Proc. of the International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME 2024)
4-6 November 2024, Male, Maldives

Designing a System-Centered View to Microservices Using Service Dependency Graphs: Elaborating on 2D and 3D visualization

Patrick Harris
patrick_harris3@baylor.edu
CS, Baylor University
Waco, TX

Mia Gortney
mia_gortney1@baylor.edu
CS, Baylor University
Waco, TX

Amr S. Abdelfattah
amr_elsayed1@baylor.edu
SIE, University of Arizona
Tucson, AZ

Tomas Cerny
tcerny@arizona.edu
SIE, University of Arizona
Tucson, AZ

Pablo Rivas
pablo_rivas@baylor.edu
CS, Baylor University
Waco, TX

***Abstract*—Cloud-native systems have become widely embraced by the industry. The decentralization characteristic of these systems provides many benefits, which have led to their widespread adoption. However, while cloud-native systems have solved many of the limitations of monolithic systems, they have introduced their own problems. A key issue is decentralization, which makes it difficult to capture a view of the system as a whole. Without a system-centric perspective, developers resort to updating assigned services without analyzing the implications on the overall system. As a result, a software system can degrade over time by introducing technical debt. A holistic system view, such as a service dependency graph, is the instrumental artifact that can help developers understand system dependencies, discern future extensions, or aid with change impact analysis. However, the established visualizations of these graphs are static, aiming to provide all information at once, which leads to complex and hard-to-understand graphs when representing microservice systems. In this paper, we advance the visualization of service dependency graphs with interactive features. We present an interactive and reactive two-dimensional and three-dimensional graph view for microservice systems that aims to simplify analysis of complex systems when looking for dependencies across services.**

***Index Terms*—Software Architecture Reconstruction, Static Analysis, Microservices, Cloud-Native**

I. Introduction

Microservice architecture is the standard for cloud-native systems. Its decentralized nature allows the system to handle much larger payloads by distributing the work to multiple services. Multiple services also allow developers to work independently and institute their own frameworks. Benefits such as these are the reason companies have moved from monolithic systems to microservices.

With these benefits and solutions to monolithic problems come new issues. One major issue is distributed system architectural degradation, which occurs because there is no centralized view of the system. Moreover, with developers working independently on individual microservices and not reconciling their changes, the system can decay over time into a completely different version from what was intended. Independent parts may not work together as a whole due to different development environments or even different programming languages. Issues can go unnoticed by other developers if they are not working on that part of the system.

With a holistic view of the system's service dependencies, developers could better understand the impact of source code changes on other services and avoid ripple effects. We performed multiple systematic literature reviews to find the most common architectural views providing a holistic perspective for microservices, and identified service dependency graphs [1]–[3]. However, current graph visualizations lack interactivity and present a static, all-in-one solution.

There is a need for more advanced visualization that provides interactivity [4], [5] to help developers better understand the system, its design qualities, and the implications of their code changes to selected system parts. In the context of cloud-native systems where the number of microservices grows significantly, an established approach to service dependency graph visualization renders too complex and too detailed to help practitioners understand the system [6]; and novel visualization approaches are necessary to adopt the microservice system specifics. In this paper, we address a novel interactive visualization for service dependency graphs.

Problem Statement: Established visualizations of service dependency graphs are static all-in-one solutions that lack interactivity and available data visualization advancements. They do not align with the complexity brought by microservices, where many decentralized services interact and render too complicated to manually reason about when dealing with system quality assessments.

Paper contribution: To provide interactive service dependency graph visualization that takes into account large-scale microservice systems, we investigate a 2D and 3D solution. We implement a visualization prototype that can render a high-level system-centered view. Finally, we use a large third-party system benchmark to demonstrate our solution.

The rest of the paper is organized as follows. Section II gives background into different visualization techniques and the benefits and issues they present. Section III details works related to our topic. Section IV describes our software reconstruction method, which includes static analysis and usage of a service dependency graph. Section V details the implementation of our visualization prototype. Section VI discusses our findings overall, the benefits they offer, and potential extensions and applications for our research. Finally, we conclude our paper in Section VII with a general summary of our implementation and research.

II. Background

In order to create a high-level view of the system, data about the system must be extracted in order to reconstruct [6] and understand it. Two general approaches exist for this extraction process: static and dynamic analysis.

