Microvision: Static analysis-based approach to visualizing microservices in augmented reality

Tomas Cerny
Department of Computer Science
Baylor University
Waco, Texas, United States
tomas_cerny@baylor.edu

Amr S. Abdelfattah
Department of Computer Science
Baylor University
Waco, Texas, United States
amr_elsayed1@baylor.edu

Vincent Bushong
Department of Computer Science
Baylor University
Waco, Texas, United States
vinbush@gmail.com

Abdullah Al Maruf
Department of Computer Science
Baylor University
Waco, Texas, United States
maruf_maruf1@baylor.edu

Davide Taibi
CloudSEA.AI Group
Tampere University
Tampere, FI-33720, Finland
davide.taibi@tuni.fi

***Abstract*—Microservices are supporting digital transformation; however, fundamental tools and system perspectives are missing to better observe, understand, and manage these systems, their properties, and their dependencies. Microservices architecture leans toward decentralization, which yields many advantages to system operation; it, however, brings challenges to their development. Microservices lack a system-centric perspective to better cope with system evolution and quality assessment. In this work, we explore microservice-specific architecture reconstruction based on static analysis. Such reconstruction typically results in system models to visualize selected system-centric perspectives. Conventional models are limited in utility when the service cardinality is high. We consider an alternative data visualization using 3D space using augmented reality. To begin testing the feasibility of deriving such perspectives from microservice systems, we developed and implemented prototype tools for software architecture reconstruction and visualization of compared perspectives.**

***Index Terms*—Microservices, Software Architecture Reconstruction, Visualization, Augmented Reality, System-centric view**

I. Introduction

Cloud-native systems frequently use microservices architecture, which revolutionized how we design, develop, and operate software systems. The primary goal of microservice architecture is to facilitate the scalability of specific system features, which means dividing the system into self-contained, self-deployable, and easy to scale-out microservices. Best practice guidelines for cloud-native systems, such as Heroku's twelve-factor app[1], suggest practices to build high-quality microservices along with system infrastructure leading to simplified management, monitoring, microservice evolution, resilience, robustness, etc.

With all benefits come drawbacks. Surveys on microservice practices and challenges [1], [2] revealed common concerns about microservice architecture, including no system-centric view, problems with overall system evolution, inter-service dependencies, and architectural complexity.

Each microservice resides in its own self-contained codebase, isolated from the holistic system perspective. This isolation promotes developers' reasoning, they often base the design decisions on the context they are familiar with, which does not necessarily lead to an optimal solution for the overall system. This is where the missing system-centric perspective could help to broaden their reasoning context.

Recent technologies make it possible to determine system communication paths across microservices using a centralized log or call tracing (e.g., Spring Cloud Sleuth [3]). This could partially determine the system-centric perspective. However, it requires all microservices to be deployed and to operate first. Typically, system endpoints are identified along with interdependencies and the dynamic characteristics and metrics of the overall system. However, these technologies are developed for system operation and DevOps engineers. Moreover, they expect user interaction or user simulation tests. Thus, additional efforts are needed to develop tests or wait for user interaction before using these tools. Therefore, such approaches might be challenging to adopt in development pipelines and serve developers, especially considering microservices evolve in an isolated codebase, and we want to reason about the most recent system state, not necessarily the deployed version.

Moreover, to reveal the white (or at least gray) box system architecture for the system-centric view, we must assess the codebase. Suppose we could derive the microservice dependencies solely using static analysis, aggregating results across codebases. As a result, the holistic system reasoning would be greatly simplified and closer to developer practices. The first step to achieve this would require determining the dependencies between microservices [4]. To address this, we

This material is based upon work supported by the National Science Foundation under Grant No. 1854049, grant from Red Hat Research, and Ulla Tuominen (Shapit).

[1]https://12factor.net

could focus on microservices' bounded contexts [5], recognize the underlying microservice data models, and then determine data model overlaps based on the similarity of data entities (fields, names, types, etc.) across distinct microservices [6]. In addition, we can identify remote calls with their specific parameters and bind them to other microservice endpoints that match [6], [7].

