Data Access-centered Understanding of Microservices Architectures

André, Maxime; Rivière, Etienne ; Cleve, Anthony

Published in:
Proceedings of the 22nd IEEE International Conference on Software Architecture (ICSA 2025)

Publication date:
2025

Document Version
Peer reviewed version

Data Access-centered Understanding of Microservices Architectures

Maxime André
Namur Digital Institute
University of Namur
Namur, Belgium
maxime.andre@unamur.be

Etienne Rivière
ICTEAM
UCLouvain
Louvain-la-Neuve, Belgium
etienne.riviere@uclouvain.be

Anthony Cleve
Namur Digital Institute
University of Namur
Namur, Belgium
anthony.cleve@unamur.be

***Abstract*—Recent studies show that developers encounter difficulties when evolving microservices architectures, especially from a code and data management perspective. Specifically, they struggle to recover a high-level view, although essential for understanding and maintaining complex interactions across various microservices codebases and their databases.**

This paper introduces an approach for systematically analyzing data access code fragments in complex applications composed of multiple microservices and distributed across several codebases. By combining heuristic-based code analysis with natural language processing, the approach holistically identify, extracts, interprets, and documents the interactions between these microservices and their databases. The resulting report is designed to support software evolution tasks, such as locating or visualizing linked data access code fragments.

A preliminary evaluation on 5 JavaScript microservices architectures with MongoDB and Redis provides emerging results.

***Index Terms*—microservices architectures, databases, data access analysis**

I. Introduction

Microservices as a software architecture pattern has significantly gained popularity over the past decade. This software architectural model is now widely adopted by large, software-intensive companies like *Amazon*, *Google*, or *Netflix* [1], [2].

In contrast with monolithic architectures, microservices architectures are praised for their qualities of *modularity*, *heterogeneity*, and *interoperability* [3]. From a data management perspective, however, microservice architectures introduce an additional level of complexity. As reported by Laigner *et al.* [4], *polyglot persistence* [5] and complex interactions, incarnated by multiple, heterogeneous, and distributed databases, complicate the recovery of high-level and holistic views of the system. Nonetheless, establishing such views accurately is essential for supporting software evolution. It ensures a completeness in tasks like re-documentation, visualization, quality assessment, evolution recommendations, or impact analysis. Otherwise, developers are required to *know* data access code fragments that depend on the change operated. Or, they may have to manually *search* through the entire codebase to identify potentially impacted code fragments. This process is time-consuming, error-prone, and cumbersome [6], especially in large codebases residing in multiple repositories, with microservices accessing multiple databases.

Developers require a comprehensive view of microservices applications, including the data layer, to effectively understand, maintain, and optimize interactions in evolving software.

Unfortunately, large language models can lead to certain problems in software comprehension [7]. Traditional works on code or runtime analysis neglects the data layer and multiple codebases [8], [9]. Finally, existing works on database access analysis [10], [11] are unsuitable for large microservices involving several databases.

In this paper, our main question is *"How to statically recover a view of data access in a microservices architecture?"*. By generating a detailed report analyzing API and database interactions between microservices, our proposed approach considers both code and data layers as suggested by Cerny *et al.* [9]. Through a heuristic-based abstract syntax tree analysis, we automatically identify the data access code fragments of a given microservices architecture, *i.e.,* the set of instructions in the source code where data is accessed, both at the API and database query levels. By leveraging natural language processing, we extract and interpret data access code fragments for producing a detailed report documenting them and related data concepts (*i.e.,* data entities). This emerging result constitutes a promising basis for software evolution tasks, such as re-documentation, visualization, quality assessment, evolution recommendations, or impact analysis on change as illustrated in Figure 1.

In Section II, we present each step of our approach. A preliminary evaluation of its current implementation [12] for JavaScript microservices architectures with MongoDB and Redis is presented in Section III, and discussed in Section IV. Section V anticipates our future plans. Section VI positions our work within the related literature.

