Evolution and Anti-patterns Visualized: MicroProspect in Microservice Architecture

Lauren Adams[1], Amr S. Abdelfattah[1], Md Showkat Hossain Chy[2], Samantha Perry[2], Patrick Harris[1], ✉ Tomas Cerny[2], Dario Amoroso d'Aragona[3], Davide Taibi[3,4]

[1] Baylor University, Waco, Texas, USA
[2] SIE, University of Arizona, Tucson, Arizona, USA
[3] Tampere University, Tampere, FI-33720, Finland
[4] University of Oulu, Oulu, FI-90520, Finland
tcerny@arizona.edu

Abstract. A microservice architecture has become the dominant direction for designing the building blocks of large-scale, distributed software systems. However, the dynamic and changing microservices within decentralized systems in contrast to available static tracing tools presents challenges for comprehending its impact on the overall architecture. Existing tracing tools uncover service call graphs but have limitations in visualizing historical changes; moreover, they are not meant to aid with architecture assessment where developers seek potential design anomalies. With the ever-growing system complexity, developers likely resort to focusing on specific subsets of the system, especially given the lack of tools to analyze the impacts of system evolution. To address these challenges, we introduce the MicroProspect tool that provides a high-level, holistic visual perspective on the system's service view, tracks its structural changes throughout system evolution, and detects and visualizes anti-patterns that could lead to architectural degradation.

Keywords: Microservices · Evolution · Degradation · Visualization

1 Introduction

Microservices are a mainstream approach to building large and scalable systems [37]. Microservice architecture offers increased flexibility and autonomy in system evolution involving independent development teams following Conway's law [17]. However, development teams typically deal with the evolution of individual microservices without paying attention to the greater system perspective as the effect of decentralization. Without instruments to advise developers about continuous changes and the system's evolution as a whole, the system becomes susceptible to architectural degradation.[1] [2]

`Architectural degradation` is defined as the process of the persistent inconsistency between the descriptive software architecture as implemented and the prescriptive software architecture as intended [5, 16]. The example

[1] MicroProspect Source Code: `https://github.com/cloudhubs/mvp`
[2] MicroProspect Demo: `https://youtu.be/HXSB4uAxRH4`

of architecture degradation in open-source software is evident as over time, software architecture erodes, deviating from its intended conceptual structure due to factors like requirement changes and new features. This divergence, termed architecture erosion or drift, introduces inconsistencies between the implemented architecture and the originally planned architecture, negatively impacting software quality and potentially necessitating a redesign of the system [5]. It is typically an outcome of the gradual injection of code anomalies (i.e., anti-patterns, poor design choices, lack of maintenance, accumulated technical debt [7, 27], etc.) as the software evolves. Degradation occurs when the critical quality attributes are violated [30].

Common detection strategies to identify degradation [5] base on design quality *metrics* (e.g., instability, cohesion, coupling) [31, 8], or on the prioritization of *architecture anomalies* referred to as smells or anti-patterns [19, 33]. `Anti-patterns` [36] are recurring design *practices, choices, or solutions to common problems* despite appearing reasonable and effective, leading to negative consequences and undermining the system's overall quality. `Bad smells` [35] describes a design characteristic that indicates a potential problem or violation of good practices, a warning sign that suggests potential issues in the design. These both prompt further analysis and consideration to identify the underlying problems and propose appropriate design improvements. To refer to both anomalies, we use the term anti-pattern.

With microservices, the system complexity easily grows to the point where system evolution becomes hard to trace, requiring new methods and tools to support history tracking for the system architecture [20]. Detecting and rectifying microservice anti-patterns apparent from the system's holistic perspective throughout the development and evolution process is crucial to avoid undesirable outcomes [4, 13, 29]. Even though distributed tracing tools (i.e., Jaeger [1], Dapper [34]) derive service call graphs, there is no visualization support for evolution, making it challenging to observe system changes. As many microservice anti-patterns are elusive within individual microservices, traditional tools like SonarQube [10] fail. The holistic system perspective (i.e., service call graph) is necessary for such analysis; however, even then, complex graphs in large systems pose difficulties in identifying issues due to information overload and distractions from numerous connections and nodes.

To address these limitations, we introduce a MicroProspect tool to provide developers with a comprehensive and interactive visualization fueled with the capability to compare system versions, highlight anti-patterns, and provide detailed architectural insights to identify design issues. By analyzing the system's historical data (i.e., from repositories), developers can visually identify degradation trends over time, leading to informed decisions leading to wise design choices.