Dynamic analysis [7], [8] is used primarily in the industry to extract service dependency graphs. Tracing tools such as Jaeger [9] and OpenTelemetry [10] can capture and monitor calls between microservices at runtime. These calls can be instrumented and aggregated to determine metrics such as end-to-end delay and failed calls. The metrics can then be used to determine problem areas within a system and treat them by implementing load balancing and hot swapping of microservices. Some small-scale visualizations, like Jaeger's service dependency graph generated from a single trace, can help to more easily see these metrics aggregated, rather than having to manually parse text [9]. However, this graph is static and limited to the data collected by tracing, not discovering many of the dependency relationships that are not executed during a trace. In addition, the distributed tracing approach does not scale well as an architecture expands over time since the implemented mitigation efforts seek to treat the symptoms as opposed to curing the underlying issues.

On the other hand, static analysis [11] strives to understand the system as it stands, without events taking place at runtime. Static analysis focuses on analyzing the code and tracking dependencies based on what is represented in the code structures. Typically, this involves parsing the code and generating abstract syntax trees and control-flow graphs that can be reduced to call graphs. However, one issue comes with system design where microservices typically build on components, and common program analysis considers low-level code [12]. Thus, many works in this area focus on language-specific solutions or include introspection methods like reflections which provide the interface perspective of used components rather than complete detail. Similar to dynamic analysis, there are limitations. Here, the limits are language dependence and restrictions on detecting components. Successful demonstration of analyzing microservice systems using static analysis has been demonstrated by Walker et al. [13], while limited to the Java platform.

No matter the approach, after extracting the structure of the system, this data can be aggregated and analyzed to visualize the system. One specific view of a decentralized system is a service dependency graph which provides a high-level view of the services that make up the system and the dependencies between them. This type of graph is the perfect framework for visualizing an entire microservice system. The visualization is simple but easily extendable. The prototype we introduce in this paper utilizes static analysis to form its own service dependency graph. We also demonstrate how such a graph can be extended to have more functionality.

III. Related Work

As the issues of architectural degradation within microservice architecture have become more apparent, research has been conducted into how this issue can be overcome through software architecture reconstruction and the outcome visualization to practitioners.

Rademacher et al. outline a modeling method to reconstruct system architecture in order to enable an analysis of the system that can help prevent architectural degradation [14]. By collecting the domain concepts, services, and operations in a system, a decentralized system can be reliably modeled. Such a model can be used to understand the architecture and its implementation and identify areas that could degrade the system. Their approach is broken down into six distinct steps. The first phase, preparation, consists of gathering the input for reconstruction, like documentation, source code, and scripts, as well as inputting any known information about the system, like the technology stack. In the next phase, domain modeling, domain concepts are identified as well as their bounded context, which is a collection of domain objects within the same scope [15]. Each microservice should be constrained to one context [16]. The third phase, service modeling, revolves around identifying microservices. Framework-based annotations are oftentimes useful for doing this, like the `@RestController` annotation in the Spring framework [17]. Next, in the operation monitoring phase, operation nodes, like the deployment containers, are uncovered using artifacts like Dockerfiles [18]. In the fifth phase, technical refinement, any information on the technology stack that was not found in the previous phases is exposed. Finally, the post-processing phase refactors the models from previous phases to form a more cohesive model and runs verification checks.

Bushong et al. devised a method to extract endpoints and calls from a decentralized system's codebase using static analysis [19]. The advantage of this method is a view of the system architecture can be generated before the deployment of the system and updated as the code changes. Their method was implemented in a prototype called Prophet [20] that uncovers the entities, properties, and relationships from a codebase. Prophet [20] generates an intermediate representation in the form of UML diagrams, describing the entities, properties, and relationships within a system. In order to accomplish this, Prophet [20] takes a two-step approach. The first step is extracting a context map from the system. This is done by first identifying the classes in each microservice using source code analysis or bytecode analysis. Then, the classes must be filtered to distinguish data entities from other classes. After filtering, a context map can be built by uncovering the relationships among the entities. The second step is to generate a communication diagram. API endpoints and where these endpoints are referenced are identified with further code analysis. The format of these endpoints is standardized, such as with HTTP, and thus makes this approach reliable. However, since this approach relies on structure uncovered using a specific syntax, it is not versatile and is limited to specific technology stacks like the Spring framework [17]. Also, testing with the prototype confirmed that some complex behavior, like multiple URLs referring to one call, may not be discovered. These behaviors may not be easily uncovered in the codebase, but instead revealed dynamically as a system is running.