No matter the used approach to reconstruct the software architecture [8], the aggregated information must be presented to practitioners in an understandable way, as the quantity of information can be exhaustive. For example, the Software Architecture Reconstruction process [9] recognizes multiple views for different needs; however, the interplay of microservice systems can still be overwhelming. Thus, we must ask a question about the appropriate visualization strategy.

Current approaches that apply dynamic analysis to support the system-centric perspective and seek opportunities in established visual models, typically rendering in the two-dimensional space. We use static analysis to demonstrate it is feasible to derive a system-centric perspective highlighting inter-service dependencies. Moreover, we believe that a more efficient visualization direction should use three-dimensional space to render the system model since it copes better with the ever-growing size requirements of cloud-native systems. We propose static code analysis performed on cloud-native system codebase and its visualization in augmented reality rendered in three-dimensional space. We build on the prototype tool Prophet using static analysis which provides input to the Microvision visualization prototype tool.

The main contribution of this article details how a static analysis-based method for software architecture reconstruction of microservices can be utilized for their visualization. Our approach is with a proof of concept tool tested on a large third-party system testbench. Such reconstruction products can be used for system reasoning. We use it to address the missing system-centric view in these systems that would outline inter-service dependencies. In order to utilize such a reconstruction product, we sought an appropriate presentation suitable for practitioners to easily interpret microservice system details to address common tasks and locate quality aspects. Given the potentially large scale of microservice systems, we research three-dimensional visualization approaches and construct a proof-of-concept visualization in augmented reality, which we compare with conventional two-dimensional representation for the service view. There are benefits and drawbacks to the three-dimensional approach.

The organization of this article is as follows: Section II discusses related work. Section III outlines the Software Architecture Reconstruction (SAR) process. This is followed by Section IV, with conventional and augmented visualization assessed on a third-party large testbench system. We assess conventional visualization and augmented visualization in Section V through the use of a small case study. Finally, Section VI draws conclusions and outlines future work.

II. Background and Related Work

Software architecture is the central focal point of the system's development and design. Since systems evolve we must often reconstruct the architecture from the actual system. Software Architecture Reconstruction (SAR) is the process by which the architecture of an implemented system is obtained from the existing system [9]. Once we have such architecture reconstructed we can reason about the system (i.e, conformance checking, verification, evolution, modification, extensions, documentation generation, etc.), or visualize the system. Since software architecture is complex and interpreted differently for various system aspects, it can be described by architectural views [10], [11]. These views capture certain system qualities or aspects. The foundation for successful SAR is the ability to reconstruct effective architectural views of a system [5]. Existing SAR work related to microservices by Rademacher et al. [12] has considered four views as their outcome. In particular, it operated with domain, technology, service, and operation views. Each of these views considered a specific perspective and related concerns within the system. However, each also relates to other views. As an example, consider the service view overlapping with the domain view to detail which data entities are involved in endpoints. The technology and domain view will then show where the data entities persist.

A. Static and Dynamic Analysis Visualization

The process of Software Architecture Reconstruction can be exhaustive with manual efforts or involve automation [12]. Given that microservices are decentralized, the whole process becomes even more complicated since multiple codebases can be involved. Rademacher et al. [12] manually collected architecture-related artifacts, constructed a canonical representation of the data model, and based on that fused module views. They then performed architecture analysis on the results to answer hypotheses about architecture implementation from the reconstructed architecture information.

