



The Microservice Dependency Matrix

Amr S. Abdelfattah¹ and Tomas Cerny²

¹ Computer Science, Baylor University, One Bear Place, Waco, TX 97141, USA
amr_elsayed1@baylor.edu

² Systems and Industrial Engineering, University of Arizona, Tucson, AZ, USA
tcerny@arizona.edu

Abstract. Microservices have been recognized for over a decade. They reshaped system design enabling decentralization and independence of development teams working on particular microservices. While loosely coupled microservices are desired, it is inevitable for dependencies to arise. However, these dependencies often go unnoticed by development teams. As the system evolves, making changes to one microservice may trigger a ripple effect, necessitating adjustments in dependent microservices and increasing maintenance and operational efforts. Tracking different types of dependencies across microservices becomes crucial in anticipating the consequences of development team changes. This paper introduces the Endpoint Dependency Matrix (EDM) and Data Dependency Matrix (DDM) as tools to address this challenge. We present an automated approach for tracking these dependencies and demonstrate their extraction through a case study.

Keywords: Microservice Dependency · Static Analysis · Service Dependency · System Evolution · Automated Reasoning

1 Introduction

Microservice Architecture is widely used for complex systems that require selective scalability or the decomposition of complex organizational structures into smaller, independently managed units handled by separate development teams. As software systems evolve due to market demands, technological shifts, patches, or optimizations, new features are implemented, and bugs are fixed, potentially introducing new services and system dependencies [2]. Isolated modifications of individual services typically do not cause disruptions to others [8]. Nevertheless, as systems undergo evolution and dependencies naturally emerge within the architecture, posing challenges to the system's consistency and maintainability. Hence, it becomes crucial to proactively monitor and uphold the principles of low coupling and minimize dependencies within the architecture. In fact, consider a scenario where a critical bug is identified in a particular microservice. By accurately tracking the system dependencies, developers can confidently modify and debug the specific microservice without worrying about unintended consequences

or unintended disruptions to other interconnected services. This highlights the importance of actively managing and preserving a low-coupling architecture to ensure the long-term stability and scalability of microservice-based systems.

Recent studies highlight the lack of methods to prevent maintainability problems in microservices [1]. While existing metrics focus on direct dependencies introduced through endpoint calls between microservices, other aspects introduce dependencies too. For example, the presence of a common data model between microservices can lead to inconsistencies and coupling, where changes in one microservice may require modifications in others. This perspective provides another dimension to understanding the interconnectivity between microservices.

The main objective of this paper is to introduce and identify system dependencies at different perspectives, including direct endpoint calls and data dependencies, by analyzing the source code of microservices-based systems. We aim to offer a comprehensive understanding of service dependencies.

One of the key contributions of this paper is the development of an automated approach that extracts this dependency information directly from the codebase, ensuring that the obtained insights are up-to-date and free from outdated or stale information. The paper's contributions are summarized as follows:

- Describing automated approaches for constructing the Endpoint Dependency Matrix (EDM) and Data Dependency Matrix (DDM) of microservice-based systems.
- Implementing a prototype that applies the proposed approaches.
- Conducting a case study on a real public microservice project to generate the dependency matrices and discuss the results.

The paper is organized as follows. Section 2 presents the proposed method for constructing the dependency matrices. Section 3 presents the case study results for validation. Section 4 discusses the approach and potential threats to validity. Section 5 introduces related works. Finally, Sect. 6 concludes the paper.

2 The Proposed Dependency Methodology

The proposed method focuses on capturing the dependencies within microservice systems by considering both endpoints and data entities. Microservices systems utilize specialized frameworks to streamline the development of diverse capabilities. These frameworks often leverage object-oriented concepts and offer robust implementations. Through the utilization of static analysis techniques applied to the source code of the microservices, the necessary components are extracted to facilitate a comprehensive understanding of the system's dependencies.