Cerny et al. implement models on top of the Prophet prototype in order to visualize the entities and relationships in augmented reality [5]. The resulting immediate representation from the static analysis done by Prophet [20] is analyzed with system reasoning to generate a system-centric view. Their prototype, Microvision [21], chooses to represent the system by combining the service and domain contexts uncovered by Prophet [20] to illustrate the scope of all entities in the system. The visualization is composed of two parts: a graph display and an API view. The graph display shows the system in three-dimensional space where a node represents a microservice and links represent the service dependencies among microservices. Each microservice may be composed of one to many API endpoints which can be shown by selecting a microservice, supplying a level of granularity where users can see the details of the overall system or of a specific service. Microvision [21] provides a high level of abstraction that helps make the system easily understandable and navigable.

Another approach taken for the visualization of a software system, described by Fittkau et al., is viewing software architecture as a metaphorical city [22]. In a 'software city' as they describe, a trace is visualized with two parts: open and closed packages. Open packages, denoted by flat green boxes, show the internal details of the package, like sub-packages and classes. Green boxes on the top represent closed packages that do not expose their internals. Classes are shown as small purple boxes and orange lines show communication. The width of these lines shows the frequency of different calls and the height of the classes represents the number of active instances. This model is displayed in virtual reality to provide a fluid method of interactivity. They found using gestures for control provided a natural way for users to examine the model. Moving the model is done like physically grabbing an object, rotating the model is done like spinning a ball with two hands, and scaling the model is done by tilting the upper body. While the approach taken here is very interactive and intuitive, the package structure is limited and not ideal for illustrating complex relationships in a decentralized system.

IV. Used Software Architecture Reconstruction Method

We chose to use static analysis to generate a system view in our prototype as dynamic analysis has a few large drawbacks, namely that sufficient runtime metrics are needed to ensure all behaviors of the system are captured [23]. For this reason, the following discussion will be focused on static analysis.

Static analysis functions by examining the source code without executing it. The code statically defines the structure of the system including service endpoints and dependencies. This structure is made apparent by the objects defined within the code and the external objects referenced by them. Thus, by analyzing the source code, a map of the system can be generated [24]. The general process of static analysis, described in Fig. 1, is extracting the structure, or model, from the source code and describing it with some kind of intermediate representation that can then be analyzed and utilized in a variety of ways, including for visualization.

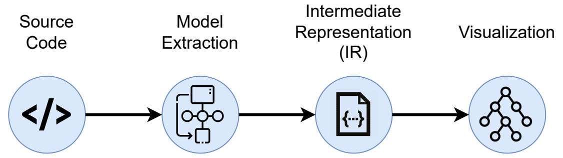


Fig. 1. Static Analysis Process

However, oftentimes in a cloud-native system, the codebase is spread across many repositories and implemented with many languages. Since this is the case, a tool for static analysis is language-dependent, meaning that it must be able to process the syntax of all the languages that make up a particular system.

Unlike dynamic analysis, static analysis does not require system instrumentation at runtime, so there is no overhead placed on the system. Furthermore, instrumentation of the entire codebase ensures that all services and static dependencies are captured. Overall, static analysis provides an effective means to reconstruct the services in a cloud-native system and how those services are interconnected.

Through the use of static analysis, a service dependency graph can be created. A service dependency graph is made up of nodes and links, as shown in Fig. 2. The nodes represent the different microservices in the system. The links represent the service calls that create dependencies between services [25]. The service dependency graph's main function is to provide a high-level view of the system with limited details. The simple and concise nature of a service dependency graph makes it ideal to visualize in our prototype; nodes and links can be easily represented on a screen while still retaining a host of information about the system's makeup.

