Exploring Architectural Evolution in Microservice Systems using Repository Mining Techniques and Static Code Analysis

Patric Genfer[1,2] and Uwe Zdun[1]

[1] Research Group Software Architecture, Faculty of Computer Science, University of Vienna, Vienna, Austria
{patric.genfer|uwe.zdun}@univie.ac.at
[2] UniVie Doctoral School Computer Science DoCS, University of Vienna, Vienna, Austria

Abstract. Microservices have gained popularity for isolating service functionality and mitigating issues such as architectural erosion and technical debt. However, their decentralized nature and rapid development often obscure the holistic view of the system and lead developers to lose sight of the overarching architecture. Our work addresses this challenge by proposing a novel approach to track and assess the evolution of microservice architectures through static source code analysis. We combine source code repository mining techniques with architectural reconstruction to measure various metrics throughout a system's development history. Our approach uses a formal API-based decomposition model that can easily be adapted for different scenarios by choosing various architectural metrics. We validated our method's scalability and robustness through a case study on an extensive open-source microservice reference system with more than 40 individual services written in different languages and more than 400 commits. Our research provides software architects with a powerful tool to identify and monitor problematic architectural trends before they become imminent threats, enabling the evolution of microservice-based systems while maintaining architectural coherence and integrity.

Keywords: Microservice API · evolution · source code repository mining · metrics · source code detectors.

1 Introduction

The constant evolution of software systems is crucial for adapting to changing environments and meeting new requirements [1]. However, it also harbors the risk of architectural drift or erosion [2], leading to uncontrolled growth that may break previously defined conceptual boundaries. A software system that evolves without clear architectural guidance becomes more complex to maintain and more challenging to develop – its components can become too coupled, resulting in individual changes affecting too many parts of the system simultaneously [3]. Also, taking architectural shortcuts can often be necessary due to rough schedules and insufficient development resources [4]. Unfortunately, this approach may

produce technical debt that has to be addressed later to prevent disproportionate delays in feature development [2].

The problem of architectural erosion is as old as software engineering itself and persists despite various prevention strategies [5]. Microservice architectures have emerged as one such strategy, with isolation as a core principle, facilitating localized changes and easier replacement of problematic services [6, 7]. The idea is that a single service whose technical debt becomes worrisome can be easily replaced or rewritten [7]. These rewrites can also be valuable if business rules become better understood over time and allow for a leaner implementation [6].

However, while microservices mitigate technical debt within individual services, they exacerbate the challenge of maintaining architectural coherence across the system. Having isolated services means developers lose sight of the big picture, as each team works only on its small excerpt but misses a clear view of the overall architectural idea that underlies the entire system. The rapid development enabled by microservices can further accelerate architectural erosion, resulting in longer development times per feature with each new version [8].

To devise effective countermeasures against this erosion, identifying any problematic trends during development as early as possible would be necessary, but tracking the overall architectural integrity in a microservice system throughout its development is challenging: Many conventional metrics that work for monolithic applications, e.g., calculating coupling and cohesion, may not apply to highly distributed systems and loosely coupled architectures [9]. Also, existing approaches often focus only on single points in time, neglecting the continuous evolution of microservices [10–12]. Focusing only on an individual commit or snapshot bears the risk that erosion has already hit a threshold where implementing countermeasures may become too costly and bind too many resources that could be better spent on feature development [13].

Our work addresses this gap by proposing a novel approach to tracking and assessing the evolution of microservice architectures based on their APIs, which, as central points of their communication, play a significant role during their development. For this, we designed a multi-staged analysis process that, for the first time in this context, combines source code repository mining techniques with API-based architectural decomposition to reconstruct the evolution of a larger microservice architecture during the whole development process. We also derived a set of various metrics from our formal architectural model to analyze and assess the system's overall quality trends over time. We evaluated our solution in a case study on an extensive open-source microservice reference system with over 40 individual services and a history of 400 commits, verifying that our method works on a scale typical for mid-sized to large microservice architectures with a rich development history and polyglot nature. In contrast to research focusing on mining source code repositories, we do not use it as our central method to gather knowledge; instead, we consider it a tool to collect the information we need to reconstruct our architectural model. Accordingly, our study seeks to answer the following questions:

RQ1 *How can the evolution of microservice APIs be tracked efficiently?*
Creating an architectural model through source-code parsing is time-consuming, especially for larger microservice systems with a long commit history. A suitable approach must scale well, even for larger systems, and be robust enough to analyze and track large amounts of commits.

RQ2 *How can our approach be used to measure the different characteristics of microservice APIs over time?*
Microservices are highly dynamic systems tailored to specific domains, each with different requirements regarding the system's overall quality. Our analysis process must consider this variance by allowing software architects to adapt the analysis to their specific needs.

To our knowledge, this is the first study that uses source code repository mining to reconstruct the architectural evolution of large-scale microservice systems by using only static-analysis techniques. While we provide an initial set of metrics to measure architectural trends, we also demonstrate that our approach can easily be adapted to individual scenarios by using different metrics.