To construct the EDM, the method identifies the direct endpoint calls within the source code, capturing the dependencies between microservices. The DDM is generated to represent the dependencies based on the shared data entities among microservices. By combining the information from EDM and DDM, a holistic depiction of the system's dependencies is achieved, providing insights into the flow of dependencies between both endpoints and data entities.

This approach serves as a valuable tool for practitioners to gain a comprehensive understanding of the intricate dependencies within microservice systems. By examining the system from both the endpoints and data perspectives, potential bottlenecks, inefficiencies, or critical dependencies can be identified, enabling better decision-making for system maintenance and evolution.

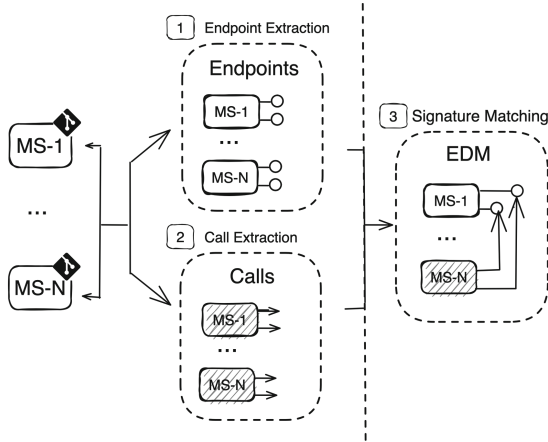


Fig. 1. Endpoint Dependency Matrix Generation Process.

2.1 Endpoint Dependency Matrix (EDM)

The dependency between endpoints reveals the interdependencies among different microservices, where one microservice’s source code contains a request call to an endpoint of another microservice. Our process examines the distributed source codes of microservices to extract the defined HTTP endpoints and request calls. This process consists of three phases, as depicted in Fig. 1: Endpoint Extraction, Call Extraction, and Signature Matching.

In the **Endpoint Extraction** phase, we identify and extract the HTTP endpoints defined in the source code. Typically, endpoints are specified using framework-specific functions or annotations. This approach ensures consistency in metadata identification. During this phase, we collect various attributes for each endpoint, including the path, HTTP method, parameters, and return type.

The **Call Extraction** phase focuses on extracting the requests made from the source code. By identifying the corresponding client, we determine where these endpoints are called from other services. Through code analysis, we can gather metadata about every call in the system by identifying the appropriate function call formats specific to the known HTTP library. Therefore, we extract the path, HTTP method, parameters’ values, and the expected return type.

The **Signature Matching** phase involves comparing endpoint method signatures with data and parameters exchanged during REST call interactions. This

process finds the matches between endpoint and request calls in the distributed source code. The collected endpoint and call details are merged to establish associations between calls and their corresponding endpoint components. However, direct matching is complex due to the endpoint definition including parameter data types, while request calls involve parameter values or variables in the request's body or path. Our approach initially considers path and parameter count matching. Subsequently, regular expressions are employed to identify the optimal match for parameter types with values in the calls. A successful match signifies a communication path between microservices via the matched endpoint.

Consequently, we can generate an EDM that illustrates the number of request calls between each pair of microservices in the system, thereby displaying the communication dependencies.

2.2 Data Dependency Matrix (DDM)

Each microservice establishes a data-bounded context that defines the scope where its specific domain model applies. To identify data dependencies, this method employs static analysis techniques to extract bounded contexts from each microservice's source code. It then proceeds to determine the correspondence between data entities across the individual bounded contexts. The construction process for data dependencies consists of three phases, as illustrated in Fig. 2: Components Extraction, Entity Filtration, and Entity Matching.

In the **Components Extraction** phase, all local classes declared in the project are extracted. Once these classes are identified, the **Entity Filtration** phase follows, which selects both Data Transfer Objects (DTOs) and classes representing persistent data. It focuses solely on data-related entities, excluding other classes like those serving as REST controllers or internal services. These two phases leverage enterprise standards and frameworks' components, such as annotation descriptors, to differentiate between class types based on their semantic purpose.