Considering current practices in microservice development and operations (DevOps), *dynamic* analysis can be used involving tracing. Tracing adds a tracing identifier to log messages generated throughout the system interaction, which allows us to centralize these messages via centralized logging and interpret their content in the holistic context by observing inter-microservice dependencies. It is common to extract dependency graphs such as directed acyclic graphs [13]. The advantage of this approach is platform independence. However, received log messages and their origins only lead to an abstract reconstruction providing more or less a black-box view. The industry practice provides monitoring, tracing, and metrics tools to capture data about the microservices [3] (i.e., OpenTelemetry, Zipkin, or Jaeger [2]). These tools seamlessly integrate with enterprise frameworks and utilize existing mechanisms such as method call interception or instrumentation. Mayer and Weinreich use the Spring framework's interceptors

[2]https://opentelemetry.io; https://zipkin.io; http://jaegertracing.io

to monitor runtime calls between services to generate an architectural view of a microservice system [14]. However, it is also possible to use API gateway [15]. Granatelli et al. [16] approached the challenge by querying the containerization framework to retrieve calls between microservices at runtime.

The *static* analysis perspective can be constructed from artifacts available before deployment. Analyzing a program's codebases has played a part in the formal verification of a system's correctness [17], [18] and other fields. However, the major challenge is the decentralized codebase. In the realm of microservices, it has been used to identify calls between microservices to generate security policy automatically [19]. Also, it has been used to analyze monolithic applications to recommend splits for converting to microservices [20]–[22]. In generating a service dependency graph, Esparrachiari et al. [23] posit that source code analysis is not sufficient since the deployment environment may impact the actual dependencies a given deployed module has. Pigazzini et al. [24] reconstructed the architecture of microservices-based systems parsing Java source files and Docker/Spring configuration files, with the goal of identifying cyclic dependencies between microservices. However, related works mainly focused on the identification of the anti-patterns [25], [26] proposing a visualization for the system architecture. Rahman et al. [27] followed a similar approach to parse the code, nevertheless, they developed a tool named "MicroDepGraph"[3] to visualize the call-graph between microservices. Ibrahim et al. used a project's Dockerfiles to search for known security vulnerabilities of the container images being used, which they overlay on the system topology extracted from Docker Compose files to generate an attack graph showing how a security breach could be propagated through a microservice mesh [28]. In preliminary work [6] we used the approach proposed by Rademacher et al. [12] and demonstrated that automated merge of microservices data models or detection of inter-service communication is feasible. In our recent work, we highlighted the power of static and dynamic analysis for detecting microservices API Patterns [29].

B. Architecture Visualization

Zhou et al. [30] sought common visualizations for enterprise architectures. The most common directions are ArchiMate, UML, Business Motivation Model (BMM), and BPMN, among others. The Open Group Architecture Framework (TOGAF) is the most frequently used framework for enterprise architecture, further extended by the Architectural Development Method using ArchiMate. It is typically modeled at four levels with different specializations: Business, Application, Data, and Technology, which to some extent correspond to the architectural views described by Rademacher et al. [12], with the exception of business architecture levels which rather drives the motivation for the implementation.

The C4 model (Context, Containers, Components, and Code) is a practical approach for modeling software architecture [31] given a hierarchical model consisting of four levels of abstraction, ranging from the high-level system context to individual code elements. Alternative visualization practices have emerged for software architectures [32], [33]. Shahin et al. [32] categorize alternative visualization as graph-based visualization involving graphs showing nodes and links similar to ontologies. Another approach is a notation-based visualization, such as UML or SysML, or Matrix-based approaches. Quite common is a metaphor-based visualization that uses familiar physical world contexts (e.g., cities, islands, or landscapes). To make the system more understandable using a visual metaphor, Virtual and Augmented Reality (VR/AR) methods have been explored for software architecture visualization. One example is a "software city"; software packages are represented as buildings and their dependencies as streets, which is an example of virtual reality [34]–[36]. Schreiber et al. proposed to show individual software modules as "islands" in an ocean displayed in AR. Software packages and classes in each module are represented as regions and buildings on the module island, and, importantly, module imports and exports are displayed as ports that connect the different islands. The VR-EA tool from Oberhauser et al. [37] is an attempt to visualize larger enterprise applications. However, it uses modeling tools as inputs to generate a 3D VR view in the virtual reality of business processes and their relationships with enterprise resources. While it can show a large group of interconnected components, it depends on a set of models that must be manually created.