This paper is organized as follows: Section 2 details the software architecture reconstruction process. Section 3 introduces our tool and Section 4 gives details on how to practcially use the tool. The evaluation of the tool on a microservice

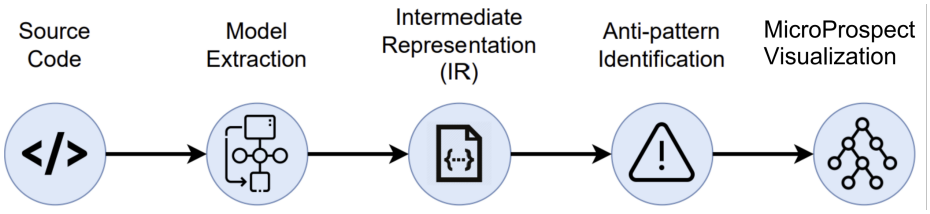

Fig. 1. Generalized SAR process for service call graph extraction and visualization used in MicroProspect processing

system is described in Section 5 and related work is briefed in Section 6. Finally, Section 7 and 8 concludes the paper.

2 Software Architecture Reconstruction (SAR)

To accurately represent the system's architecture, the process of Software Architecture Reconstruction (SAR) aims at extracting the architecture from an existing software system [9]. It involves reverse-engineering the system's structure, components, and interactions based on its implementation artifacts, such as source code and configuration files. Automating the process requires constructing a system Intermediate Representation (IR) that captures the system structure and component dependencies [9]. Such a representation can be extracted from the system by various means (i.e., static or dynamic analysis).

Our target result is a visualized system service view perspective [11] decorated with information pertaining to the occurrence of anti-patterns and the system evolution. The service view represented by a service call graph is the mostly adopted visual approach for microservices [23]. In such a graph, services are represented as nodes, and requests between services as links. The overall SAR process phases are detailed in Fig. 1 and described in the following subsections.

Model Extraction: The first phase involves static analysis of the microservices codebase to extract architectural components to construct the model representation of the system. The two-phase process involves the analysis of individual microservice codebases and then their interconnections.

The first phase analyzes the individual microservices, it can be assumed that component-based development frameworks are used to develop microservices. Resorting to low-level language use would lead to wheel reinvention and bring significant disadvantages to system evolution [32]. Therefore, our methodology assumes that the framework follows enterprise communication standards, organizing components into Controller, Service, and Data Repository within projects [15]. We scrutinize microservices' codebases, extracting source files and parsing them to pinpoint method declarations and bodies. This extracted content encompasses an individual microservices component call graph, depicted in Fig. 2. It illustrates that endpoint calls are received from the Controller component and then delegated to the implemented Service component. The Service component is tasked with communicating with the data sources of the Data Repository component and other microservices, initiating remote calls to

fulfill the required tasks. Schiewe et al. [32] demonstrated the identification of high-level constructs and components from abstract syntax trees. This approach reveals controllers and their endpoints with specific REST properties, along with identifying remote REST calls within the code.

The second phase interconnects the individual microservices with each other. The extracted remote calls from component call graphs reveal the connections between the microservices, forming the foundation for constructing service dependencies. The process combines individual component call graphs using merging ingredients like call signature match to endpoints or data simulates [3]. For extracting system interconnections, we consider the Prophet static analysis tool [9]. For explicit dependencies, Prophet uses approximation via signature matching between the REST endpoint and remote REST calls to identify connections. This approach has shown to be reliable through repeated experimentation, yet, it must be understood that static analysis provides only an approximation. However, the requisite cost is low, given no system execution is necessary, as in the case of dynamic system analysis.

The result then follows in the format of endpoint and rest call interconnections which, with the trace to their original microservices, leads to derived dependencies across microservices. A similar approach is possible for asynchronous calls (i.e., messaging) but was not considered in this work.

Intermediate Representation: Following the model extraction phase, the extracted component call graphs evolve into the IR, which represents the system components and their interconnections. From this IR, we can derive information about services, types, and dependencies which can be used to demonstrate the interconnections between services to construct a service view. Its format describes a composite structure listing the component call from endpoints within all its microservices and links between services throughout the particular endpoint route, as shown in Fig. 3. Such information makes it possible to render a service dependency graph of the system at a high level as a directed network.

Anti-pattern Identification: From the IR, we can seek to identify various anti-patterns within the microservice system that are traceable from the service dependency perspective. We can traverse the intermediate representation and label nodes with information pertaining to anti-patterns they may be a part of. We demonstrate the detection of selected anti-patterns based on information derived from the structure of the service call graph. In particular, we targeted the following:

- **High outgoing coupling**(variable threshold) - Service (outgoing connections) is interconnected or dependent on too many other services.
- **Cyclic dependency** - A cyclic chain of calls between two or more services that depend on each other directly or indirectly. Various cyclic dependency shapes can be recognized. Involved services can be hard to release and maintain. Likely, responsibilities are not separated correctly across services. This leads to problems with deployment, scalability, and co-change coupling. [13]

– **Bottleneck service** (variable threshold) - A service that is highly used (incoming connections) by other services. Its response time can be high because too many services use it. Service availability may go down due to the traffic.
– **Megaservice** (variable threshold) - Services should be small, independent, independently deployable units and serve a single purpose. A mega service has a high number of endpoints and a high fan-in. It is a result of poor system decomposition when a service combines multiple functionalities that multiple services should handle. [13] Creates maintenance issues, reduced performance, and difficult testing.