The remaining paper is organized as follows: Section 2 will look at the related research in this area, Section 3 will describe the approach we followed when implementing our work. The evaluation of our technique through a case study is the subject of Section 4. Section 5 presents and analyzes the findings we gathered through our case study. Potential validity threats affecting our results will be discussed in Section 6, while Section 7 concludes our work and contains an outlook on future work.

2 Related Work

The field of software architecture reconstruction [14] generally refers to works that reconstruct the software architecture or design from the source code, usually as a model or another intermediate representation. Our work aims to use repository mining techniques to reconstruct a microservice architecture during its evolution. Multiple works suggest approaches for microservice architecture reconstruction [15, 16], but so far, none considers the evolution history in the source code repository systematically.

High-level architectural metrics can streamline the evaluation of software system quality, abstracting away implementation details and reducing information overload. Numerous studies offer metrics-based analyses of microservice systems: Walker et al. [11] use static source code analysis to detect code smells in microservice architectures, Ma et al. [12] propose an automated process to generate service-dependency graphs, and Zdun et al. [10] define a set of constraints to detect potential architectural erosions. Another study focusing on reconstructing the architecture of microservice systems through static analysis is the work of Bushong et al. [17]. They extract HTTP information from source code to create a communication diagram for visualizing inter-service communication.

When extending the analysis over the whole software evolution process, Barnes et al. [18] present a strategy for tracking the evolutional steps that affect software architectures over time. While their approach is not specific to microservice systems, they introduce the interesting concept of evolution operators to categorize architectural changes. In addition, there are also studies more microservice-related: Moreira and De França [19] show in their research how the cohesiveness of microservices changes over time. Similar research is performed by Tizzei et al. [20]: They track several size-oriented metrics, like lines of the code and the number of service operations throughout development. While both works go in the same direction as ours, they focus mainly on single-service metrics. In contrast, we concentrate on inter-service communication. Sampaio et al. [21] developed a service evolution model based on configuration files and by tracing runtime service communication. He et al. [22] are focused on runtime analysis, as they develop an online prediction system based on different runtime metrics to precalculate the costs and effects of microservice evolution. In contrast, our approach relies only on collecting static code artifacts, making the integration into development pipelines easier as expensive system execution can be avoided.

Stocker and Zimmermann [23] conducted a survey to analyze the reasons for microservice evolution. According to their results, new functional requirements are the primary reasons for API changes, followed by improving architectural quality. The main obstacle they identify that prevents developers from architectural refactoring is a need for more resources. Lercher et al. [24] further identified the tight organizational coupling that leads to additional communication overhead as a challenge to microservice evolution and consumer lock-in that requires long-term support for legacy APIs.

3 Approach

3.1 Source Code Repository Mining

Reconstructing a software system's architecture from the source code requires parsing and analyzing numerous source code artifacts. This is challenging as modern systems are complex and often use various programming languages and technologies. It becomes even more difficult for microservice systems, where the architecture is distributed across loosely coupled services. Tracking the evolution of such architectures further adds another level of difficulty to this already complex problem, as hundreds of commits, all containing several changes, must be analyzed. While exhaustively analyzing the entire development history of large microservice systems is feasible with sufficient resources, more efficient strategies exist. Considering the two primary aspects of a code repository – (1) the number and size of source code artifacts and (2) the number of commits containing changes over time – we present two strategies to reduce each.

Our first mining strategy capitalizes on the fact that modern source code repositories store information as sequential lists of patches [25], each containing atomic changes applied to individual files. This strategy is termed *Change-Based Repository Mining*. In our approach, we suggest a novel approach using this

strategy for reconstructing an architectural representation during evolution: This method starts with an empty architectural model or a previously generated snapshot (see below) and iteratively transforms and applies each commit to evolve it until we reach the final architecture's state. Our method updates only the parts of the model affected by a single commit (inspired by the approach taken in [26]), making it highly efficient, under the assumption that each commit is atomic and contains only a few changes [27]. While efficient, this approach has a downside: By focusing solely on changes instead of source code artifacts, every commit must be investigated, as missing even a single one can lead to incomplete models, rendering subsequent commits unprocessable.

Our second mining strategy tackles these problems with a different approach: Instead of reducing the source code to be analyzed by focusing only on the changes, we reduce the number of commits to be examined and include all artifacts of our codebase. We call this approach *Snapshot-Based Repository Mining*, as it operates only on a self-contained snapshot of the whole repository at a given time. Although seeming less efficient due to the need to parse more code artifacts, this approach offers greater flexibility and fault tolerance, as operating on the entire code base rather than incremental changes ensures a self-contained model independent of previous iterations. Also, since every commit can be processed in isolation, the analysis can be parallelized more easily.

We combined both strategies into a single mining process for our final approach, allowing us to benefit from the advantages of both methods (Figure 1). In a first mining pass, we use the Snapshot-based approach on selected commits to create a macro-analysis of the overall system's architectural evolution. While this requires parsing more source code artifacts, it enables us to choose only specific commits we consider relevant. If we encounter any parsing errors, we skip the faulty commits and choose a nearby sample instead.