II. Approach

Our approach takes as input one or more microservice repositories and automatically identifies API invocations and database access code fragments. It further extracts data to characterize these fragments based on their similarities, enabling the interpretation and analysis of related and linked code fragments. Finally, the approach generates a comprehensive and detailed report as output. Our approach relies on a 6-step process, as illustrated in Figure 2.

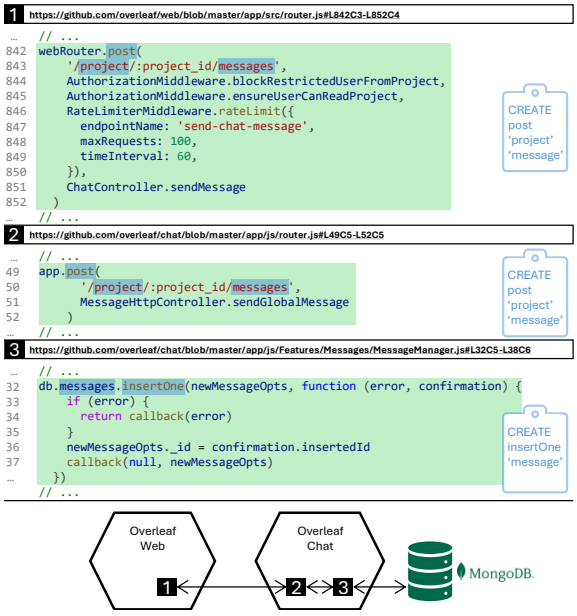

Fig. 1. This example illustrates the process of locating linked code fragments across separate microservices that are likely to co-evolve during change propagation. A change made to the first code fragment should be propagated to the other two. In this case, our approach contributes to automatically identify code fragments sharing similarities (*e.g.,* 'CREATE' operation, 'message' concept) and thus to facilitate co-evolution tasks.

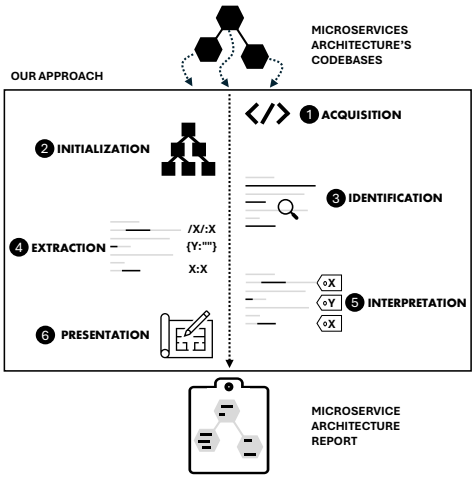

Fig. 2. Overview of our 6-step approach.

Acquisition. Step ① retrieves in one go all the code repositories corresponding to the microservices architecture of interest.
Initialization. Step ② parses the source code of each repository to create a queryable Abstract Syntax Tree (AST). We rely on static program analysis to obtain comprehensive internal details at the source code granularity level without being restricted to particular program execution scenarios [13], [14]. Moreover, we do not require deploying or running the complete microservices application [15]. This is aligned with DevOps principles [8] and helps practitioners commit to an architecture whose deployment details are not known [8], [9], [13], [14], [16], [17]. However, static analysis brings challenges. As of now, we focus on popular languages like JavaScript [18]. Those are dynamically interpreted, which complicates static analysis [19], [20]. To tackle this, we rely on *GitHub*'s *CodeQL* [21]. This mature and reliable solution is particularly helpful as it offers predefined and customizable queries for deepening the analysis.
Identification. Step ③ identifies, through heuristics matching, the code fragments that invoke a specific API endpoint or perform a database access. Indeed, despite robust tools like *CodeQL*, static analysis of highly dynamic languages remains challenging due to the lack of type checking or parameter value interpretation. As illustrated in Figure 3, our solution identifies a set of AST nodes, *i.e.,* code fragments, satisfying several heuristics. The heuristics, presented in Table I, are rules on the source code defined in accordance with official documentation of the programming language (*i.e.,* JavaScript), the architectural style (*i.e.,* REST by *ExpressJS* [22]), and database technologies (*i.e., MongoDB* [23] and *Redis* [24]) we currently support. The heuristics go further than merely text-based filtering on the code. The code fragments are identified through pattern matching (*e.g.,* method names, argument value patterns) like M1 and M3, and rules matching (*e.g.,* number of arguments, inferred types, code structure, and dependencies inside the code) like M2, M4, M5, and M6. We compute a likelihood score for each candidate code fragment equal to the number of matching heuristics. Combining those matching heuristics aims to reduce false positives (towards higher precision, *i.e.,* ratio between relevant code fragments identified on the total number of code fragments identified) while ignoring non-matching heuristics aims to reduce false negatives (towards higher recall, *i.e.,* ratio between relevant code fragments identified on the total number of relevant code fragments to identify). This flexibility is required by the highly dynamic nature of the programming languages used in microservices [18]. In this context, each analysis can set a minimum threshold representing the optimal balance between precision and recall. Only candidate code fragments that reach the minimum threshold score are selected. Figure 3 gives an example where a string-based search on some keywords like method call names is insufficient. Indeed, 'find' is a popular JavaScript method used in many other contexts. A string-based filtering would lead to many wrong or missing source code locations. This shows the importance of combining various heuristics to distinguish code candidates.