The decorated version of the graph JSON can then be used by the MicroProspect tool to display information pertaining to anti-patterns within and across system versions, highlighting changes.

3 MicroProspect Tool

We sought to develop a comprehensive visualization approach to enable developers to view the system service dependency graph regardless of the system

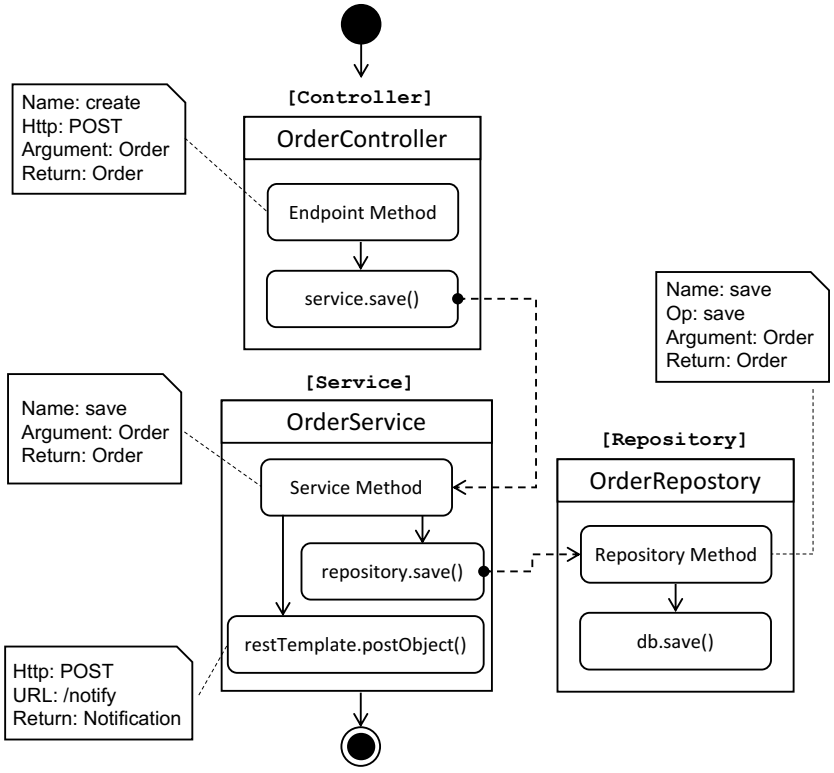

Fig. 2. Component Call Graph example

```
"source": "ts-preserve-other-service",
"target": "ts-security-service",
"requests": [
    {
        "type": "GET",
        "argument": "@PathVariable String accountId, @RequestHeader HttpHe
        "msReturn": "org.springframework.http.HttpEntity",
        "endpointFunction": "security.controller.SecurityController.check"
    }
```


Fig. 3. Example intermediate representation of service dependency graph from the train ticket system benchmark

scale. This necessitated an interactive tool that allows for various features suggested by Abdelfattah et al. [2], such as search, tracking, and isolation of particular services and their neighbors to successfully divide large microservice systems into manageable components. Additionally, Abdelfattah et al. found that a 3D visualization enabled novices to perform on the same level as experts in identifying relationships between services and outperform those using a 2D tool [2]. As a result of this and the service dependency graph's focus on relationships among services, we targeted a 3D visualization. Moreover, to understand system degradation, we desired to incorporate a fourth dimension in comparing system versions and anti-patterns over time. To further understand the system degradation, we can also display how anti-patterns change between system versions by comparing the individual occurrences between the two versions. Tracking system degradation can be accomplished by repeating the aforementioned SAR process for several system iterations and ordering them on a timeline that can be paged through.

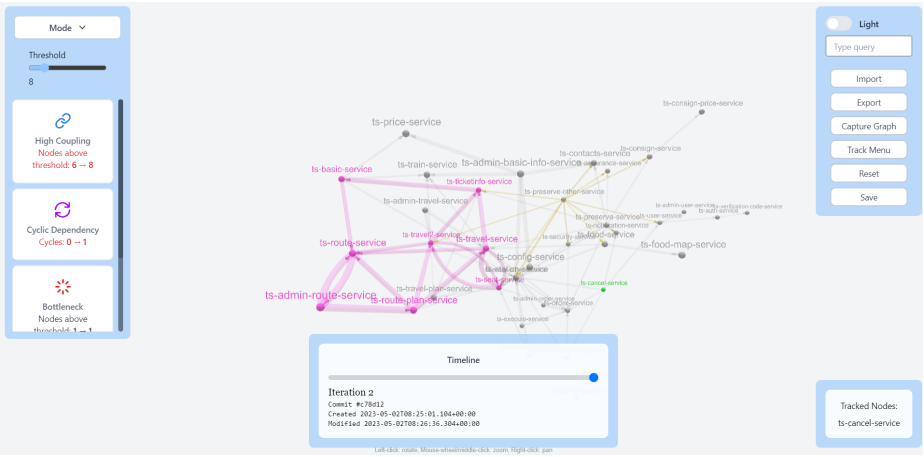

Fig. 4. Capture of the Train Ticket system in the MicroProspect tool

Interactivity: MicroProspect offers interactive navigation of a visual service dependency graph in 3D space. The graph can be rotated, panned, zoomed, and rearranged via dragging nodes. Services are visualized as nodes in the graph and can be focused on via click to obtain specifics and to isolate the service and its neighbors visually. Requests between services are visualized as arrows of width dependent on request quantity and have the flow direction animated on hover. The tool offers many visualization options that can be toggled from menus on the left side of the screen, in addition to search functionality to isolate services by name for easier navigation of complex graphs.