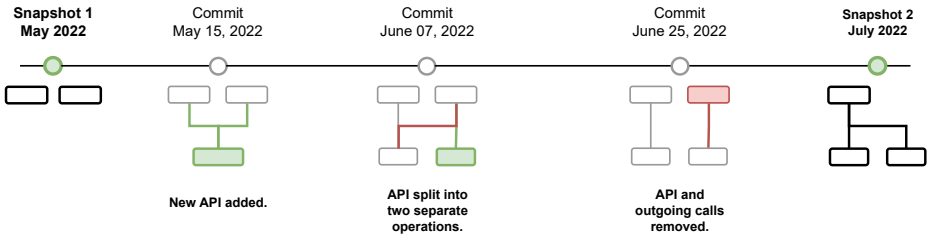

Fig. 1: The *Snapshot-based* mining approach creates the reference models for our start and endpoints. We then use the *Changed-based* mining approach to evolve our model iteratively between the snapshots to get a more in-depth view of the evolution.

After we pick enough commits to accurately recreate the architecture's development history, we then identify specific areas of interest between two commits and apply the change-based approach to get further a seamless visualization within the specific time range. With this approach, we can also detect any local minima or maxima we would otherwise not recognize.

Figure 2 shows a schematic view of our overall mining process. We begin by cloning the repository and creating local copies of all relevant commits, allowing further parallel execution. During the next phase, we use source code detectors, a lightweight parsing technology (see Section 3.2), to traverse each directory, identifying architectural patterns and capturing structural components and dependencies, resulting in our architectural decomposition model (also Section 3.2). Another set of detectors connects the identified services through API invocation call chains, forming a directed graph representing the system's static communication model. Using this structure, we calculate and store architectural metrics (see Section 3.3) to assess the system's health for each snapshot. After processing all selected commits, the resulting metrics are merged into a continuous trend diagram, showcasing the system's evolution over time. If desired, the change-based method can be run as a second pass between specific commits to fill any potential gaps in the analysis.

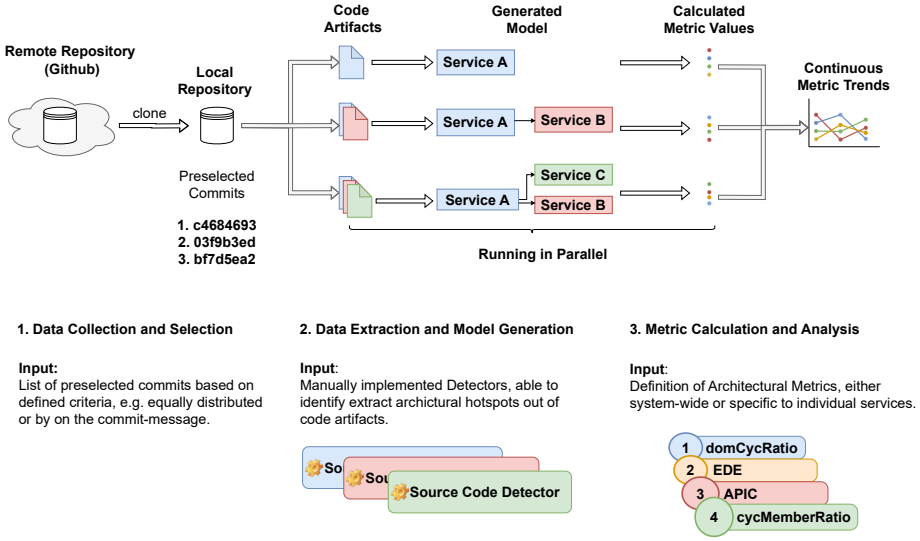

Fig. 2: The whole analysis process consists of three steps, where the second and parts of the third step can run in parallel to reduce the overall processing time.

3.2 Model Generation

Identifying architectural problems on the source code level can be challenging due to the lower level of abstraction, and even in well-documented systems, the documentation is not guaranteed to be precise and reflects the current architectural state [28]. To ensure our process accurately represents the overall system, we use the underlying code base as our single source of truth to reconstruct an abstract architectural model of each service and create a communication model showing the interconnection between these services. The model we decompose from the implementation focuses on API operations as the primary communication point within a microservice system. These include synchronous APIs and

asynchronous message handlers, which we consider both part of a service's public interface. Besides API operations and their invocations, our model also contains optional elements like API Interfaces and Connection Hosts that help to create better traceability between model elements and code.

We use *Source Code Detectors*, lightweight Python-based parsers [29], to extract and generate our architectural model. These detectors look only for specific patterns, like a REST API method definition or asynchronous pipe subscriptions, and ignore all unrelated details, making their implementation more efficient than techniques that rely on full-fledged abstract-syntax-tree reconstruction. When identifying such a pattern, the detector can create a new architectural model element or enhance an existing one to reflect the newly gathered knowledge. Once the detectors traverse all artifacts, the resulting elements form an unlinked model of all architectural hotspots we consider relevant for our analysis [30]. In a second run, a different set of detectors identifies bridges between elements, such as local invocations or remote API calls, enhancing the model's coherence. Figure 3 illustrates this workflow.