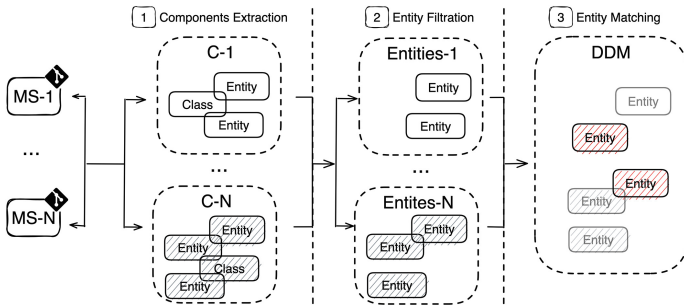


Fig. 2. Data Dependency Generation Process.

Finally, the **Entity Matching** phase examines all extracted entities across the microservices to generate a matching list between them. Different bounded

contexts may have distinct intentions for the shared entities, resulting in potential variations in the fields they retain. This phase matches entities based on their names, considering if they are the same or similar. Additionally, it examines whether some of their fields share the same data type and possess similar or identical names. This process yields the DDM, which provides insight into the common data entities among microservices.

3 Case Study

To demonstrate the effectiveness of our method, we apply it to a real-world scenario. It showcases the capabilities of capturing and understanding the dependencies present in microservice systems. Additionally, we seek to provide valuable insights into the interconnectedness of endpoints and data dependencies, leading to a comprehensive understanding of the system’s overall dependency landscape.

Our approach was implemented into a prototype, which we utilized to analyze a publicly available testbench. This allowed us to construct matrices depicting the dependencies of endpoints and data. The comprehension of system dependencies provided by these matrices serves as a valuable tool for facilitating seamless modifications and preserving the maintainability of the system. Moreover, these matrices play a crucial role in monitoring the evolution of dependencies throughout system changes. By generating and analyzing the dependency matrix, developers can track the impact of each commit on the system, observe how it affects system dependencies, and evaluate system coupling and stability. This enables informed decision-making and proactive management of dependencies, ultimately leading to a more robust and adaptable system architecture.

3.1 Prototype Implementation and Testbench

We developed a prototype¹ implementation of our proposed approach specifically designed for analyzing Java-based microservices projects utilizing the Spring Boot framework. The prototype utilizes Graal [5], the runtime system developed by Oracle Labs. The prototype takes a GitHub repository containing microservices-based projects as input. It downloads the repository and generates a list of directories for each microservice project.

For the **Endpoint Dependency Matrix**, the prototype scans the project files for JAX-RS annotations that define endpoints. By combining class-level and method-level annotations, it creates a comprehensive definition for each endpoint, including its path, HTTP method, parameters, and return type. The prototype also scans each microservice to identify the Spring Boot REST client (RestTemplate client) and detects HTTP calls between services. It then applies the signature matching technique to match the detected calls with the corresponding endpoints. The prototype generates a JSON structure that represents

¹ Prototype: <https://github.com/cloudhubs/graal-prophet-utils>.

the dependencies between microservices and the matched calls. Each microservice name serves as a key in the structure, containing two list values: Dependencies and Dependant services. These lists provide detailed information about the involved endpoints associated with each microservice node.

Regarding the **Data Dependency Matrix**, the prototype extracts all local classes in the project using a source code analyzer. It filters this list down to classes serving as data entities using persistence annotations (JPA standard entity annotations such as `@Entity` and `@Document`). It also considers annotations from Lombok², a tool for automatically creating data entity objects (e.g., `@Data`), although these annotations do not explicitly indicate persistence. The prototype then examines the entities of different bounded contexts and their fields, applying the matching rules described above. To detect name similarity, the prototype employs the WS4J³ project, which relies on the WordNet [4] dictionary. The prototype generates a JSON format, including entities and relationships for each microservice. It also presents a list of entities that provides a holistic context map of the system after eliminating duplicated matched entities.