Fig. 3. A code fragment and its data-related samples.

TABLE I
IDENTIFICATION HEURISTICS FOR EXPRESSJS (E*), MONGODB (M*), AND REDIS (R*) CODE FRAGMENTS.

ID	Description
E1	Has an ExpressJS-like method name (*e.g.,* 'post', 'put').
E2	Has a string as the first argument.
E3	Has an ExpressJS route-like string as first argument.
E4	Has a function as a second argument.
E5	Has an ExpressJS-like receiver name (*e.g.,* 'app').
E6	Has an ExpressJS-like import around.
E7	Has an ExpressJS-like client assignment around.
E8	Is linked to an ExpressJS-like client assignment around.
M1	Has a MongoDB-like method name (*e.g.,* 'findOne').
M2	Has a string or an object as the first argument.
M3	Has a MongoDB-like receiver name (*e.g.,* 'db', 'collection').
M4	Has a MongoDB-like import around.
M5	Has a MongoDB-like client assignment around.
M6	Is linked to a MongoDB-like client assignment around.
R1	Has a Redis-like method name (*e.g.,* 'scan', sadd', 'rpush').
R2	Has a string as the first argument.
R3	Has an ExpressJS-like receiver name (*e.g.,* 'client').
R4	Has a Redis-like import around.
R5	Has a Redis-like client assignment around.
R6	Is linked to a Redis-like client assignment around.

Extraction. Step ④ extracts data-related samples (*i.e.,* subset of the identified code fragments). They are intended to provide insights into the microservice data and related data access operations, as illustrated in blue in the Figure 3. The extracted samples are generally method receivers, method names, and method arguments if they meet certain conditions. The extraction mechanism leverages condition-identified AST nodes through custom *CodeQL* queries.

Interpretation. Step ⑤ interprets the data-related samples extracted at Step ④, using natural language processing (NLP). We lemmatize, clean and unify similar tokens to infer common data concepts (as illustrated on the right of Figure 1). In this work, we use two NLP libraries, *Natural* [25] and *winkJS* [26].

Presentation. Step ⑥ produces a report documenting all data accesses in the complete microservices architecture. The report follows an underlying model, shown in Figure 4. We represent a microservices architecture as a set of repositories subdivided into directories and files containing collections of code fragments. We enrich each code fragment with additional details extracted during static analysis, such as lines of code (LoC), the technology used for data access, the associated Create, Read, Update, or Delete (CRUD) operation, the specific Object-Relational Mapping (ORM) method employed, and, when available, a sample of the data objects or values affected by the operation. Finally, each code fragment may be linked to one or more data concepts extracted at Step ⑤.