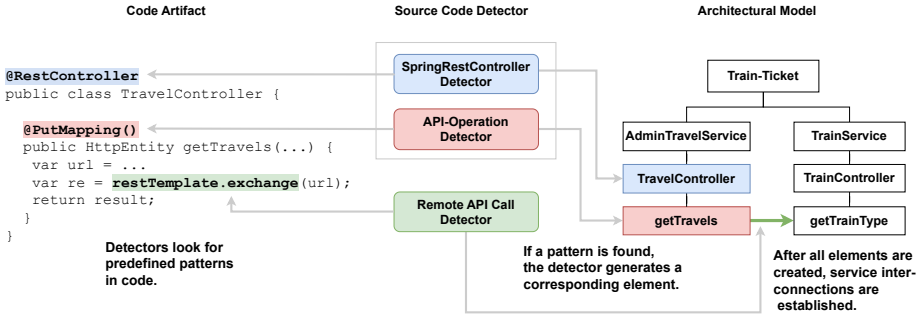

Fig. 3: Source Code Detectors are lightweight parsers that search the underlying code base for specific concepts. Whenever such a concept is found, a corresponding model element is generated.

Since our detectors are hand-crafted for every system we analyze, we can use heuristics based on project-specific coding conventions and style guides to simplify their implementation. However, our detection logic must remain robust enough to handle anomalies like typos or deviations, which may occur occasionally, especially when tracking a system's implementation over a longer time than we do in our research.

3.3 Architectural Metrics

To use our model to identify and track potential architectural trends, we give it a formal description that we then use to derive several architectural metrics. For this, we interpret our generated model as a directed graph $G = (V, E, F)$ [30]. Here, V denotes all architectural elements identified by our detectors, further categorized by their role within the system, e.g., V^{ms} for the set of all microservices and V^{api} for all API operations. E represents edges forming relations

between elements, describing either a *has a* relation (e.g., between a microservice and its APIs) or an API or method invocation. F specifies predicates and utility functions we use for retrieving meta-data attached to the nodes. These can be annotations such as synchronous/asynchronous protocol usage or secure communication channels. For our case study, we collected a set of existing metrics from the literature that operate on higher abstraction levels and are thus implementation-independent. By purpose, we also chose metrics that measure different architectural and qualitative aspects to show that our approach is not restricted to any specific scenario. Table 1 shows the metrics we used for our case study and how we derived them from our formal model.

Name	Formula	Description
Average Service Interface Count (ASIC)	$\frac{\lvert V^{api}\rvert}{\lvert V^{ms}\rvert}$	Variant of *Weighted Service Interface Count* metric [31], with a uniform weight of 1. Quantifies the number of distinct API operations per microservice.
Average Path Length (APL)	$\frac{\sum\{p \in P(v) : \lvert p\rvert\}}{\lvert P(v)\rvert}$	Quantifies the consecutive API operations invoked when accessing any public API, with $P(v)$ being the set of all paths starting in v [30].
Cycle Ratio (cycRatio)	$\frac{\lvert\{p \in P(v) : cycle(p)\}\rvert}{\lvert P(v)\rvert}$	Tracks any cyclic paths when following the call chain of an API operation v. The predicate $cycle \rightarrow [0, 1]$ identifies paths containing at least one cycle [30].
Service Interaction via Intermediary Component (SIC)	$\frac{\lvert\{v \in V^{con} : is\_async(v)\}\rvert}{\lvert V^{con}\rvert}$	Ratio of service connections mediated through components like event buses or message brokers [32]. As this communication happens primarily asynchronously, we use the $is\_async \rightarrow [0, 1]$ predicate to identify such connectors.
Connection Anomaly Ratio (CAR)	$\frac{\lvert\{v \in V^{con} : is\_error(v)\}\rvert}{\lvert V^{con}\rvert}$	Ratio of service connections where the link to the target API could not be reconstructed. We also use this metric to verify that our detectors identified interservice calls correctly.

Table 1: A set of high-level architectural metrics we employ to obtain a comprehensive perspective of the overall structure of the microservice.

While the expressiveness of each metric may be limited when viewed in isolation, combining them can help provide a holistic view of the architectural trend of the system under observation, as we will show in our case study.

4 Case Study

We evaluated our approach on the *Train Ticket* repository[3], an open-source microservice benchmark system. It has more than 30 microservices, making it

[3] `https://github.com/FudanSELab/train-ticket`

considerably larger than most other open-source implementations [33]. It was already part of various research studies [34, 35], and according to the designers of the systems, they intended to provide a system with more public service APIs and deeper invocation chains than other open-source projects to better align with today's industry standards [33]. These architectural decisions and its extensive development history make it an ideal candidate for our analysis.

For our study, we focused on the backend architecture of the system and the API-based service interaction. i.e., we excluded the frontend-related *ts-ui-dashboard* microservice and the API Gateway as we do not consider these as parts of the backend. We also skipped three other services that were only connected to the UI or did not make any other inter-service calls: *ts-ticket-office*, *ts-avatar* (only called from UI without any other service interaction) and *ts-news-service* (does not contain any production code yet). This left us with 44 backend domain and infrastructure services, mostly implemented in Java, with the *voucher service* implemented in Python. This number should be large enough to represent a typical mid-size microservice system adequately.

