on endpoint connections, whereas event-based connections may require additional considerations and interpretations, given the potential one-to-many relationships between a single producer and multiple consumers in an indirect communication manner. Additionally, the intermediate representation can be augmented to encompass system viewpoints, including the technology view and operational view.

Intermediate Representation as a Supportive Infrastructure. The proposed methodology presents metrics as indicators of system evolution and also the intermediate representation of the centric service and data views. The interpretation and visualization of the intermediate representation serve as a valuable tool for practitioners, enabling them to visually analyze key system aspects, as demonstrated in Figures 7 and 8.

Additionally, other visualization techniques, such as augmented reality, 2D, and 3D models discussed in [6,19,31], can be applied. These approaches utilize different models, such as UML (unified modeling language) and graph models, to represent the constructed views as a dependency graph. Therefore, the proposed intermediate representation paves the way for advancing knowledge in the field and integrating multiple parts together for a more holistic view for practitioners.

Moreover, the proposed calculations can be integrated into continuous integration and continuous delivery (CI/CD) pipelines within practitioners' workspaces. This would provide insights into system evolution and offer visual representations for each new change before it is deployed to different environments through the CI/CD process.

Positioning Our Approach Among Existing Methods. It is important to compare and position our approach with existing methods, such as those in the study by Genfer et al. [29]. Their methods rely on using regular expressions to detect patterns in source code, which is effective for solving relatively straightforward problems. In contrast, our approach leverages static code analysis through source code parsers specifically designed for microservice development frameworks. Unlike regular expression-based methods, our approach focuses on the component-based architecture of microservices, where microservices are typically built around key components like controllers, repositories, and services. These concepts are largely language-agnostic, allowing us to build a more generalized and adaptable model [34]. Our method traverses an abstract syntax tree (AST) to identify system components and their interconnections. This enables us to approximate microservice dependencies by analyzing remote calls, endpoint signatures, and data overlaps. The resulting system representation is built from these interconnected components, providing a comprehensive view of the microservices and their relationships. Additionally, our approach allows for the inclusion of further artifacts, such as deployment descriptors and build files, ensuring a holistic system view.

Unlike other approaches in the literature, such as Tight et al. [44], which focus on issues like technical debt and antipatterns, our contribution presents holistic quantitative metrics to evaluate both the service and data viewpoints within cloud-native microservice systems. This enables more refined reasoning about system behavior and maintainability, which is a well-established area in monolithic systems but less explored in microservice architectures. Our approach provides developers with timely feedback, enabling them to assess the impact of their changes before deployment and ensuring system integrity throughout its evolution.

6. Threats to Validity

The potential validity threats consider Wohlins classification [45]. For *Construct Validity*, our methodology focuses on constructing service and data viewpoints. We utilize the TrainTicket testbench, a widely accepted benchmark in the microservices community.

In terms of *Internal Validity*, manual analysis for validating the extracted data is performed by the authors. To ensure unbiased analysis, the data validation and the prototype are executed by different authors. However, potential threats may arise from the testbench project's structure and conversion. For instance, some entities in v0.0.1 lack Lombok annotations, causing our methodology to miss them. This inconsistency can lead to inaccurate indicators of extracted data entities, necessitating further investigation. Also, the prototype does not detect event-based communication, although it does occur a few times within the assessed testbench, where the main connections primarily rely on REST endpoint calls. However, this does not affect the metrics' evolution, as these communication patterns remain consistent across different versions.

Regarding *External Validity*, our methodology offers general processes for constructing service and data viewpoints, applicable regardless of the programming language or framework. However, the implemented prototype is specific to the Java language and the Spring

Boot framework. Adapting it to different languages would require non-trivial changes to the underlying logic. Additionally, the choice of TrainTicket as the case study testbench is a limitation since being a testbench rather than a real-world microservices system somewhat limits its authenticity in reflecting actual system evolution. Nonetheless, we selected a broad range of system versions to emphasize the evolutionary changes.

Conclusion Validity is drawn from the case study's results, which illustrate the analysis capabilities of our method concerning specific architectural views. The case study encompasses multiple reasoning approaches, thereby validating and justifying our methodology for reasoning about architectural evolution. The results clearly demonstrate modifications in the system architecture and the resolution of multiple issues as the system evolves.