III. PRELIMINARY EVALUATION

Microservices architecture selection. We evaluate our approach on 5 different microservices architectures[1] found in benchmarks [27], [28] and on *GitHub*. We target projects in JavaScript relying on MongoDB and/or Redis databases,

[1]https://github.com/overleaf — https://github.com/instana/robot-shop — https://github.com/dev-mastery/comments-api — https://github.com/crizstian/cinema-microservice — https://github.com/CloudBoost/cloudboost

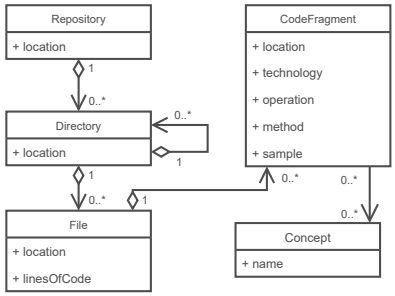

Fig. 4. Model of data access code fragments in a microservices architecture.

updated at least after 2022, with a minimum size of 500 kB, reaching a minimum of 500 stars, and maintained by more than 1 contributor. The codebases sizes range from 6 to 100 kLoC.

Annotation. In order to establish a ground truth, we manually annotate the source code (tests excluded) of each codebase looking for code fragments related to data accesses in each target technology. We manually identified a total of 694 code fragments. For each identified fragment we reported detailed information (*i.e.,* location URL, access operation, sample extracted, matching heuristics and score) in a spreadsheet (§VII).

Heuristics evaluation. For evaluating the individual relevance of our code fragment identification heuristics, we compute separately their precision and recall for each project codebase, based on the ground truth. We remove heuristics that make no contribution to the score.

Code fragments identification evaluation. For evaluating the performance of our approach, we compare the report automatically obtained by our implementation with the ground truth. For each project, we compute the precision and the recall depending on the minimum score threshold set.

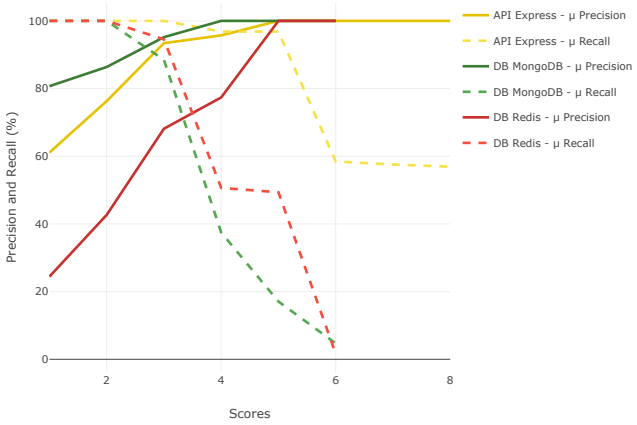

Fig. 5. Mean trend (μ) of precision and recall in code fragments identification, depending on the minimum score threshold.

We aggregate the results in Figure 5 to highlight trends in the evolution of precision and recall as a function of score thresholds. Lines represent code fragments identification for *ExpressJS* in yellow, for *MongoDB* in green, and for *Redis* in red. Note that the curves for MongoDB and Redis stop at a

score threshold of 6 because these technologies are associated with 6 heuristics, for a maximum score of 6, while ExpressJS is associated with 8 heuristics.

We see that the higher the score threshold, the higher the precision, but the lower the recall. This is not surprising, as a higher score threshold corresponds to a strict combination of heuristics, which causes a certain level of false negatives (more silence) for the benefit of a higher precision (less noise). Overall, this shows the relevance of our approach.

In particular, regarding the recall we observe a stable mean trend of 100% up to a score threshold of 2 for all groups of heuristics. This means that a minimum score of 2 could be set for all identification without negatively impacting the recall (no false negatives). Developers can therefore expect better identification results than via a simple string search, since every identified fragment corresponds to a minimum of 2 matching heuristics.

Finally, we can observe that, on average, the optimal precision and recall are respectively 100% and 96.84% with a minimum score of 5 for ExpressJS, 95.19% and 88.48% with a minimum score of 3 for MongoDB, and 68.13% and 94.55% with a minimum score of 3 for Redis. Those are average and can be refined depending on the analyzed project by adapting minimum scores for each technologies. More detailed results, by project and by heuristics, are available in our companion repository (see link in Section VII).