Testbench: To demonstrate our case study, we utilized a public microservices testbench known as the train-ticket⁴ testbench system. It comprises 47 microservices, with 42 of them based on the Java-based Spring Boot framework. The system adheres to enterprise conventions by employing distinct controllers, services, and repositories for layering the application. Inter-service communication between microservices in the system is facilitated through REST API calls.

3.2 Results

The prototype was executed on the testbench to construct the endpoint and data dependencies. To ensure the data extraction's completeness, the prototype outcomes were manually validated. The resulting dependencies were analyzed separately and subsequently combined to form a comprehensive dependency view of the system. The heatmap is used as the visualization approach for the dependencies. Due to space constraints, the discussion refers to the microservices IDs listed in Table 1. For more detailed results, please refer to the provided dataset⁵.

Endpoint Dependency: The endpoint dependency matrix (EDM) is depicted in Fig. 3. The first column represents the microservices IDs containing request calls to the microservices listed in the first row. The values within each cell indicate the number of endpoint calls between each pair of microservices. Microservices containing no request calls to other microservices have been removed from the first column. This includes the following 16 microservices: 1, 4, 7, 9, 11, 13, 16 – 18, 20 – 22, 31, 32, 40, and 42. Similarly, microservices that do not have any request calls made to them have been eliminated from the

² Lombok: <https://projectlombok.org>.

³ WS4J: <https://github.com/Sciss/ws4j>.

⁴ Train-ticket V1.0.0: <https://github.com/FudanSELab/train-ticket/tree/v1.0.0>.

⁵ Dataset: <https://zenodo.org/record/8106860>.

first row, resulting in removing the following 16 microservices: 1, 19, 23, 24, 26 – 30, 32 – 35, 39, 41, and 42.

The dependency matrix showcases dependencies between multiple microservices, primarily consisting of one or two endpoint calls. However, there are four dependencies with a degree of three: $25 \rightarrow 18$, $27 \rightarrow 7$, $29 \rightarrow 17$, and $39 \rightarrow 36$. Notably, these dependencies originate from different microservices. The highest degree of dependencies observed is four, which occurs in seven pairs of microservices: $23 \rightarrow 6$, $27 \rightarrow \{9, 16, 18, 22\}$, and $28 \rightarrow \{14, 15\}$. The microservice **ts-admin-basic-info-service** (ID 27) exhibits a fourth-degree dependency on four distinct microservices, while the microservice **ts-admin-order-service** (ID 28) relies on the microservices **ts-order-other-service** (ID 14) and **ts-order-service** (ID 15), each with four endpoint calls.

Examining the longest rows containing values in the matrix reveals microservices with the highest number of dependencies, indicating that they make requests to a significant number of other microservices. For instance, the **ts-rebook-service** (ID 24) exhibits dependencies on eight different microservices, while the longest row belongs to **ts-preserve-other-service** (ID 34) and **ts-preserve-service** (ID 35) with eleven dependencies. On the other hand, analyzing the longest column highlights the microservices with the most dependants, meaning they receive requests from a greater number of microservices. The matrix indicates that **ts-route-service** (ID 17) and **ts-train-service** (ID 22) have a length of seven dependent microservices. However, the longest column contains eight dependants, which are microservices with IDs 14, 15, and 18.