7. Conclusions

This paper addresses current gaps in the evolution of microservice systems by utilizing two established architectural viewpoints: service and data. It provides a detailed process for analyzing and quantifying system evolution. An intermediate representation was constructed to capture the essential attributes of both viewpoints. Seven metrics were defined to serve as indicators, providing a holistic understanding of system attributes and changes and enabling more effective reasoning about system evolution and version comparisons.

An evaluation case study was conducted to demonstrate and evaluate the methodology, emphasizing the significance of metrics in improving the understanding of system evolution properties. The study revealed that changes in system architecture can be approached from different levels of granularity, providing holistic insights for various reasoning perspectives. A visualization approach, adapted from the literature, was used to illustrate the intermediate representation, enabling human-centric reasoning and enhancing system analysis.

Future work will involve applying these metrics at various granularity levels, including architectural component granularity and change-level granularity. This approach will allow us to track the system's evolution with each modification and assess how these changes impact the metrics. Additionally, we plan to extend the intermediate representation to incorporate event-driven calls and dynamic data, providing a more detailed view of the system's behavior during runtime.

Furthermore, we aim to include other architectural viewpoints, such as technology and deployment viewpoints, to offer a more comprehensive understanding of the system by addressing additional aspects of its structure and operation. We will also explore the visualization methods previously highlighted, tailoring them to present information more effectively to practitioners and enhancing the overall usability of our approach representation.

Author Contributions: Conceptualization, A.S.A. and T.C.; methodology, A.S.A., T.C. and J.Y.S.; software, A.S.A.; validation, A.S.A., T.C. and J.Y.S.; formal analysis, A.S.A. and J.Y.S.; investigation, A.S.A.; resources, A.S.A.; data curation, A.S.A.; writing—original draft preparation, A.S.A.; writing—review and editing, A.S.A., T.C., X.L., D.T. and E.S.; visualization, A.S.A.; supervision, T.C.; project administration, A.S.A. and 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. OISE-1854049 and OISE-2409933.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original data presented in the study are openly available in Zenodo at https://zenodo.org/records/10052375 (accessed on 20 July 2024).