All of these features serve to enable navigability despite graph complexity, which necessitated this 3D interactive approach. Fig. 4 illustrates the Train Ticket benchmark in MicroProspect [3]

Anti-pattern Visualization: Previously mentioned anti-patterns are visualized by MicroProspect, including High outgoing coupling, Cyclic Dependency, Bottleneck, and Mega service. The anti-pattern information is extracted from the labeled system IR and used to highlight services and interactions based on the selected anti-pattern. Anti-pattern information is compared to the previous graph instance across system evolution as well to address the need to understand when and how anti-patterns developed in the system. Fig. 4 highlights Cyclic Dependency in purple, and the left panel informs that a cycle did not appear in the previous system version.

System Evolution: A timeline slider enables linear paging through major graph versions over time that was extracted and uploaded to the tool. Graph versions are grouped by a unique named identifier and include metadata about creation and update time, as well as a mock git commit number. This is to be utilized in future tool iterations to extract the system timeline from a continuous integration pipeline using our SAR process. The timeline enables us to page through versions and compare anti-pattern occurrences between system iterations to determine degradation sources. This can also pair with the 'track node' menu to focus on specific services in the graph across versions.

4 System Use Overview

The system provides two primary modes of operation that can be toggled between in the top left corner menu (Fig. 5 ref 1). Visual Mode and Anti-Pattern Mode each equipped with distinctive tools and capabilities. This subsection offers an in-depth overview of these modes and their features.

Visual Mode Visual Mode is designed to provide users with a comprehensive visualization of the system's microservices and their dependency relationships. Key features of Visual Mode include:

[3] https://dblp.uni-trier.de/rec/conf/icse/ZhouPX0XJZ18.html?view=bibtex

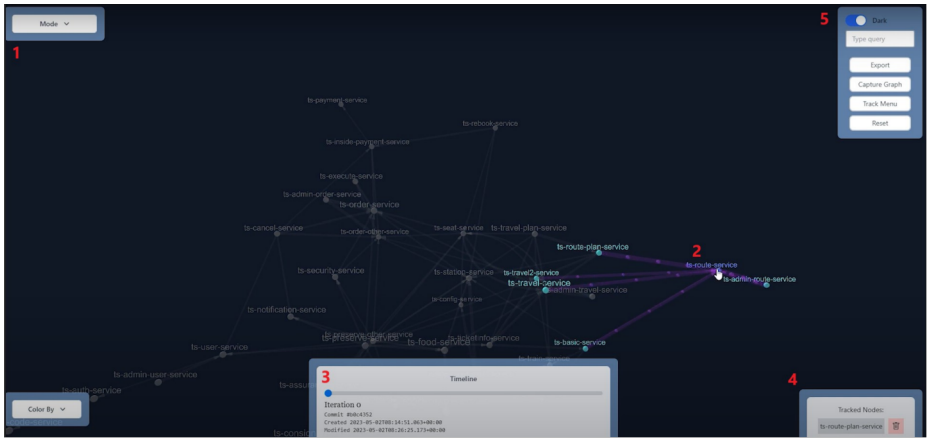

Fig. 5. MicroProspect capture with tools labeled

- **3D Visualization:** Users can explore the system in a 3D environment with six degrees of freedom.
- **Interactive Controls:** Features such as rotation, panning, and zooming in and out offer intuitive navigation.
- **Node Interaction:** Users can interact with nodes by clicking and dragging them for enhanced visualization (Fig. 5 ref 2).
- **Relationship Insight:** Hovering over nodes reveals dependency relationships. Purple links represent incoming dependencies, while yellow links represent outgoing dependencies.
- **Cyclic Dependency Visualization:** Cyclic dependencies are easily identified when a node is selected.

Fig. 6. Example of a selected Node's details

- **Node Details:** Clicking on nodes provides access to the following information (Fig 6):

- Dependency relationships.
- Endpoints encapsulated by relationships.
- Antipatterns detected on the node.
- Threshold settings, if applicable.

Furthermore, Visual Mode offers a set of tools and capabilities to enhance the user's experience:

- **Timeline:** The timeline tool allows users to explore the system's evolution by sliding between timeslices of development (Fig. 5 ref 3). Timeslices per commit provide insights into changes and evolution over time.
- **Track Nodes:** Users can track specific nodes across different timeslices by right-clicking on a node and adding it to the time slice menu (Fig. 5 ref 4). Tracked nodes are marked in green and persist through various timeslices. Users can access in-depth data for each time node by clicking on a tracked node, and they have the option to remove it from the menu.
- **Upper Right Panel:** This panel provides several toggles and options, including light and dark mode, the ability to switch between 2D and 3D views, reactive search to filter nodes based on queries, JSON schema export, capturing screenshots of the current camera angle, a track menu toggle, and a reset option to return to the default camera angle (Fig. 5 ref 5).