Table 1. List of train-ticket microservices and associated IDs

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

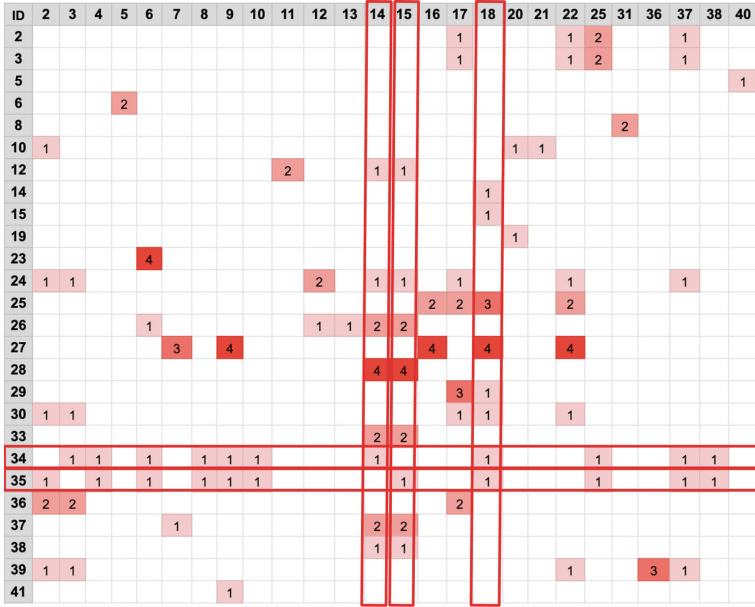


Fig. 3. Endpoint Dependency Matrix (EDM). The longest rows and columns are visually marked using a red rectangle. (Color figure online)

Table 2. Endpoints receiving more than three calls from other microservices.

ID	Endpoint Path	Method	#Calls	# μs
17	ts-route-service/api/v1/routeservice/routes	GET	8	7
18	ts-station-service/api/v1/stationservice/stations/id	GET	4	3
22	ts-train-service/api/v1/train-service/trains/byName	GET	6	6
25	ts-basic-service/api/v1/basic-service/basic/travel	POST	6	4

Further analysis delves into whether the dependants of a microservice make requests to the same endpoint or if they are spread across multiple endpoints within the microservice. The table presented in Table 2 highlights the endpoints that receive multiple requests from other microservices, specifically focusing on endpoints with more than three requests. It is important to note that not every call originates from a distinct microservice as shown in column ($\# \mu s$). Notably, the GET endpoint with the path `ts-route-service/api/v1/routeservice/route` receives eight calls from seven different microservices. This observation could indicate a potential functional bottleneck in the system, where multiple microservices rely on this endpoint to fulfill their respective use cases.

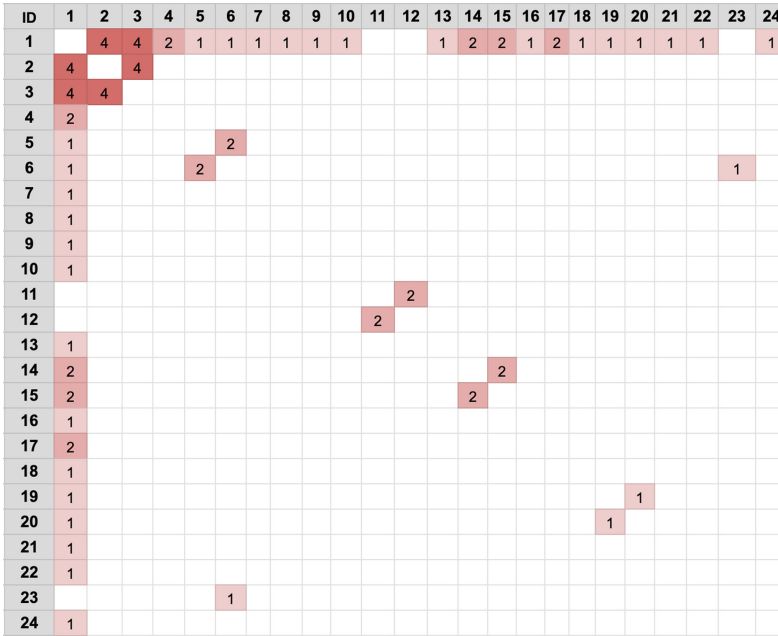


Fig. 4. Data Dependency Matrix (DDM). The longest rows and columns are visually marked using a red rectangle. (Color figure online)