Comparing the different branches and their commit frequency, we decided to run our analysis on the *Reconstruction* branch. From an evolutionary perspective, this branch provided the best source for reconstructing the system's evolution, as it contained the most commits and was frequently merged into the main branch. We selected every fourth entry from 440 commits in this branch, resulting in around 110 samples for our analysis process. Although our selection is not equally distributed throughout the repository's timeline, it should still cover enough of the development process to let us identify any possible trends (see Figure 4).

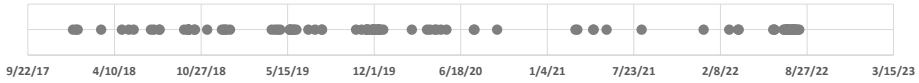

Fig. 4: Distribution of selected commits throughout the repository's timeline.

As most service implementations use the Java-based Spring Framework, we only had to implement 13 different detectors to track all relevant features from the Java services and seven detectors for the Python-based service. The size of our detectors varies, with some containing only a few lines required to detect specific keywords and others having up to 50 lines of code needed to extract class methods and their annotations. Overall, the implementation effort for a system of such a size is manageable, especially since most detectors can be reused throughout different services and commits.

4.1 Connection Anomaly Ratio

We first focused on the *Connection Anomaly Ratio* metric of every sample to verify our approach's correctness. All spikes we encountered in the trendline were cross-checked with our detector implementation to determine whether the reason for each missed connection could be a flawed detector. We continued

this process iteratively until our detectors could cover most edge cases, and the remaining detector-related connection faults accounted for less than 1%, which we consider a value good enough. Aiming to reach an even lower fault rate would be challenging, as there were cases where target API addresses were constructed through various conditions depending on the input variables. These scenarios are hard to solve solely through pattern detection and require more complex parsing techniques, like reconstructing abstract syntax trees.

We identified only a short time range between 2018 and 2019, where the CAR metric peaked at 0.15, meaning that roughly 15% of interservice connections could not be resolved (see Figure 5). Further investigation showed that it was due to a wrongly placed section in the configuration file of the *ts-config-service*. Whether this would manifest in real connection problems at runtime depends on how tolerant the system's configuration loader is against this type of error. However, we would still clearly flag this as an anomaly as it diverges significantly from all other services' configurations.

Despite that, we found only minor errors, e.g., isolated API calls using wrong target addresses. Overall, the system's interconnectivity implementation is very correct and mostly free of errors or issues.

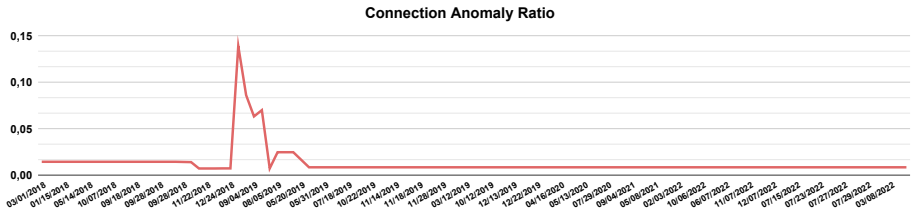

Fig. 5: *Connection Anomaly Ratio* throughout the *Reconstruction* branch. Despite a service misconfiguration at the beginning of 2019, which was later corrected, service interconnection remains highly error-free throughout the development history.

4.2 Average Service Interface Count and Path Length

The *ASIC* and *APL* metrics can both provide meaningful insights when assessing a system's architecture. A combination of high *ASIC* and *APL* values could indicate many small and atomic API operations that must be composed to implement more complex use cases, which in turn can result in longer invocation chains. Looking at the *train-ticket* system, we can see a similar pattern beginning in 2018 (Figure 6): The number of available operations per service (blue line) increases slightly during development, which could indicate that the service interfaces have become more use-case-specific and less generic. However, verifying this assumption would require a deeper analysis of the APIs' semantics, which was out of the scope of our study.

Regarding the average number of API invocations (red line), we also see an increasing metric value till July 2022, i.e., calling a single API results in a growing number of follow-up calls. This trend is not unproblematic, as longer call paths create stronger interservice coupling, especially when these calls are synchronous and directed, as is the case here. Nevertheless, it seems the system's

authors were also aware of this trend and restructured the architecture towards reduced invocation chains as the value dropped significantly in July 2022.

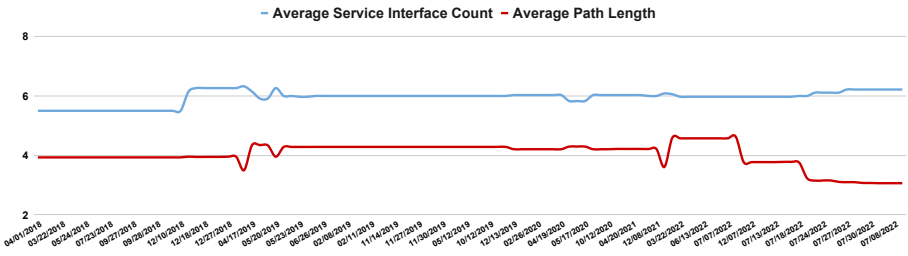

Fig. 6: While increasing *Average Service Interface Count* (blue) and *Average Path Length* (red) metrics indicate a stronger service coupling, several system redesigns were applied to constantly reduce the path length starting in the middle of 2022.