Anti-Pattern Mode Anti-Pattern Mode focuses on the identification of various anti-patterns within the system. Users can set specific thresholds for anti-pattern detection. Notable features of Anti-Pattern Mode include:

- **High Coupling Detection:** The system highlights high coupling by shifting affected nodes to green, orange, and red colors. Red nodes indicate high coupling, green nodes represent low coupling, and orange nodes indicate medium coupling. The coloring is controlled by user-defined thresholds or dependency relationships. Additionally, the system provides a count of nodes above the specified threshold.
- **Cyclic Dependency Identification:** Cyclic dependencies are highlighted in purple for easy recognition.
- **Bottleneck Detection:** Services at risk of becoming bottlenecks or having dependencies above the threshold are displayed in purple.
- **Megaservice Indication:** Megaservices are identified and displayed in purple.

5 Evaluation

MicroProspect renders a service dependency graph by analyzing the IR of microservices and interconnections between endpoints and REST calls. For an assessment, we deployed our MicroProspect tool[4] with a microservice benchmark Train Ticket [38] and considered thirty-six of its microservices in

[4] MicroProspect Tool: `https://cloudhubs.ecs.baylor.edu/mavp`

our analysis since we considered only Java-based microservices. Table 1 presents the outcomes of manual code analysis versus our tool, including the number of microservices and REST Calls and their corresponding service dependency graph representations.

Table 1. Service Dependency Graph Data Analysis

Numbers/Approaches	Manual extraction	MicroProspect
Microservices	36	36
REST calls	135	135
Nodes in SDG	36	36
Links in SDG	87	87
Cycles in SDG	2	2
Highly-Coupled Nodes in SDG	8	8

The above analysis depicts that our tool successfully extracted all services and REST calls from the Train Ticket system we chose to visualize. Our representation combines multiple REST calls between the same two microservices into one singular link, which explains the different number of REST calls and links in the service dependency graph. This comparison between manual analysis and our tool shows that MicroProspect can provide an accurate representation of a system through the use of a service dependency graph.

Our tool is novel in the idea of incorporating a measure of system evolution in visualization and anti-pattern detection, allowing for distinctive use cases. Scrolling through a timeline of all the commits in a codebase would allow developers to quickly identify changes and the commit associated, which could greatly reduce the costs associated with debugging. Similarly, the automatic detection and visualization of anti-patterns is a great aid in locating the causes of deficiencies at a glance. Thus, resources could be focused more on the service being provided rather than on troubleshooting a faulty or ill-performing system.

Prior Evaluation of MicroProspect: Our prior research [25] yielded insightful data that emphasized the need for our tool when applied to microservice systems. We performed a user study involving 28 participants. The study considered the detection of two anti-patterns: cyclic dependency and knot in service-dependency graphs. The outcomes revealed that manual detection of cyclic dependencies resulted in only 66% to 70% accuracy, even dropping to as low as 32% for complex systems, despite the participants' experience levels. Similarly, in knot detection, practitioners achieved only 72% accuracy in small systems and 53% in larger ones, with false positives reported by 22% to 39% of participants. Interestingly, even highly familiar developers struggled, with accuracy rates similar to or only marginally better than less familiar peers. Notably, the time spent on detection tasks didn't significantly decrease with increased familiarity, emphasizing the inefficiency of manual detection. Naturally, the visual highlight of the selected anti-pattern explained the problem instantly.

Our prior research [25] underscores the necessity for such tools within microservice systems. The study revealed the limitations of manual detection in identifying anti-patterns, with accuracy rates often falling short, regardless of practitioners' experience levels. Notably, the time spent on detection tasks didn't

significantly decrease with increased familiarity, highlighting the inefficiency of manual approaches. These findings strongly advocate for the adoption of automated tools like MicroProspect to enhance accuracy and efficiency in identifying and addressing anti-patterns within microservice architectures as demonstrated in this evaluation.

6 Related Works

Several approaches have been proposed to address the challenges of understanding and maintaining complex microservice architectures [6].

Gaidels et al. [21] explored leveraging service call graphs to identify microservice system issues using centrality and community recognition methods. Their techniques extracted meaningful metrics and visualizations, offering valuable insights into system dynamics. However, their approach lacked tool support, potentially limiting practical implementation and wider adoption. On the other hand, our solution offers a thorough analysis with integrated tool support, making it practical and accessible for evaluating and improving systems.