Data Dependency: The data dependency matrix (DDM) in Fig. 4 represents the number of common data entities between microservice pairs. The rows and columns correspond to microservice IDs, while the cell values indicate the count of matched data entities. Unlike the endpoint dependency matrix (EDM), this matrix is symmetric and undirected, meaning the values remain the same regardless of whether one starts from the rows or columns. A total of 18 microservices (IDs 25–42) have been excluded from the rows and columns of the DDM because they do not share any common data entities with other microservices.

The matrix reveals that multiple microservices share one or two common data entities with other microservices. However, the maximum number of common entities between a pair of microservices is four, observed between **ts-common** (ID 1) and both **ts-travel-service** (ID 2) and **ts-travel2-service** (ID 3), and also between **ts-travel-service** (ID 2) and **ts-travel2-service** (ID 3).

Moreover, the longest row in terms of values belongs to **ts-common** (ID 1), indicating that this microservice shares the most common entities with other twenty microservices. However, the next longest row corresponds to **ts-user-service** (ID 6) with a length of only three, highlighting a significant disparity in data dependencies among the microservices, with a concentration of dependencies in a single microservice (**ts-common**). Upon further examination of the most common data entities across all microservices, we identified eight commonly shared entities: **AdminTrip**, **Order**, **OrderAlterInfo**,

StationFoodStore, Travel, Trip, TripAllDetail, and User. All these entities also exist in `ts-common`, but are shared only across three distinct microservices.

Comprehensive Service Dependency: By combining the EDM and the DDM, we generate a comprehensive perspective of the system’s dependencies known as the Service Dependency Matrix (SDM), as shown in Fig. 5. The SDM represents microservice IDs as both columns and rows. The cell values in the SDM are decimal numbers, where the integer part corresponds to the endpoint dependency degree from the EDM, and the fractional part corresponds to the data dependency degree from the DDM.

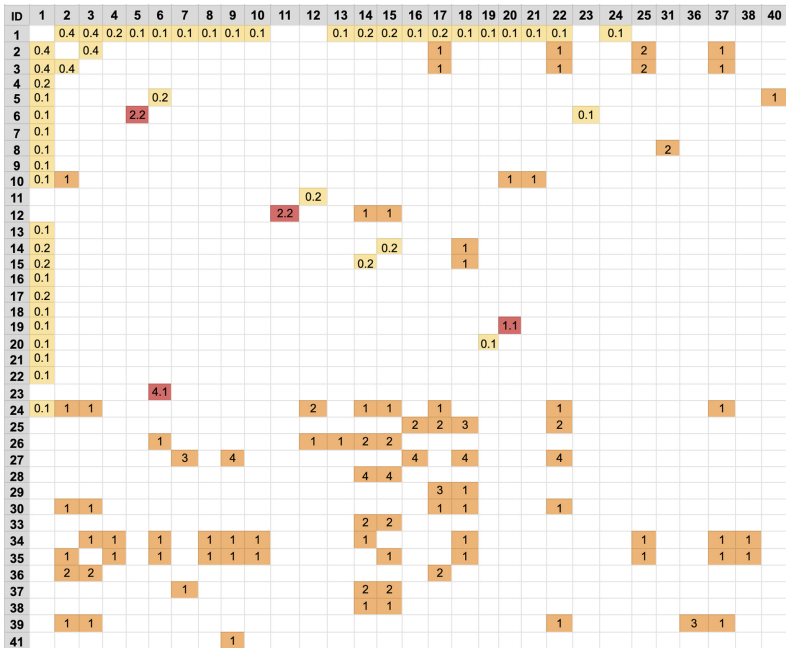


Fig. 5. Service Dependency Matrix (SDM).