V. Microservice Visualization Prototype

Analysis:: Microservice systems can contain hundreds of nodes that are connected through remote calls or messages,

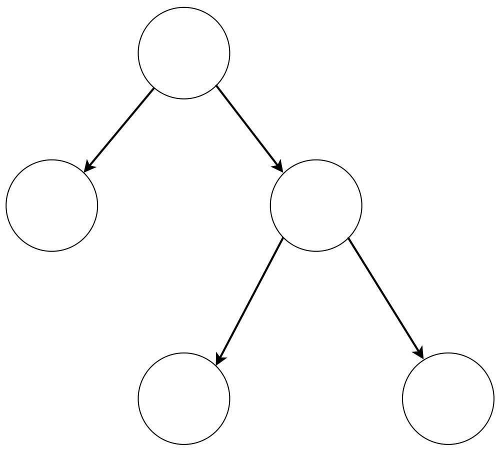


Fig. 2. Service Dependency Graph

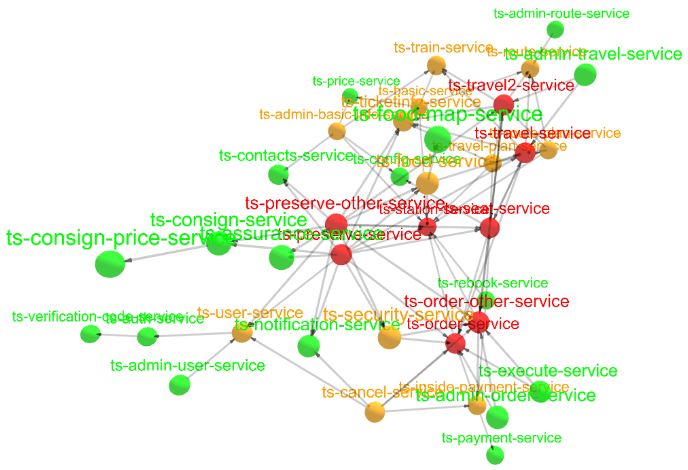


Fig. 3. Visualization of Train Ticket Service Call Graph in 3D

leading to dependencies. To deal with system complexity, 2D space might render limitations that can be better solved in 3D. Moreover, the perspective should not consider an all-in-one static solution but rather engage developers with selective options. For instance, service nodes should become selectable to render more details about their connections illustrated via edges, or one should be able to drag a selected node to highlight its specifics, as developers might limit their work on that one specific node. Cerny et al. [5] approached an interactive model of service dependency graph in augmented reality. In a later user study, Abdelfattah et al. [4] identified that such a model brings many benefits with space rendering, however, there are outstanding features that practitioners expect such as the ability to search. Furthermore, augmented reality requires a distinct interaction with a camera-equipped device that is used to manipulate the graph. For long-term interaction, this might be impractical as it is disjoint from the workstation used to develop the system. This brings the opportunity to fill the gap with web-based interactive models that can render nearly any system and enable team collaboration.

Prototype:: To develop our visualization prototype of the service dependency graph we started by considering the large third-party microservice system benchmark Train-Ticket [26]. The main objective of the prototype was to visualize a microservice architecture in a holistic and interactive manner that provides a quality level of detail while making it easy to use and find information. Ultimately, we aimed at creating a user-defined experience that can be useful to both novices and experts in a variety of computing fields. We chose JavaScript for our codebase to support a lightweight web application for ease of use. Data visualization is well-established in JavaScript. For instance, D3 library provides a two-dimensional force-directed graph as well as capabilities for a force-directed graph in three-dimensional space.

Model intermediate representation:: An important piece of the prototype is how the visualization is generated. This core functionality is provided through a JSON schema that statically describes a system at a high level. A decentralized system is described as a collection of nodes (services) and links (dependencies between services). This schema can be easily extracted through static analysis of a system's codebase [19]. Our prototype takes in this schema as input and generates the visualization as the collection of nodes and links that the schema specifies. Using static analysis, we extracted this JSON schema from the Train Ticket benchmark microservice system [26], and the resulting visualizations are shown in Fig. 3 and Fig. 4.

The schema is also easily extensible to allow for a more enhanced visualization that provides more information in an easily accessible manner. For example, by providing a node type, meaning whether a system part is a service, database, proxy, storage, or some other kind of element, we can render a visualization with multiple shapes where each shape maps to a certain kind of system part.

Interactive features:: To create an interactive user-defined experience, many features were considered on top of the baseline visualization.