4.3 Cycle Ratio and Asynchronous Service Communication

As Figure 7 shows, there were times when between seven and up to 15% of API call chains in the system had at least one *weak* cycle, meaning the same service was addressed more than once during a call sequence [12]. These cyclic relations increase service coupling and can lead to higher network traffic, as calls must travel to the same service several times.

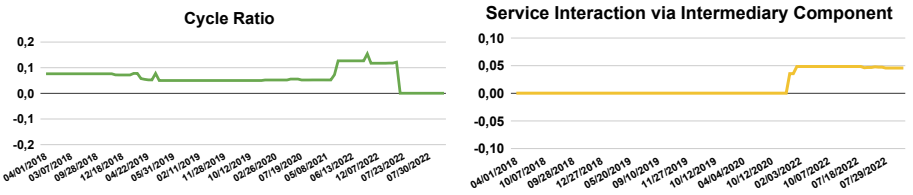

Fig. 7: *Ratio of cyclic connections* (left) and *Service Interaction via Intermediary Component Ratio* and other (right). While the amount of cycles decreases over time, the asynchronous communication increases. Both measures can help reduce the service coupling.

However, the system developers were probably aware of this problem and introduced a redesign as both metrics changed significantly by the end of the development history. Comparing two snapshots of the same API call from April and July 2022 (Figure 8), the earlier one indicates a cyclic relation between the *Travel* and the *Seat Service*, creating a strong dependency and also additional overhead as the *TravelService* has to be visited twice. Moving further in the commit history, the system designers resolved this cycle: The second call to the *Travel Service* has been moved up in the hierarchy, making the actual caller – the *Route Plan Service* – responsible for orchestrating the API calls. Albeit this reduces the coupling, the problem of additional communication overhead remains. Here, the architecture could, for instance, be further improved by introducing a new API in the *TravelService* that bundles both requests.

Besides removing cycles, in 2021, the system's authors added asynchronous communication channels, mainly used for notification mechanisms. Compared to earlier versions of the system, this is another step towards a less coupled and

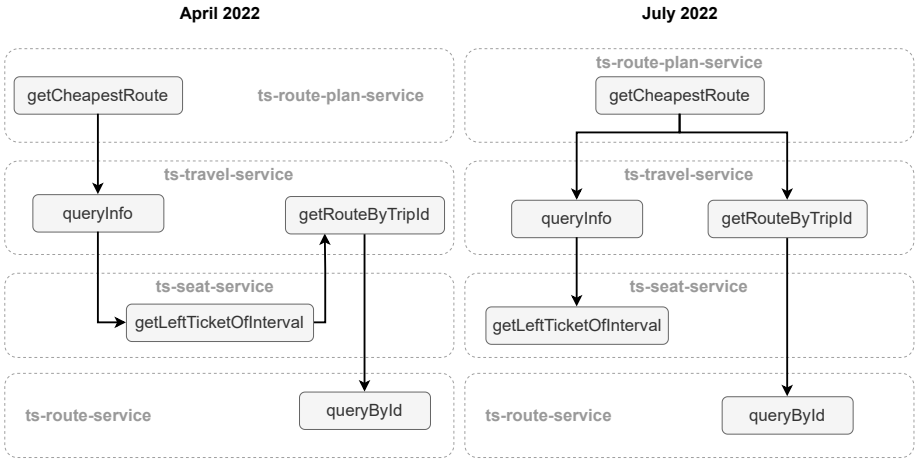

Fig. 8: Redesign of an API call chain. While the version from April 2022 contains a cyclic reference between two services, this was later resolved by letting the caller orchestrate the API calls instead.

more scalable solution, as it allows the services to be changed independently as long as they conform to the same message protocol and payload. Whether this approach could also be used to decouple additional parts of the business logic may depend on the individual scenario.

4.4 Summary

Overall, our static commit analysis revealed that the system is very stable. While there were some misconfigurations in the past, these were all corrected within a relatively short time frame. Following the trend of our metrics throughout *train ticket's* development history, we further see evidence suggesting the maintainers were constantly improving their architecture towards less coupling and better maintenance. One of the most vital indicators is the reduced length of API invocation chains and removing all cyclic paths in the system. Asynchronous communication patterns, added during the last third of the commit, further improve this trend towards decoupling.

In contrast, the number of API operations per service increases, which gives room for interpretation: Either existing functionality was split up into more fine-grained API operations, allowing for better API composition, or over time, more features were added to the services, increasing the overall number of available APIs. The simultaneous reduction of the average invocation path length suggests the latter, but a more semantic-based API analysis would be necessary here to verify this assumption.

5 Discussion

Considering **RQ1**, our case study demonstrated that our approach makes reconstructing a microservice's architectural evolution manageable, even for larger

systems with several highly interconnected services. We developed a mining process consisting of two different strategies, one based on the analysis of individual snapshots while the other one focusing on applying the changes of every commit iteratively. Combining both methods makes our approach robust against errors while allowing for detailed and seamless in-depth analysis for specific points of interest. For this, we first analyzed a selection of snapshots to get an overall impression of the system's architectural trend while using change-based mining later to fill the gaps between relevant commits. We could also show that implementing the detectors to parse the code artifacts and decompose the architectural model is manageable when the system's code base follows coding guidelines and best patterns or practices. In that case, most detectors can be reused throughout different services, reducing the overall implementation effort.