To visually distinguish between different types of dependencies, the matrix utilizes different colors for endpoints-only dependencies, data-only dependencies, and dependencies involving both endpoints and data. The inclusion of data dependency in the fractional part of the SDM does not diminish its value compared to the endpoint dependencies. The construction of the decimal value is primarily related to the data formatting rather than the absolute significance of the cell value. For instance, consider the cell at position (row: 23, column: 6) in the SDM, which has a value of 4.1. This value indicates that **ts-admin-user-service** microservice (ID 23) has made four calls

to `ts-user-service` microservice (ID 6), and there is one common entity (`UserDto`) shared between them.

Analyzing the SDM, it becomes apparent that the responsibility of holding common data entities among microservices is predominantly concentrated in the `ts-common` microservice. This concentration results in distinct separations between the dependencies of endpoints and data entities. However, some overlaps can still be observed between the following four microservice pairs: $6 \rightarrow 5$, $12 \rightarrow 11$, $19 \rightarrow 20$, and $23 \rightarrow 6$. These pairs demonstrate a strong dependency within the system, as they depend on each other for both direct endpoint calls and the presence of common data entities. These dependencies highlight their interconnected nature and the importance of their mutual interaction.

4 Discussion

In the proposed method, we aim to provide a comprehensive understanding of system dependencies by considering both the endpoints and data perspectives. The introduced dependency matrices present system-centric perspectives that have the potential to provide a scalable visualization approach, helping practitioners in comprehending the system architecture and its dependencies. Blending endpoint dependencies (EDM) and data dependencies (DDM) within a unified matrix (SDM) has the potential to unveil more profound architectural concerns within microservices applications, surpassing what can be discerned from the separate EDM and DDM matrices. Moreover, by comparing the metrics across different versions, we can track the evolution of system dependencies over time.

While our method and prototype are valuable, it is important to acknowledge their limitations, particularly regarding the consideration of other perspectives of dependencies. The asynchronous communication model between microservices (e.g., publish-subscribe pattern), is not currently covered by our approach and they are not used in the train-ticket testbench as well. Incorporating such perspectives would provide additional insights into the interconnections between system components beyond the direct endpoint calls. Furthermore, this study focuses on analyzing the system's source code to gain a holistic understanding of all possible execution paths. However, considering the runtime interactions captured in logs and traces could provide valuable insights into the actual number of calls made to a particular microservice. This additional perspective could offer an additional depiction of the dependencies between microservices and enhance our understanding of the system's behavior.

Threats to Validity: The method does not address all potential microservice dependencies, its purpose is to illustrate how dependency matrices can assist in system analysis. Our prototype tool is tailored for the Java platform, potentially restricting its relevance to other programming languages. However, it's important to emphasize that the focus was on introducing the methodology rather than creating an exhaustive tool. In certain cases, the prototype tool might encounter challenges in accurately matching method signatures, particularly in situations where there are ambiguous method names. Additionally, the entity

matching process is currently restricted to basic similarities such as names and field matches, indicating that there are inherent limitations in approximation.

The case study analysis may be influenced by specific constructs present in the selected testbench, potentially limiting the prototype's generalizability across different systems. However, manual validation of the prototype's outcomes was performed to ensure the completeness of information extraction from the source code. Furthermore, the chosen testbench is employed in various research and is regarded as a well-established and representative microservice system.

5 Related Work

Numerous studies underscore the significance of managing dependencies in microservice architectures. According to Lewis and Fowler [8], loosely coupled microservices offer advantages in independent modifications but pose challenges as systems evolve. To analyze such dependencies, scholars have introduced various techniques. Apolinário et al. [1] focus on endpoint calls, Sangal et al. [11] employ static analysis for dependency models, and Eski and Buzluca [6] use evolutionary code coupling. Our approach uses static analysis to extract and integrate both endpoint and data dependencies for a comprehensive system view.