IV. Discussion

Our preliminary evaluation leads to promising results, yet it currently focuses on particular languages and technologies. We target JavaScript REST APIs implemented using the popular ExpressJS framework and relying on MongoDB and Redis databases. According to the literature [4], [8], [20], [22]–[24], [29], [30], these technologies are widely used in microservice architectures. Nevertheless, our approach must be generalized to cover other languages, architectural styles, APIs, and database technologies. This will require the translation of our identification heuristics or the implementation of new ones.

Our preliminary evaluation is also limited in its representativeness, as we considered 5 microservices applications. We relied on existing benchmarks to select recent, diverse, and popular repositories, but it is possible to consider a broader set of microservices architectures.

Finally, conducting a dedicated user study involving professional microservices developers would help us further assess our approach in real-life usage scenarios.

V. Future Plans

Our future directions will first focus on extending the languages and database technologies our static analysis approach supports. We will rely on the *CodeQL* extensions already available for other popular languages (*e.g.,* Java, Python, C#). As heuristics are defined as rules on source code, they are generalizable and transposable to other languages. We will also implement new heuristics to improve identification performance.

In addition, we will evaluate new increments with a more extensive set of representative microservices architectures.

We also plan to offer interactive visualizations of the output documentation report to increase user-friendliness.

Finally, we plan to enrich our output model with runtime information to provide a more comprehensive view of the microservices architecture of interest [15].

VI. Related Work

Several authors highlight the importance of understanding microservices architectures [8] with code and database accesses analysis to support software evolution [4]. This section summarizes related works to position the novelty of our.

Static analysis of microservices architectures. In a study [15], Bushong *et al.* review works aiming to retrieve, analyze, understand, and explain microservices architectures statically, especially while deriving structural dependencies, examining code, and reconstructing a holistic centralized view with tools like *MSANose* [31] and *MICROLYZE* [32].

In a review on works detecting microservice API patterns, Bakhtin *et al.* [13] show that static analysis is popular.

Cerny *et al.* also argue that static analysis deserves more attention [9], [16]. In DevOps, it can be used for analysis at a particular level of abstraction, helping developers to identify problems before deployment. In a follow-up work [17], they use it to reconstruct a microservices architecture regarding endpoints and messages exchanged.

Singh *et al.* automatically document Message-oriented Middleware-based microservices with source code static analysis to extract component-based architectures and their behavior in asynchronous communication [33].

Other approaches use static analysis with dynamic analysis for architecture reconstruction. For instance, Mayer *et al.* extract service and API descriptions from configuration files [14].

Unfortunately, these works limit their scope to the API level, ignoring the data perspective. Moreover, they are generally limited to a single codebase. Finally, for those relying on dynamic analysis, live deployment is required, which may affect results, as it does not guarantee covering the entire application architecture.

Static analysis of database accesses. Meurice *et al.* offer a static analyzer for pointing source code locations of dynamically generated SQL queries by JDBC, Hibernate and JPA [34]. They also propose an approach to detect inconsistencies between database queries and their evolving schema [35].

Nagy *et al.* propose *SQLInspect*, a tool able to statically extract and assess SQL queries from Java programs [36].

Liu *et al.* present *SLocator*, a similar approach combining static analysis and information retrieval, to locate the origin of SQL queries generated by the JPA ORM framework [11].

Scherzinger *et al.* propose *ControVol*, a framework supporting schema evolution with static type checking in Java applications relying on NoSQL data stores [10].

Meurice *et al.* propose a support tool based on static analysis on Java applications for inferring the implicit schema(s) of NoSQL data stores like MongoDB [37].

Regrettably, these works mainly focus on architectures involving a single database or codebase, losing the big picture, which is unsuitable for large microservices.