Still, we also recognized that our implementation leaves room for improvement, e.g., using more optimized pattern matching.

When assessing and analyzing a microservice system's evolution (**RQ2**), we could also verify that the formal graph-based model generated by our detectors provides a solid ground for deriving various architectural metrics. By combining different metrics, we tracked various architectural qualities of our benchmarking system throughout its development history and identified significant architectural trends. In the case of our reference system, we could verify that some of our measured quality indicators, like cyclic dependencies or asynchronous communication, improved over time. We consider both valuable strategies to reduce the overall coupling within the system.

6 Threats to Validity

Construct Validity: Our detectors create an architectural model from source code, and with all models, there is always the risk of missing crucial details. We manually reviewed and cross-checked our model with the underlying implementation and documentation[4] to ensure all service interactions were tracked. We also added error checks during the reconstruction process to catch anomalies like missing target operations. However, implementing automated verification processes, such as additional runtime tests, would improve our work, a path we are exploring for future research.

External Validity discusses whether our results are generalizable to other systems or on a larger scale. While we doubt that there is such a thing as a general microservice architecture – every system is tailored to a specific domain and has to deal with unique requirements – the example application we chose for our case study is a widely accepted benchmark system with a relatively large scale compared to most other open-source implementations[5], making it well comparable to real-world systems. Besides, our approach could easily be applied to larger systems, considering the additional effort required to implement the detectors.

[4] https://github.com/FudanSELab/train-ticket/wiki/Service-Guide-and-API-Reference

[5] see, for instance, this curated list of various microservice open source systems: https://github.com/davidetaibi/Microservices_Project_List

Internal Validity: Our whole approach to reconstructing the architecture relies on the source code and configuration files that evolve. While these artifacts are undoubtedly the most essential source of knowledge when investigating such systems, other aspects, like non-functional requirements or specific domain knowledge, may also affect the overall design of the architecture.

7 Conclusions and Future Directions

In this paper, we propose a method for monitoring microservice architecture evolution by reconstructing an API-based communication model through static source code repository analysis. Our mining process creates an architectural trend and allows seamless evolution analysis between selected commits. We employ lightweight, project-specific source code detectors to extract relevant architectural hotspots. The initial effort to implement these detectors is manageable even for large systems as long as recurring coding patterns and guidelines are applied. Our generated graph-based architectural model enables the derivation of various metrics to assess and identify quality trends. We validate our method with a case study on an extensive benchmark system.

To our knowledge, this is the first study to demonstrate such an approach across the entire development history of a complex microservice system, providing software architects with a tool to monitor evolution and detect negative quality trends early.

8 Data Availability

We offer the whole source code and data of our study in a data set published on Zenodo: `https://doi.org/10.5281/zenodo.10961768`.

Acknowledgments: This work was supported by: FWF (Austrian Science Fund) projects API-ACE: I 4268.

Bibliography

[1] M. W. Godfrey and D. M. German, "The past, present, and future of software evolution," in *2008 Frontiers of Software Maintenance*. IEEE, 2008, pp. 129–138.

[2] E. Whiting and S. Andrews, "Drift and erosion in software architecture: summary and prevention strategies," in *Proceedings of the 2020 the 4th International Conference on Information System and Data Mining*, 2020, pp. 132–138.

[3] D. Baum, J. Dietrich, C. Anslow, and R. Müller, "Visualizing design erosion: how big balls of mud are made," in *2018 IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 2018, pp. 122–126.

[4] N. Rios, R. O. Spínola, M. Mendonça, and C. Seaman, "The most common causes and effects of technical debt: first results from a global family of industrial surveys," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2018, pp. 1–10.

[5] L. De Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *Journal of Systems and Software*, vol. 85, no. 1, pp. 132–151, 2012.

[6] 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.

[7] S. Newman, *Building microservices.* " O'Reilly Media, Inc.", 2021.

[8] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, "Limiting technical debt with maintainability assurance: an industry survey on used techniques and differences with service-and microservice-based systems," in *Proceedings of the 2018 International Conference on Technical Debt*, 2018, pp. 125–133.

[9] J. Bogner, S. Wagner, and A. Zimmermann, "Automatically measuring the maintainability of service-and microservice-based systems: a literature review," in *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, 2017, pp. 107–115.

[10] U. Zdun, E. Navarro, and F. Leymann, "Ensuring and assessing architecture conformance to microservice decomposition patterns," in *Service-Oriented Computing: 15th International Conference, ICSOC 2017, Malaga, Spain, November 13–16, 2017, Proceedings.* Springer, 2017, pp. 411–429.

[11] 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.

[12] S.-P. Ma, C.-Y. Fan, Y. Chuang, I.-H. Liu, and C.-W. Lan, "Graph-based and scenario-driven microservice analysis, retrieval, and testing," *Future Generation Computer Systems*, vol. 100, pp. 724–735, 2019.