III. Static analysis-based SAR of Microservices

Microservice systems are by nature decentralized. The Heroku's 12-factor app methodology [3], [24], [38] recommends that each microservice be self-contained with its own codebase and database to facilitate and improve evolution, scalability, and dependency management. However, microservices are not isolated; they interact using interfaces or message queues. Thus, there is a dependency between microservices; however, it is typically a loose one. Since the interaction happens through interfaces, perhaps the most notable dependency is on the endpoint names and the parameters that represent data or transfer objects. Based on domain-driven development [12], [39], each module considers a bounded context [3], which includes a limited perspective of the system holistic data model called context map, and often bounded contexts partially overlap through certain data entities with other modules. We can use this overlap as an ingredient to determine the system-centric view. Apart from this, the inter-service interaction, such as REST/RPC endpoint calls, is another ingredient we can consider. These two strategies are illustrated in Fig. 2.

Our SAR process considers the static-code analysis of individual microservice's codebases. As suggested by Carnell et al. [3], the codebase contains the source code and build and deployment configuration files possibly relevant to the process.

Our process is illustrated in Fig. 1, it starts with the *extraction* phase, operating with Abstract Syntax Trees (AST) parsed from the code. Next, walking through the tree and

[3]MicroDepGraph https://github.com/clowee/MicroDepGraph

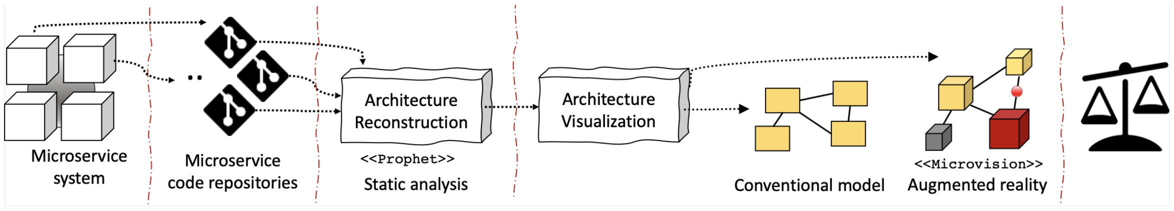


Fig. 1: Microvision Construction Process

recognizing method calls we extract call-graphs. Top methods are candidates for endpoints; in addition, frameworks typically augment these endpoints with additional information (i.e., HTTP types, constraints, etc.) that indicate the endpoint (i.e., in the form of annotations or external files). With endpoints detected, we assess their parameters and trace the calls down through the controllers, services, and repositories to referenced and involved data entities. We detect these components' types by assessing the associated properties in the AST and the call-graph from the endpoint. This tracing also allows us to derive dependencies between endpoints and involved data entities. We determine which endpoints operate with specific data entities in the reverse perspective. Walking through the call-graphs, we can detect involved constraints, apparent policies, conditions, branches, and loops. Specific attention is placed on data entities. We assess their properties and methods to detect relationships across the entities in a given microservice codebase and extract the underlying data model.

Next, we continue with the *construction* phase and convert the microservice-specific information into a graph format. This phase operates with components identified when traversing call-graphs. We further augment recognized components with additional information that might co-exist at the component definition level. For instance, REST controller endpoints might enforce access rights [7], [40]. Paying attention to components corresponds to the microservice development practice [3]. In other words, the microservice will always process data and provide endpoints. Therefore, aggregating components and combining the call paths represents a graph, which we use as an intermediate representation of the processed microservice.

The *manipulation* phase deals with the fusion of multiple microservice intermediate representations. As depicted in the previous discussion, we use two main ingredients: combining overlapping data entities and inter-microservice endpoint interactions. This is highlighted on Fig. 2. However, these strategies can be further extended, for instance, by information

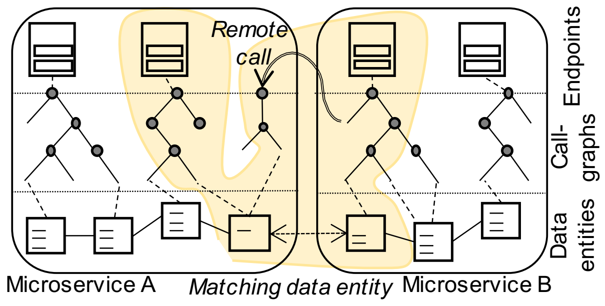


Fig. 2: Microservice dependencies

from build and deployment scripts. Moreover, the event-based approach with message brokers, such as Kafka or Messaging Queues, could be integrated here.