- Each node can be dragged and moved around in order to allow for better manipulation of the data.
- Hovering over a node, displayed in Fig. 6 and Fig. 7, simplifies the observability of other nodes this service is connected to.
- Clicking on a node can show an information box that provides more detail about that node as shown in Fig. 5.
- An interactive search, highlighted in Fig. 8 and Fig. 10, makes it quick to find needed information, especially in a large, decentralized system consisting of hundreds of nodes.
- Moreover, each node is dynamically colored according to the degree to which it is coupled with other nodes, which can be user-defined.
- Additionally, a right-click context menu, shown in Fig. 9, provides access to multiple options to better be able to interact with the nodes and links.
  - These options include limiting a graph to just a node

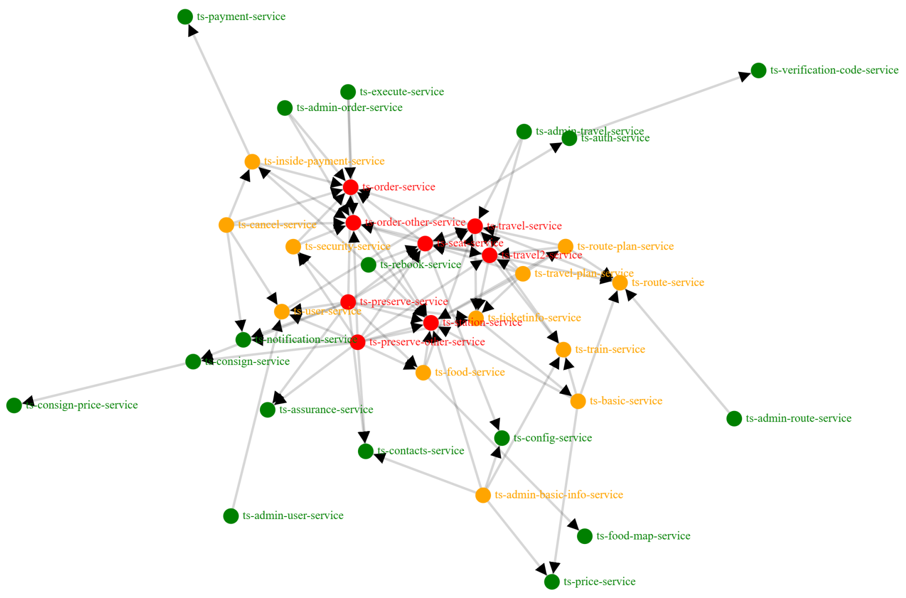
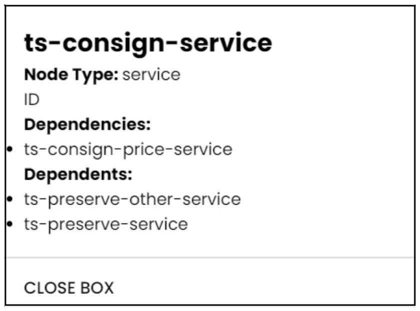


Fig. 4. Visualization of Train Ticket Service Call Graph in 2D

and its neighbors, adding a new link, deleting a node, keeping track of a node as a user traverses a large graph, showing how data moves from one node to others, and highlighting nodes to distinguish them. This menu enables additional features for user interaction with the system.

Another avenue explored by the prototype is predicting system evolution. Services and their dependency relationships can be added or removed from the visualization to illustrate how these changes may impact the architecture at large. Developers could test how the introduction, modification, or removal of a microservice could change the system before deploying the changes. This aims to help prevent ripple effects where a small change may have unintended adverse effects.

Other possible features:: In the future, we hope to further enhance the usefulness of the prototype by expanding the scope of features offered. One option is incorporating dynamic analysis which could help to visualize the flow of data throughout a system when a request is made. Furthermore, by extending the JSON schema to support dynamic data about microservices, real-time monitoring and data flow analytics could be displayed. Another direction could be presenting views beyond the service dependency graph. A physical architecture view could open the door to displaying performance metrics such as CPU, RAM, and disk utilization by different microservices. Alternatively, providing an option to switch to


ts-consign-service

Node Type: service

ID

Dependencies:

- ts-consign-price-service

Dependents:

- ts-preserve-other-service
- ts-preserve-service

CLOSE BOX

Fig. 5. Information Box Describing a Node

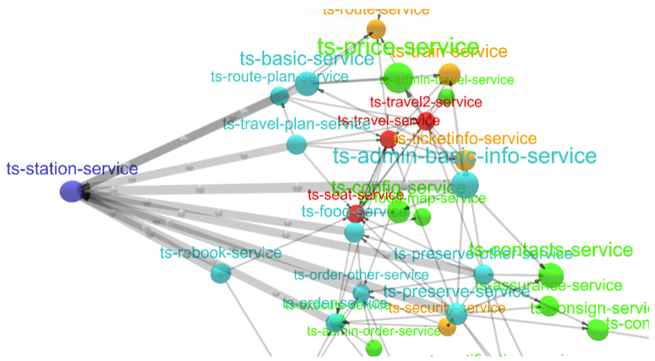


Fig. 6. Hovering Over A Node in 3D

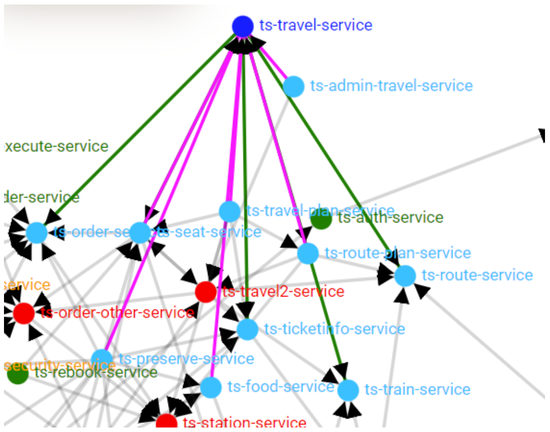


Fig. 7. Hovering Over A Node in 2D

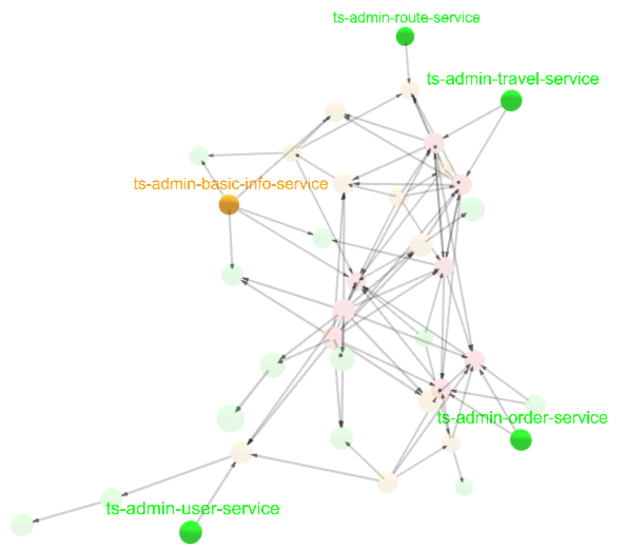


Fig. 8. 3D Visualization of Query for 'ts-admin' Service

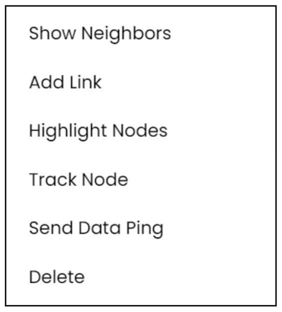


Fig. 9. Context Menu To Further Interact with Node

a lower-level call graph that exposes the endpoints within each microservice may help to provide more granularity. Another option is implementing a time scale to show how the graph changes over time. A third option is using the visualization to highlight deficiencies, also called code smells, within the system. Illustrating where a cyclic dependency or knot exists can help raise awareness of system issues and guard against architectural degradation. There are lots of opportunities for extension in different directions through the customizability of our visualization. The size of links, color of nodes and links, or grouping of nodes can be manipulated to reflect different attributes. All of these avenues are achievable through the availability and ease of use of our prototype.

VI. Discussion

In our microservice visualization prototype, we used static analysis to create a 2D and 3D service dependency graph. Throughout deciding on this software architecture reconstruction technique and implementing it in our prototype, we have discovered several benefits of using this particular approach.

One benefit to using static analysis in conjunction with a service dependency graph is simplicity. At its core, a service dependency graph is composed of services and dependencies. These are relatively simple to derive from the code base and visualize as a whole.