[13] S. S. de Toledo, A. Martini, A. Przybyszewska, and D. I. Sjøberg, "Architectural technical debt in microservices: a case study in a large company," in *2019 IEEE/ACM International Conference on Technical Debt.* IEEE, 2019, pp. 78–87.

[14] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573–591, 2009.

[15] T. Cerny, A. S. Abdelfattah, V. Bushong, A. Al Maruf, and D. Taibi, "Microservice architecture reconstruction and visualization techniques: A review," in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE).* IEEE, 2022, pp. 39–48.

[16] V. Bushong, D. Das, A. Al Maruf, and T. Cerny, "Using static analysis to address microservice architecture reconstruction," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 2021, pp. 1199–1201.

[17] V. Bushong, D. Das, and T. Cernỳ, "Reconstructing the holistic architecture of microservice systems using static analysis." in *CLOSER*, 2022, pp. 149–157.

[18] J. M. Barnes, D. Garlan, and B. Schmerl, "Evolution styles: foundations and models for software architecture evolution," *Software & Systems Modeling*, vol. 13, pp. 649–678, 2014.

[19] M. G. Moreira and B. B. N. De França, "Analysis of microservice evolution using cohesion metrics," in *Proceedings of the 16th Brazilian Symposium on Software Components, Architectures, and Reuse*, 2022, pp. 40–49.

[20] 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 *2020 IEEE International Conference on Web Services (ICWS).* IEEE, 2020, pp. 102–109.

[21] A. R. Sampaio, H. Kadiyala, B. Hu, J. Steinbacher, T. Erwin, N. Rosa, I. Beschastnikh, and J. Rubin, "Supporting microservice evolution," in *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 539–543.

[22] X. He, Z. Shao, T. Wang, H. Shi, Y. Chen, and Z. Wang, "Predicting effect and cost of microservice system evolution using graph neural network," in *International Conference on Service-Oriented Computing*. Springer, 2023, pp. 103–118.

[23] M. Stocker and O. Zimmermann, "From code refactoring to api refactoring: Agile service design and evolution," in *Symposium and Summer School on Service-Oriented Computing*. Springer, 2021, pp. 174–193.

[24] A. Lercher, J. Glock, C. Macho, and M. Pinzger, "Microservice api evolution in practice: A study on strategies and challenges," *arXiv:2311.08175*, 2023.

[25] W. K. Assunção, J. Krüger, S. Mosser, and S. Selaoui, "How do microservices evolve? an empirical analysis of changes in open-source microservice repositories," *Journal of Systems and Software*, p. 111788, 2023.

[26] F. Heseding, W. Scheibel, and J. Döllner, "Tooling for time-and space-efficient git repository mining," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 413–417.

[27] C. Kolassa, D. Riehle, and M. A. Salim, "A model of the commit size distribution of open source," in *International Conference on Current Trends in Theory and Practice of Computer Science*. Springer, 2013, pp. 52–66.

[28] J. Rosik, A. Le Gear, J. Buckley, M. A. Babar, and D. Connolly, "Assessing architectural drift in commercial software development: a case study," *Software: Practice and Experience*, vol. 41, no. 1, pp. 63–86, 2011.

[29] E. Ntentos, U. Zdun, K. Plakidas, P. Genfer, S. Geiger, S. Meixner, and W. Hasselbring, "Detector-based component model abstraction for microservice-based systems," *Computing*, vol. 103, no. 11, pp. 2521–2551, 2021.

[30] P. Genfer and U. Zdun, "Identifying domain-based cyclic dependencies in microservice apis using source code detectors," in *Software Architecture: 15th European Conference, ECSA 2021, Virtual Event, Sweden, September 13-17, 2021, Proceedings*. Springer, 2021, pp. 207–222.

[31] M. Hirzalla, J. Cleland-Huang, and A. Arsanjani, "A metrics suite for evaluating flexibility and complexity in service oriented architectures," in *Service-Oriented Computing–ICSOC 2008 Workshops: ICSOC 2008 International Workshops, Sydney, Australia, December 1st, 2008*. Springer, 2009, pp. 41–52.

[32] E. Ntentos, U. Zdun, K. Plakidas, S. Meixner, and S. Geiger, "Assessing architecture conformance to coupling-related patterns and practices in microservices," in *Software Architecture: 14th European Conference, ECSA 2020, L'Aquila, Italy, September 14–18, 2020, Proceedings 14*. Springer, 2020, pp. 3–20.

[33] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking microservice systems for software engineering research," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 323–324. [Online]. Available: https://doi.org/10.1145/3183440.3194991

[34] B. Li, X. Peng, Q. Xiang, H. Wang, T. Xie, J. Sun, and X. Liu, "Enjoy your observability: an industrial survey of microservice tracing and analysis," *Empirical Software Engineering*, vol. 27, pp. 1–28, 2022.

[35] C. Zhang, X. Peng, C. Sha, K. Zhang, Z. Fu, X. Wu, Q. Lin, and D. Zhang, "Deeptralog: Trace-log combined microservice anomaly detection through graph-based deep learning," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 623–634.