In the research conducted by Gamage et el. [22], they employed dynamic analysis to retrieve the dependency graph of the microservice system. By applying various graph algorithms, such as *Degree centrality* and *Clustering coefficient*, they successfully identified five common anti-patterns in the system: *Bottleneck*, *Knot*, *Cyclic Dependencies*, *Nano Service*, *Service Chain*. This approach solely relies on dynamic information for obtaining the dependency graph. Due to the dynamic nature of data extraction, sufficient time should be allocated to collect all the communication data between services. Additionally, the tool is limited to tracking synchronous systems that communicate in a RESTful style.

Cerny et al. [11] describes use of major microservice architecture tools in industry, although microservice architecture comes primarily from practitioners, so there are limited publications on the subject.

Amazon X-Ray console utilizes a map visual representation, featuring service nodes for requests, upstream client nodes for request origins, and downstream service nodes representing web services and resources. Embedded views enable users to inspect service maps and traces [11].

Netflix interactive visualization employs a service graph to depict system-wide service dependencies, allowing users to analyze different topologies by reconstructing the services communication graph. However, this approach may not be optimal for debugging specific service issues [11].

Jaeger tracing offers a Jaeger UI that renders service dependencies with dynamic data capabilities. Visualizes Directed Acyclic Graphs (DAGs) along with call frequencies to observe system architecture [1].

Kiali provides visualization tools for Istio, producing graphs representing traffic flow through the service mesh. Graph types include application, versioned application, workload, and service, each offering different levels of aggregation for system analysis [11].

With regards to additional existing tools, Engel et al. [18] developed a tool using architectural principles to uncover architectural issues in microservice systems. Their proposed approach evaluates dependency graphs based on metrics such as synchronous and asynchronous dependencies. While the tool assists in identifying design flaws, it has limited integration of graph theory concepts, potentially restricting the analysis depth. In contrast, our proposed solution takes a comprehensive approach, enabling a thorough understanding of architectural degradation and effective mitigation strategies.

MicroART [24] stands out as a tool that extracts both static and dynamic data to create a visual representation of the system's architecture. By leveraging model-driven engineering concepts, MicroART primarily focuses on recovering the system's deployment architecture and subsequently improving it. However, MicroART does not possess the capability to highlight issues and anti-patterns within the system. Hence, manual effort is required to analyze the system and identify architectural design problems.

Ma [28] introduces a tool that automatically generates the system's dependency graph for a microservice system by analyzing the source code through reflection. The tool identifies cyclic dependencies using Tarjan's Strongly Connected Component graph technique. However, its capability is limited to detecting only the cyclic dependencies anti-pattern. In contrast, our proposed method enables the analysis and comparison of multiple versions, facilitating efficient monitoring and management of architectural changes and degradation.

Several software tools are available for visualizing architecture, such as Appdash, Datadog, Dynatrace, ElasticAPM, Hypertrace, Honeycomb.io, Instana, Jaeger, Kamon, LightStep, Logit.io, Lumigo, OpenCensus, OpenTelemetry, Splunk, Signoz, Site24x7, Uptrace, Victoriametrics, and Zipkin. These tools vary in their supported programming languages, licensing models, pricing, and functionalities. For instance, Datadog is renowned for its broad language support and comprehensive monitoring capabilities, while Zipkin and Jaeger offer free distributed tracing with simplicity in visualization. On the other hand, tools like Appdash and Grafana Tempo emphasize simplicity and are available for free, although they might lack certain advanced features. Janes et al. [26] delve into a comparative analysis of these tools, discussing their features, performance, and limitations, offering valuable insights into their effectiveness in diverse architectural contexts. Nevertheless, they reveal limitations in processing visualization to identify and incorporate anti-patterns, a gap addressed by our proposed tool. Moreover, our tool emphasizes the evolutionary aspect, providing complementary features that align with the dynamic nature of architectural changes, further enriching the toolset.

Recently, Cerny et al. introduced *Microvision* [12], a cutting-edge tool that offers the ability to reconstruct and visualize microservice systems in a captivating 3D virtual reality (VR) environment. By leveraging Prophet static analysis tool [9], *Microvision* can automatically reconstruct the architecture of Java-based microservice systems. However, it's important to note that *Microvision* only relies on static analysis and is limited to Java-based

microservice systems, potentially overlooking the dynamic aspects of system behavior. Similarly, our tools share the same limitations.

7 Conclusion

The evolution of microservice systems faces multiple challenges related to decentralized development teams operating at individual microservice levels, focusing little on the overall system perspective. Such and many other factors might lead to system architecture degradation. In this work, we present a MicroProspect tool that uses a service call graph extracted from microservice systems to provide developers with a system-centered view of the system's dependencies. Given the extraction of the graph happens statically, they do not need to wait for the system to deploy and undergo comprehensive testing as common for established instruments. To mitigate architecture degradation, MicroProspect goes beyond presenting the system-centered view to developers expecting them to reason about the system. It utilizes the service dependency graph to detect and visualize selected anti-patterns. While we demonstrated a few examples, there are no limitations to continuing the effort by adding more anti-patterns to be detected. Furthermore, the evolution aspect is considered as the tool takes into account service call graphs from multiple system versions to detect differences in and inform on newly formed anti-patterns. All these aspects are integrated through an intuitive visual approach using a 3D perspective that is more likely to better cope with more complex systems. The benefit of such a visualization approach is that it points developers to the specific place in the system's architecture that needs their attention rather than presenting a plain message that the system has a certain issue, which allows developers to analyze the problem in a greater context and make an informed decision on evolution. It is our belief that this approach has the potential to significantly improve the maintainability and evolvability of microservice systems and can be integrated into existing developer tools for wider adoption.