VII. Conclusions

While microservices architectures are gaining popularity, recent studies report on difficulties and challenges faced by practitioners. They need help to understand such architectures from a code and data management perspective, affecting software evolution tasks. This is due, among other things, to the heterogeneous and distributed nature of the underlying databases and their access code fragments.

To address this problem, we propose a heuristics-based approach for statically identifying and analyzing data access code fragments in codebases of microservices architectures. Its goal is to provide developers with a holistic view facilitating the evolution of data access in such architectures, e.g., in the context of impact analysis and change propagation.

Our preliminary evaluation shows promising results for JavaScript REST APIs relying on NoSQL databases.

Companion repository

The open-source implementation of our approach, our complete preliminary evaluation results, and examples are publicly available in our companion repository: https://doi.org/10.5281/zenodo.14740539.

Acknowledgments

This research is supported by the Federation Wallonie-Bruxelles (FWB), as part of the ARC project RAINDROP.

References

[1] C. Richardson, *Microservices Patterns: with Examples in Java*. Simon and Schuster, 2018.

[2] S. Newman, *Building Microservices*. O'Reilly Media, Inc., 2021.

[3] M. André, "Automated Database Schema Evolution in Microservices," in *Proc. of VLDB 2023*, vol. 3452. CEUR-WS, 2023, pp. 37–40.

[4] R. Laigner, Y. Zhou, M. A. V. Salles, Y. Liu, and M. Kalinowski, "Data Management in Microservices: State of the Practice, Challenges, and Research Directions," *VLDB Endowment*, vol. 14, no. 13, pp. 3348–3361, 2021.

[5] J. Lewis and M. Fowler, "Microservices," 2014, https://martinfowler.com/articles/microservices.html.

[6] A. Lercher, J. Glock, C. Macho, and M. Pinzger, "Microservice API Evolution in Practice: A Study on Strategies and Challenges," *Journal of Systems and Software*, vol. 215, p. 112110, 2024.

[7] T. Lehtinen, C. Koutcheme, and A. Hellas, "Let's Ask AI About Their Programs: Exploring ChatGPT's Answers To Program Comprehension Questions," in *Proc. of the ICSE-SEET 2024*. Association for Computing Machinery, 2024, pp. 221–232.

[8] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, "Towards Recovering the Software Architecture of Microservice-based Systems," in *Proc. of ICSAW 2017*. IEEE, 2017, pp. 46–53.

[9] T. Cerny and D. Taibi, "Static Analysis Tools in the Era of Cloud-native Systems," in *Proc. of Microservices 2022*, 2022.

[10] S. Scherzinger, T. Cerqueus, and E. C. De Almeida, "Controvol: A Framework for Controlled Schema Evolution in NoSQL Application Development," in *Proc. of ICDE 2015*. IEEE, 2015, pp. 1464–1467.

[11] W. Liu and T.-H. Chen, "SLocator: Localizing the Origin of SQL Queries in Database-Backed Web Applications," *IEEE Transactions on Software Engineering*, vol. 49, no. 6, pp. 3376–3390, 2023.

[12] M. André, E. Rivière, and A. Cleve, "DENIM Reverse Engineering," Jan. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.14740539

[13] A. Bakhtin, A. Al Maruf, T. Cerny, and D. Taibi, "Survey on Tools and Techniques Detecting Microservice API Patterns," in *Proc. of SCC 2022*. IEEE, 2022, pp. 31–38.

[14] B. Mayer and R. Weinreich, "An Approach to Extract the Architecture of Microservice-based Software Systems," in *Proc. of SOSE 2018*. IEEE, 2018, pp. 21–30.

[15] V. Bushong, A. S. Abdelfattah, A. A. Maruf, D. Das, A. Lehman, E. Jaroszewski, M. Coffey, T. Cerny, K. Frajtak, P. Tisnovsky *et al.*, "On Microservice Analysis and Architecture Evolution: A Systematic Mapping Study," *Applied Sciences*, vol. 11, no. 17, p. 7856, 2021.

[16] T. Cerny, A. S. Abdelfattah, V. Bushong, A. Al Maruf, and D. Taibi, "Microservice Architecture Reconstruction and Visualization Techniques: A Review," in *Proc. of SOSE 2022*. IEEE, 2022, pp. 39–48.