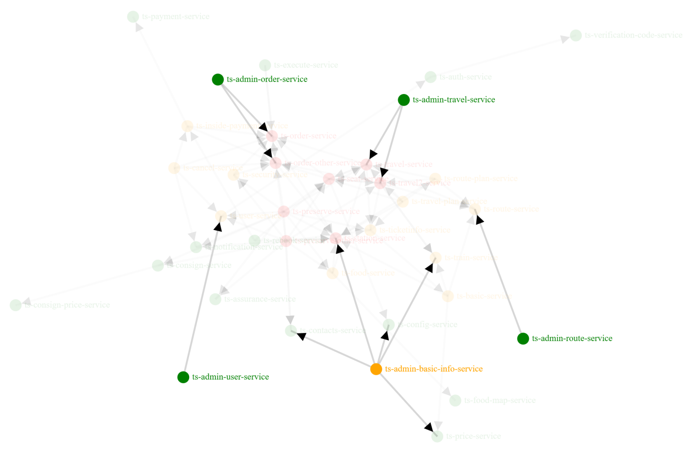


Fig. 10. 2D Visualization of Query for 'ts-admin' Service

Another benefit we found is that by using the service dependency graph, we can extend the visualization's features more if needed. It provides the opportunity to add dynamic analysis and visualize data flows in the service dependency graph as well as other additional functionality. This ability to extend is an advantage to the developer as they can pick and choose what they would like to see. It also creates the opportunity for multiple views where each graph serves a different purpose.

Our prototype showcases some of these benefits by illustrating the different features we were able to add to the original service dependency graph. We were able to classify different node types and groups. The user has the ability to dynamically add and remove nodes and links as they please. We also have a system that identifies high coupling based on the dependencies. Even with all these new features, our graph remains simple to look at as well. Being able to move the nodes around and rotate the graph enables the user to create different views too.

Overall, our prototype serves not only as a tool for generating a service dependency graph from a JSON schema but more importantly as a framework that supplies a foundation for how cloud-native systems can be visualized. The visualization this framework outlines has many practical applications including combating architectural degradation, change impact analysis, analyzing performance and health metrics, and providing general organization for large-scale projects.

While our framework is a good foundation for cloud-native system visualization, the question of whether 2D or 3D visualization is more useful still needs to be answered. We have conducted a user study with 6 expert participants to determine if 2D or 3D visualization is more desirable for understandability and usage. In this study, each participant interacted with two different versions of either a small or large system with the 2D and 3D visualizations. They were given a list of questions to answer such as finding certain nodes or implementing a specific feature. After interacting with the system, we asked for their feedback on what they liked, what they didn't like, if they would recommend the prototype for daily work, and if they preferred one view over the other. At the study's conclusion, the participants favored the 2D system. The overall accuracy of their answers was similar between 2D and 3D, though the 3D system took longer to navigate. As of now, the 3D model does not outperform the 2D model, but with further development, there is potential for the future.

The prototype is available for future research and general use on GitHub.[1]

VII. Conclusion

Using a microservice system offers several benefits with architectural degradation as its con. One way to combat architectural degradation is to completely understand the system through service dependency graphs that promote expert reasoning about planned changes or their impact. This paper demonstrates advancements in service dependency graph visualization that take into account the specifics of microservice systems and illustrates how these graphs can become interactive to better serve developer needs. The service dependency graph offers simplicity while still being easily extensible. We demonstrate the usefulness of this approach through our 2D and 3D visualization prototypes. The visualization advancements are not bound to the specific system data extraction; however, we illustrated the connection with the software architecture reconstruction process that involves using static analysis to create an intermediate representation of the service dependency graph, which we visualized for a large third-party system benchmark.

In future work, we aim to elaborate on additional features and continue to discern whether a 2D or 3D approach would be more useful for visualizing microservice systems.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. OISE-1854049, OISE-2409933, and CISE-CNS 2210091.

References

[1] 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)*, 2022, pp. 39–48.

[2] M. E. Gortney, P. E. Harris, T. Cerny, A. A. Maruf, M. Bures, D. Taibi, and P. Tisnovsky, "Visualizing microservice architecture in the dynamic perspective: A systematic mapping study," *IEEE Access*, vol. 10, pp. 119 999–120 012, 2022.

[3] G. Parker, S. Kim, A. A. Maruf, T. Cerny, K. Frajtak, P. Tisnovsky, and D. Taibi, "Visualizing anti-patterns in microservices at runtime: A systematic mapping study," *IEEE Access*, vol. 11, pp. 4434–4442, 2023.