Conflicts of Interest: The authors declare no conflicts 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. Nadareishvili, I.; Mitra, R.; McLarty, M.; Amundsen, M. *Microservice Architecture: Aligning Principles, Practices, and Culture*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2016.
2. Bogner, J.; Fritzsch, J.; Wagner, S.; Zimmermann, A. Industry practices and challenges for the evolvability assurance of microservices. *Empir. Softw. Eng.* **2021**, *26*, 104. [CrossRef]
3. ISO/IEC/IEEE 42010:2022 Systems and Software Engineering—Architecture Description. 2022. Available online: https://www.iso.org/standard/74393.html (accessed on 20 July 2024).
4. Alshuqayran, N.; Ali, N.; Evans, R. Towards Micro Service Architecture Recovery: An Empirical Study. In Proceedings of the 2018 IEEE International Conference on Software Architecture (ICSA), Seattle, WA, USA, 30 April–4 May 2018; p. 47-4709. [CrossRef]
5. Rademacher, F.; Sachweh, S.; Zündorf, A. A Modeling Method for Systematic Architecture Reconstruction of Microservice-Based Software Systems. In *Enterprise, Business-Process and Information Systems Modeling*; Nurcan, S., Reinhartz-Berger, I., Soffer, P., Zdravkovic, J., Eds.; Springer International Publishing: Cham, Switzerland, 2020; pp. 311–326. [CrossRef]
6. Abdelfattah, A.S.; Cerny, T.; Taibi, D.; Vegas, S. Comparing 2D and Augmented Reality Visualizations for Microservice System Understandability: A Controlled Experiment. In Proceedings of the 2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC), Melbourne, Australia, 15–16 May 2023; pp. 135–145. [CrossRef]
7. 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]
8. Guo, G.Y.; Atlee, J.M.; Kazman, R. A software architecture reconstruction method. In Proceedings of the Working Conference on Software Architecture, San Antonio, TX, USA, 22–24 February 1999; Springer: Boston, MA, USA, 1999; pp. 15–33. [CrossRef]
9. Abdelfattah, A.S.; Cerny, T. Roadmap to reasoning in microservice systems: A rapid review. *Appl. Sci.* **2023**, *13*, 1838. [CrossRef]
10. Amoroso d'Aragona, D.; Li, X.; Cerny, T.; Janes, A.; Lenarduzzi, V.; Taibi, D. One microservice per developer: Is this the trend in OSS? In Proceedings of the European Conference on Service-Oriented and Cloud Computing, Larnaca, Cyprus, 24–25 October 2023; Springer: Cham, Switzerland, 2023; pp. 19–34. [CrossRef]
11. AWS Prescriptive Guidance. Service per Team. 2023. Available online: https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-decomposing-monoliths/service-per-team.html (accessed on 20 July 2024).
12. Lercher, A.; Glock, J.; Macho, C.; Pinzger, M. Microservice API Evolution in Practice: A Study on Strategies and Challenges. *J. Syst. Softw.* **2024**, *215*, 112110. [CrossRef]
13. Cerny, T.; Chy, M.; Abdelfattah, A.; Soldani, J.; Bogner, J. On Maintainability and Microservice Dependencies: How Do Changes Propagate? In Proceedings of the 14th International Conference on Cloud Computing and Services Science-CLOSER, Prague, Czech Republic, 1–3 May 2024; pp. 277–286. [CrossRef]
14. Lelovic, L.; Huzinga, A.; Goulis, G.; Kaur, A.; Boone, R.; Muzrapov, U.; Abdelfattah, A.S.; Cerny, T. Change Impact Analysis in Microservice Systems: A Systematic Literature Review. *J. Syst. Softw.* **2024**, *219*, 112241. [CrossRef]
15. Godfrey, M.W.; German, D.M. The past, present, and future of software evolution. In Proceedings of the 2008 Frontiers of Software Maintenance, Beijing, China, 28 September–4 October 2008; pp. 129–138. [CrossRef]
16. Baabad, A.; Zulzalil, H.B.; Hassan, S.; Baharom, S.B. Software Architecture Degradation in Open Source Software: A Systematic Literature Review. *IEEE Access* **2020**, *8*, 173681–173709. [CrossRef]
17. Aversano, L.; Guardabascio, D.; Tortorella, M. An empirical study on the architecture instability of software projects. *Int. J. Softw. Eng. Knowl. Eng.* **2019**, *29*, 515–545. [CrossRef]
18. Feng, Q.; Cai, Y.; Kazman, R.; Cui, D.; Liu, T.; Fang, H. Active hotspot: An issue-oriented model to monitor software evolution and degradation. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019; pp. 986–997. [CrossRef]
19. Cerny, T.; Abdelfattah, A.S.; Yero, J.; Taibi, D. From static code analysis to visual models of microservice architecture. *Clust. Comput.* **2024**, *27*, 4145–4170. [CrossRef]
20. Maffort, C.; Valente, M.T.; Terra, R.; Bigonha, M.; Anquetil, N.; Hora, A. Mining architectural violations from version history. *Empir. Softw. Eng.* **2016**, *21*, 854–895. [CrossRef]
21. Soldani, J.; Muntoni, G.; Neri, D.; Brogi, A. The μTOSCA toolchain: Mining, analyzing, and refactoring microservice-based architectures. *Softw. Pract. Exp.* **2021**, *51*, 1591–1621. [CrossRef]
22. Richner, T.; Ducasse, S. Recovering high-level views of object-oriented applications from static and dynamic information. In Proceedings of the IEEE International Conference on Software Maintenance-1999 (ICSM'99), 'Software Maintenance for Business Change' (Cat. No. 99CB36360), Oxford, UK, 30 August–3 September 1999; pp. 13–22. [CrossRef]
23. Riva, C.; Rodriguez, J.V. Combining static and dynamic views for architecture reconstruction. In Proceedings of the Sixth European Conference on Software Maintenance and Reengineering, Budapest, Hungary, 11–13 March 2002; pp. 47–55. [CrossRef]
24. Huang, G.; Mei, H.; Yang, F.Q. Runtime recovery and manipulation of software architecture of component-based systems. *Autom. Softw. Eng.* **2006**, *13*, 257–281. [CrossRef]
25. 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, 16–29 March 2018; pp. 21–30. [CrossRef]