We aim to present the advancements our tool provides to the scientific community to join efforts to aid the maintenance and evolution of infrastructure for microservice systems. There are many avenues for future work extension, including dynamic analysis integration, mining software repository integration, more experimentation, and broader anti-pattern support. In ongoing works, efforts are made to catalog over 50 microservice anti-patterns [14] and some of these can be detected and visualized to bring direct utility to developers.

8 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 2409933, and a grant from the Academy of Finland (grant n. 349488 - MuFAno).

References

1. Jaeger: Open source, distributed tracing platform, https://www.jaegertracing.io/, accessed 2023/11/17
2. Abdelfattah, A.S., Cerny, T., Taibi, D., Vegas, S.: Comparing 2d and augmented reality visualizations for microservice system understandability: A controlled experiment. In: 2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC). pp. 135–145 (2023). https://doi.org/10.1109/ICPC58990.2023.00028
3. Abdelfattah, A.S., Cerny, T.: The microservice dependency matrix. In: European Conference on Service-Oriented and Cloud Computing. pp. 276–288. Springer Nature Switzerland Cham (2023)
4. Abdelfattah, A.S., Cerny, T.: Roadmap to reasoning in microservice systems: A rapid review. Applied Sciences **13**(3), 1838 (2023)
5. Baabad, A., Zulzalil, H.B., Hassan, S., Baharom, S.B.: Software architecture degradation in open source software: A systematic literature review. IEEE Access **8**, 173681–173709 (2020). https://doi.org/10.1109/ACCESS.2020.3024671
6. Bakhtin, A., Li, X., Soldani, J., Brogi, A., Tomas, C., Taibi, D.: Tools reconstructing microservice architecture: A systematic mapping study. In: Agility with Microservices Programming, co-located with ECSA 2023 (2023)
7. Bogner, J., Fritzsch, J., Wagner, S., Zimmermann, A.: Limiting technical debt with maintainability assurance – an industry survey on used techniques and differences with service- and microservice-based systems. In: 2018 IEEE/ACM International Conference on Technical Debt (TechDebt). pp. 125–133 (2018)
8. Bogner, J., Wagner, S., Zimmermann, A.: Automatically measuring the maintainability of service-and microservice-based systems – a literature review (10 2017). https://doi.org/10.1145/3143434.3143443
9. Bushong, V., Das, D., Cerny, T.: Reconstructing the holistic architecture of microservice systems using static analysis. In: Proceedings of the 12th International Conference on Cloud Computing and Services Science-CLOSER (2022)
10. Campbell, G.A., Papapetrou, P.P.: SonarQube in Action. Manning Publications Co., USA, 1st edn. (2013)
11. 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)
12. 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 (2022). https://doi.org/10.1109/SOSE55356.2022.00012
13. Cerny, T., Abdelfattah, A.S., Maruf, A.A., Janes, A., Taibi, D.: Catalog and detection techniques of microservice anti-patterns and bad smells: A tertiary study. Journal of Systems and Software **206**, 111829 (2023). https://doi.org/https://doi.org/10.1016/j.jss.2023.111829, https://www.sciencedirect.com/science/article/pii/S0164121223002248
14. Cerny, T., Maruf, A., Janes, A., Taibi, D.: Microservice anti-patterns and bad smells. how to classify, and how to detect them. a tertiary study. SSRN Electronic Journal (01 2023). https://doi.org/10.2139/ssrn.4328067
15. Cerny, T., Svacina, J., Das, D., Bushong, V., Bures, M., Tisnovsky, P., Frajtak, K., Shin, D., Huang, J.: On code analysis opportunities and challenges for enterprise systems and microservices. IEEE access **8**, 159449–159470 (2020)