[4] A. S. Abdelfattah, T. Cerny, D. Taibi, and S. Vegas, "Comparing 2d and augmented reality visualizations for microservice system understandability: A controlled experiment," in *The 31st IEEE/ACM International Conference on Program Comprehension (ICPC 2023)*. IEEE, 2023.

[5] T. Cerny, A. S. Abdelfattah, V. Bushong, A. Al Maruf, and D. Taibi, "Microvision: Static analysis-based approach to visualizing microservices in augmented reality," in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2022, pp. 49–58.

[6] A. S. Abdelfattah and T. Cerny, "Roadmap to reasoning in microservice systems: A rapid review," *Applied Sciences*, vol. 13, no. 3, 2023. [Online]. Available: https://www.mdpi.com/2076-3417/13/3/1838

[7] A. Al Maruf, A. Bakhtin, T. Cerny, and D. Taibi, "Using microservice telemetry data for system dynamic analysis," in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2022, pp. 29–38.

[8] A. Bakhtin, A. Al Maruf, T. Cerny, and D. Taibi, "Survey on tools and techniques detecting microservice api patterns," in *2022 IEEE International Conference on Services Computing (SCC)*, 2022, pp. 31–38.

[9] "Open source, end-to-end distributed tracing." [Online]. Available: https://www.jaegertracing.io/

[10] "What is opentelemetry?" Jun 2022. [Online]. Available: https://opentelemetry.io/docs/concepts/what-is-opentelemetry/

[11] T. Cerny, J. Svacina, D. Das, V. Bushong, M. Bures, P. Tisnovsky, K. Frajtak, D. Shin, and J. Huang, "On code analysis opportunities and challenges for enterprise systems and microservices," *IEEE Access*, vol. 8, pp. 159 449–159 470, 2020.

[12] M. Schiewe, J. Curtis, V. Bushong, and T. Cerny, "Advancing static code analysis with language-agnostic component identification," *IEEE Access*, vol. 10, pp. 30 743–30 761, 2022.

[13] A. Walker, I. Laird, and T. Cerny, "On automatic software architecture reconstruction of microservice applications," in *Information Science and Applications*, H. Kim, K. J. Kim, and S. Park, Eds. Singapore: Springer Singapore, 2021, pp. 223–234.

[14] F. Rademacher, S. Sachweh, and A. Zündorf, "A modeling method for systematic architecture reconstruction of microservice-based software systems," *Enterprise, Business-Process and Information Systems Modeling21st International Conference, BPMDS 2020, 25th International Conference, EMMSAD 2020, Held at CAiSE 2020, Grenoble, France, June 8–9, 2020, Proceedings*, vol. 387, p. 311—326, January 2020. [Online]. Available: https://europepmc.org/articles/PMC7254549

[15] E. Evans and E. J. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

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

[17] "Spring." [Online]. Available: https://spring.io/

[18] "Docker compose overview." [Online]. Available: https://docs.docker.com/compose/

[19] V. Bushong, D. Das, and T. Cerny, "Reconstructing the holistic architecture of microservice systems using static analysis," in *Proceedings of the 12th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, INSTICC. SciTePress, 2022, pp. 149–157.

[20] Cloudhubs, "Prophet," https://github.com/cloudhubs/prophet, 2021.

[21] ——, "Microvision," https://github.com/cloudhubs/microvision, 2021.

[22] F. Fittkau, A. Krause, and W. Hasselbring, "Exploring software cities in virtual reality," in *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, 2015, pp. 130–134.

[23] K. Sellami, M. A. Saied, A. Ouni, and R. Abdalkareem, "Combining static and dynamic analysis to decompose monolithic application into microservices," in *Service-Oriented Computing*, J. Troya, B. Medjahed, M. Piattini, L. Yao, P. Fernández, and A. Ruiz-Cortés, Eds. Cham: Springer Nature Switzerland, 2022, pp. 203–218.

[24] 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)*, 2021, pp. 1199–1201.

[25] S.-P. Ma, C.-Y. Fan, Y. Chuang, W.-T. Lee, S.-J. Lee, and N.-L. Hsueh, "Using service dependency graph to analyze and test microservices," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 02, 2018, pp. 81–86.

[26] 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

[1]Prototype: https://github.com/patrickeharris/ArchitectureVisualizationPOC