26. 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. https://doi/org/10.1109/ICSME.2017.63.
27. de Freitas Apolinário, D.R.; de França, B.B.N. Towards a method for monitoring the coupling evolution of microservice-based architectures. In Proceedings of the 14th Brazilian Symposium on Software Components, Architectures, and Reuse, Natal, Brazil, 19–23 October 2020; pp. 71–80. [CrossRef]
28. Moreira, M.G.; De França, B.B.N. Analysis of Microservice Evolution using Cohesion Metrics. In Proceedings of the 16th Brazilian Symposium on Software Components, Architectures, and Reuse, Uberlandia, Brazil, 3–4 October 2022; pp. 40–49. [CrossRef]
29. Genfer, P.; Zdun, U. Exploring Architectural Evolution in Microservice Systems Using Repository Mining Techniques and Static Code Analysis. In *Proceedings of the Software Architecture*; Galster, M., Scandurra, P., Mikkonen, T., Oliveira Antonino, P., Nakagawa, E.Y., Navarro, E., Eds.; Springer: Cham, Switzerland, 2024; pp. 157–173. [CrossRef]
30. Rahman, M.I.; Panichella, S.; Taibi, D. A curated dataset of microservices-based systems. *arXiv* **2019**, arXiv:1909.03249. [CrossRef]
31. Oberhauser, R.; Pogolski, C. VR-EA: Virtual reality visualization of enterprise architecture models with ArchiMate and BPMN. In Proceedings of the International Symposium on Business Modeling and Software Design, Lisbon, Portugal, 1–3 July 2019; pp. 170–187. [CrossRef]
32. Abdelfattah, A.S.; Cerny, T. The Microservice Dependency Matrix. In Proceedings of the European Conference on Service-Oriented and Cloud Computing, Larnaca, Cyprus, 24–26 October 2023; pp. 276–288. [CrossRef]
33. 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]
34. Schiewe, M.; Curtis, J.; Bushong, V.; Cerny, T. Advancing Static Code Analysis with Language-Agnostic Component Identification. *IEEE Access* **2022**, *10*, 30743–30761. [CrossRef]
35. Zhou, X.; Peng, X.; Xie, T.; Sun, J.; Xu, C.; Ji, C.; Zhao, W. Benchmarking microservice systems for software engineering research. In Proceedings of the Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, 27 May–3 June 2018; ACM: New York, NY, USA, 2018; pp. 323–324. [CrossRef]
36. Spring Boot. 2005. Available online: https://spring.io/projects/spring-boot (accessed on 20 July 2024).
37. Project Lombok. 2009. Available online: https://projectlombok.org (accessed on 20 July 2024).
38. dmeoli. WS4J. 2018. Available online: https://github.com/dmeoli/WS4J (accessed on 20 July 2024).
39. Fellbaum, C. WordNet. In *Theory and Applications of Ontology: Computer Applications*; Poli, R., Healy, M., Kameas, A., Eds.; Springer: Dordrecht, The Netherlands, 2010; pp. 231–243. [CrossRef]
40. Cloudhubs. Prophet Web. A Tool for Cloud-Native Microservices Analysis. 2024. Available online: https://github.com/cloudhubs/prophet-web (accessed on 20 July 2024).
41. Walker, A.; Das, D.; Cerny, T. Automated Code-Smell Detection in Microservices Through Static Analysis: A Case Study. *Appl. Sci.* **2020**, *10*, 7800. [CrossRef]
42. Esparrachiari, S.; Reilly, T.; Rentz, A. Tracking and Controlling Microservice Dependencies: Dependency management is a crucial part of system and software design. *Queue* **2018**, *16*, 44–65. [CrossRef]
43. Cinque, M.; Cotroneo, D.; Della Corte, R.; Pecchia, A. A framework for on-line timing error detection in software systems. *Future Gener. Comput. Syst.* **2019**, *90*, 521–538. [CrossRef]
44. Tighilt, R.; Abdellatif, M.; Trabelsi, I.; Madern, L.; Moha, N.; Guéhéneuc, Y.G. On the maintenance support for microservice-based systems through the specification and the detection of microservice antipatterns. *J. Syst. Softw.* **2023**, *204*, 111755. [CrossRef]
45. Wohlin, C.; Runeson, P.; Hst, M.; Ohlsson, M.C.; Regnell, B.; Wessln, A. *Experimentation in Software Engineering*; Springer Publishing Company: Berlin/Heidelberg, Germany, 2012. [CrossRef]