In the realm of heterogeneous dependencies in distributed systems, Fang et al. [7] devised specialized tools for compile-time dependency extraction through static analysis. They targeted entity dependencies within components and hard-coded API dependencies, using text comparison. In contrast, our method goes beyond text-based analysis, incorporating semantic similarities and fine-grained dependency capture through signature matching.

Effective visualization is crucial for comprehending system dependencies. Multiple studies [3, 9, 10] propose graph-based visualizations depicting microservice dependencies, focusing on communication patterns via endpoint calls. In contrast, our approach employs dependency matrices to visualize and analyze the system, offering a distinct view of microservices' dependencies.

6 Conclusion

System dependency analysis in microservices provides valuable insights for practitioners to comprehend the system. This paper integrates endpoint and data dependencies, offering a comprehensive understanding of system dependencies and facilitating informed decision-making in developing and evolving microservice-based systems. The analysis is addressed through static code analysis providing perspectives that enable reasoning about system maintainability and monitoring system dependency evolution across different versions. Our approach encompasses a detailed analysis of individual microservices, combining the results to a holistic dependency perspectives that can be visualized and interpreted. The focus was on generating the EDM and DDM from the source code and further combining them to create the SDM for a more comprehensive perspective. The proposed methodology was implemented in a prototype

and validated through a case study, highlighting its efficacy in understanding system dependencies.

Future work will include asynchronous call dependencies, recognizing their importance. We also aim to expand the prototype for analyzing system polyglots.

Acknowledgement. This material is based upon work supported by the National Science Foundation under Grant No. 2245287.

References

1. Apolinário, D.R.F., de França, B.B.N.: A method for monitoring the coupling evolution of microservice-based architectures. *J. Braz. Comput. Soc.* **27**(1), 1–35 (2021). <https://doi.org/10.1186/s13173-021-00120-y>
2. Cerny, T., Abdelfattah, A.S., Bushong, V., Al Maruf, A., Taibi, D.: Microservice architecture reconstruction and visualization techniques: a review. In: 2022 IEEE International Conference on Service-Oriented System Engineering (SOSE), pp. 39–48. IEEE (2022)
3. Cerny, T., Abdelfattah, A.S., Bushong, V., Al Maruf, A., Taibi, D.: Microvision: static analysis-based approach to visualizing microservices in augmented reality. In: 2022 IEEE International Conference on Service-Oriented System Engineering (SOSE), pp. 49–58. IEEE (2022)
4. Christiane, F., Brown, K.: Wordnet and wordnets. In: *Encyclopedia of Language and Linguistics*, pp. 665–670. Oxford: Elsevier (2005)
5. Duboscq, G., Stadler, L., Würthinger, T., Simon, D., Wimmer, C., Mössenböck, H.: Graal IR: an extensible declarative intermediate representation. In: *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, pp. 1–9 (2013)
6. Eski, S., Buzluca, F.: An automatic extraction approach: transition to microservices architecture from monolithic application. In: *Proceedings of the 19th International Conference on Agile Software Development: Companion*, pp. 1–6 (2018)
7. Fang, H., Cai, Y., Kazman, R., Lefever, J.: Identifying anti-patterns in distributed systems with heterogeneous dependencies. In: 2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C), pp. 116–120 (2023)
8. Lewis, J., Fowler, M.: Microservice. <https://www.martinfowler.com/articles/microservices.html>. Accessed 13 Dec 2022
9. Oberhauser, R., Pogolski, C.: VR-EA: virtual reality visualization of enterprise architecture models with archimate and BPMN. In: Shishkov, B. (ed.) *BMSD 2019*. LNBIP, vol. 356, pp. 170–187. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24854-3_11
10. Rahman, M.I., Panichella, S., Taibi, D.: A curated dataset of microservices-based systems. In: *SSSME-2019* (2019)
11. Sangal, N., Jordan, E., Sinha, V., Jackson, D.: Using dependency models to manage complex software architecture. In: *20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 167–176 (2005)