16. Cerny, T., Taibi, D.: e static analysis: opportunities, gaps, and advancements. In: Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022). Schloss Dagstuhl–Leibniz-Zentrum für Informatik GmbH (2023)
17. Conway, M.E.: How do committees invent? Datamation (April 1967)
18. Engel, T., Langermeier, M., Bauer, B., Hofmann, A.: Evaluation of microservice architectures: A metric and tool-based approach. In: Information Systems in the Big Data Era. pp. 74–89. Springer International Publishing, Cham (2018)
19. Fontana, F.A., Roveda, R., Zanoni, M.: Tool support for evaluating architectural debt of an existing system: An experience report. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing. p. 1347–1349. SAC '16, ACM (2016). https://doi.org/10.1145/2851613.2851963
20. de Freitas Apolinário, D.R., de França, B.B.N.: Towards a method for monitoring the coupling evolution of microservice-based architectures. In: Proceedings of the 14th Brazilian Symposium on Software Components, Architectures, and Reuse. p. 71–80. SBCARS '20, ACM (2020). https://doi.org/10.1145/3425269.3425273
21. Gaidels, E., Kirikova, M.: Service dependency graph analysis in microservice architecture. In: Buchmann, R.A., Polini, A., Johansson, B., Karagiannis, D. (eds.) Perspectives in Business Informatics Research. pp. 128–139. Springer (2020)
22. Gamage, I.U.P., Perera, I.: Using dependency graph and graph theory concepts to identify anti-patterns in a microservices system: A tool-based approach. In: 2021 Moratuwa Engineering Research Conference (MERCon). pp. 699–704 (2021). https://doi.org/10.1109/MERCon52712.2021.9525743
23. Gortney, M.E., Harris, P.E., Cerny, T., Al Maruf, A., Bures, M., Taibi, D., Tisnovsky, P.: Visualizing microservice architecture in the dynamic perspective: A systematic mapping study. IEEE Access (2022)
24. Granchelli, G., Cardarelli, M., Francesco, P., Malavolta, I., Iovino, L., Di Salle, A.: Towards recovering the software architecture of microservice-based systems. pp. 46–53 (04 2017). https://doi.org/10.1109/ICSAW.2017.48
25. Huizinga, A., Parker, G., Abdelfattah, A., Li, X., Cerny, T., Taibi, D.: Detecting microservice anti-patterns using interactive service call graphs: Effort assessment. In: Southwest Data Science Conference. Springer Nature Switzerland Cham (2023), (In print)
26. Janes, A., Li, X., Lenarduzzi, V.: Open tracing tools: Overview and critical comparison. Journal of Systems and Software **204**, 111793 (2023). https://doi.org/https://doi.org/10.1016/j.jss.2023.111793, `https://www.sciencedirect.com/science/article/pii/S0164121223001887`
27. Lenarduzzi, V., Lomio, F., Saarimäki, N., Taibi, D.: Does migrating a monolithic system to microservices decrease the technical debt? Journal of Systems and Software **169**, 110710 (2020). https://doi.org/https://doi.org/10.1016/j.jss.2020.110710
28. Ma, S.P., Fan, C.Y., Chuang, Y., Liu, I.H., Lan, C.W.: Graph-based and scenario-driven microservice analysis, retrieval, and testing. Future Generation Computer Systems **100**, 724–735 (11 2019). https://doi.org/10.1016/j.future.2019.05.048
29. Parker, G., Kim, S., Maruf, A.A., Cerny, T., Frajtak, K., Tisnovsky, P., Taibi, D.: Visualizing anti-patterns in microservices at runtime: A systematic mapping study. IEEE Access **11**, 4434–4442 (2023). https://doi.org/10.1109/ACCESS.2023.3236165

30. Riaz, M., Sulayman, M., Naqvi, H.: Architectural decay during continuous software evolution and impact of 'design for change' on software architecture. In: Advances in Software Engineering. pp. 119–126. Springer Berlin Heidelberg (2009)
31. Roveda, R., Arcelli Fontana, F., Pigazzini, I., Zanoni, M.: Towards an architectural debt index. In: 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). pp. 408–416 (2018). https://doi.org/10.1109/SEAA.2018.00073
32. Schiewe, M., Curtis, J., Bushong, V., Cerny, T.: Advancing static code analysis with language-agnostic component identification. IEEE Access **10**, 30743–30761 (2022)
33. Schmitt Laser, M., Medvidovic, N., Le, D.M., Garcia, J.: Arcade: An extensible workbench for architecture recovery, change, and decay evaluation. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 1546–1550. ESEC/FSE 2020, ACM (2020). https://doi.org/10.1145/3368089.3417941
34. Sigelman, B.H., Barroso, L.A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C.: Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Google, Inc. (2010), `https://research.google.com/archive/papers/dapper-2010-1.pdf`
35. Taibi, D., Lenarduzzi, V.: On the definition of microservice bad smells. IEEE Software **35**(3), 56–62 (2018). https://doi.org/10.1109/MS.2018.2141031
36. Taibi, D., Lenarduzzi, V., Pahl, C.: Microservices Anti-patterns: A Taxonomy, pp. 111–128. Springer International Publishing (2020). https://doi.org/10.1007/978-3-030-31646-4_5
37. Xiao, L., Cai, Y., Kazman, R., Mo, R., Feng, Q.: Identifying and quantifying architectural debt. In: Proceedings of the 38th International Conference on Software Engineering. p. 488–498. ICSE '16, ACM (2016). https://doi.org/10.1145/2884781.2884822
38. Zhou, X., Peng, X., Xie, T., Sun, J., Xu, C., Ji, C., Zhao, W.: Benchmarking microservice systems for software engineering research. In: The 40th International Conference on Software Engineering. p. 323–324. ICSE '18, ACM (2018). https://doi.org/10.1145/3183440.3194991