[17] ——, "Microvision: Static Analysis-based Approach to Visualizing Microservices in Augmented Reality," in *Proc. of SOSE 2022*. IEEE, 2022, pp. 49–58.

[18] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proc. of ASPLOS 2019*. Association for Computing Machinery, 2019, pp. 3–18.

[19] B. Cherry, P. Benats, M. Gobert, L. Meurice, C. Nagy, and A. Cleve, "Static Analysis of Database Accesses in MongoDB Applications," in *Proc. of SANER 2022*. IEEE, 2022, pp. 930–934.

[20] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, Today, and Tomorrow," *Present and Ulterior Software Engineering*, pp. 195–216, 2017.

[21] GitHub, "About CodeQL — CodeQL," https://codeql.github.com/docs/codeql-overview/about-codeql/, 2024.

[22] ExpressJS, "Fast, Unopinionated, Minimalist Web Framework for Node," https://github.com/expressjs/express, 2024.

[23] MongoDB, "MongoDB: The Developer Data Platform," https://www.mongodb.com/, 2024.

[24] Redis, "Redis - The Real-time Data Platform," https://redis.io/, 2024.

[25] NaturalNode, "Natural," https://naturalnode.github.io/natural/, 2024.

[26] graype systems, "winkJS — NLP, Machine Learning & Stats in Node.js," https://winkjs.org/, 2024.

[27] M. I. Rahman, S. Panichella, and D. Taibi, "A Curated Dataset of Microservices-based Systems," in *Proc. SSSME 2019*. CEUR-WS, 2019, pp. 1–9.

[28] D. Amoroso d'Aragona, A. Bakhtin, X. Li, R. Su, L. Adams, E. Aponte, F. Boyle, P. Boyle, R. Koerner, J. Lee *et al.*, "A Dataset of Microservices-based Open-source Projects," in *Proc. of MSR 2024*, 2024, pp. 504–509.

[29] M. Garriga, "Towards a Taxonomy of Microservices Architectures," in *International Conference on Software Engineering and Formal Methods (SEFM)*. Springer, 2018, pp. 203–218.

[30] L. P. Tizzei, L. Azevedo, E. Soares, R. Thiago, and R. Costa, "On the Maintenance of a Scientific Application based on Microservices: an Experience Report," in *Proc. of ICWS 2020*. IEEE, 2020, pp. 102–109.

[31] A. Walker, D. Das, and T. Cerny, "Automated Code-smell Detection in Microservices Through Static Analysis: A Case Study," *Applied Sciences*, vol. 10, no. 21, p. 7800, 2020.

[32] M. Kleehaus, Ö. Uludağ, P. Schäfer, and F. Matthes, "MICROLYZE: A Framework for Recovering the Software Architecture in Microservice-based Environments," in *Information Systems in the Big Data Era*, vol. 317. Springer, 2018, pp. 148–162.

[33] S. Singh and A. Koziolek, "Automated Reverse Engineering for MoM-Based Microservices (ARE4MOM) Using Static Analysis ," in *Proc. of ICSA 2024*. IEEE Computer Society, 2024, pp. 12–22.

[34] L. Meurice, C. Nagy, and A. Cleve, "Static Analysis of Dynamic Database Usage in Java Systems," in *Proc. of CAiSE 2016*. Springer, 2016, pp. 491–506.

[35] ——, "Detecting and Preventing Program Inconsistencies Under Database Schema Evolution," in *Proc. of QRS 2016*. IEEE, 2016, pp. 262–273.

[36] C. Nagy and A. Cleve, "SQLInspect: A Static Analyzer to Inspect Database Usage in Java Applications," in *Proc. of ICSE 2018*, 2018, pp. 93–96.

[37] L. Meurice and A. Cleve, "Supporting Schema Evolution in Schema-less NoSQL Data Stores," in *Proc. of SANER 2017*. IEEE, 2017, pp. 457–461.