We begin the fusion by entity matching, specifically by looking for entities from distinct modules with a subset match of properties, data types, and possibly names. For this matching, we considered natural language processing strategies (Wu-Palmer algorithm [41]). Then, combining all involved microservices, we derive the canonical data model (context map), and, through the matched entities, we promote data and control dependencies.

The second ingredient considers inter-service interaction. First, we identify all endpoints, parameter types, and metadata, and then the remote procedure calls within the methods [7]. Next, we match them, generate a complete system service overview, and augment the canonical model resulting from the previous strategy.

Our manipulation phase results in a combined intermediate graph representation for the holistic system. This broadens the perspective for the consequent analysis with access to the canonical data model, inter-service dependencies, and the overall system service endpoints. It also maintains the specifics of each microservice, such as its bounded context with overlaps, technology information, and aggregate list of the technologies, broken up by layer, in the centralized perspective. Finally, since each microservice contains the build, deployment, and operation information, it allows the centralized perspective to render a graph of connected deployments.

The final *analysis* phase of the SAR process is reasoning about the whole system. This article limits the discussion to our new overall architectural visualization process, described next.

To perform SAR for microservices, we implemented the Prophet tool[4]. It follows the steps detailed in this section and performs static code analysis of Java-based source code. It recognizes component-based constructs behind Spring Boot and Enterprise Java. It utilizes the Java Parser library [5] and a graph database (Neo4j)[6] to store the microservice and system holistic results.

The result is an intermediate graph representation of the system accessible through REST API. This representation can be used for system reasoning. In general, any kind of reasoning could operate with the intermediate system representation. For

[4]The code for the Prophet utility can be found at GitHub https://github.com/cloudhubs/prophet-utils,https://github.com/cloudhubs/prophet-utils-app,https://github.com/cloudhubs/prophet

[5]https://javaparser.org

[6]https://neo4j.com

instance, we could detect design smells or security policy violations [7], [42]. However, overall system visualization is best suited to facilitate reporting, facilitate navigation, and improve comprehension.

The microservice codebase has the most up-to-date system details. With an intermediate graph representation of the system received from static analysis applied across the decentralized codebases (i.e., through the continuous integration pipeline), any update can be reflected in a reconstructed system-centric view.

IV. Visualization of Microservices

Many means can be used to articulate the reconstructed system architecture to stakeholders such as architects, developers, or DevOps. However, appropriate visualization can speed up comprehension of the reconstructed system and lead to expedited assessments of dependencies, bottlenecks, architectural smells [42], [43] (i.e., poor design choices and anti-patterns), or consistency errors.

This article considers two approaches: a conventional architectural visualization and a 3D visualization. We will assess each approach based on the following capabilities: *Visualization ability*, *Comprehensibility*, *Navigation*, and *Interaction of services*.

The SAR process may result in multiple architectural views. However, it is sufficient to limit our attention to a few views for a proof of concept. Thus, we pick the most beneficial views for microservices, those that support a system-centric perspective.

Mayer and Weinreich [44] identified that supporting a view of service APIs and their interactions should be one of the most important goals that a tool designed for microservice analysis should achieve. When we consider Rademacher et al. [12], focusing on the service view is well justified. The service view defines the microservice's APIs and the inter-service calls between them. Furthermore, this view is also relevant for developers seeking to understand how the system operates.

Rademacher et al. [12] also focused on the domain view. This view defines the domain model used by the microservice system, also known as the canonical model or context map. This view is necessary because microservices do not depend on a formal specification of a domain model, with each service instead of operating on its own bounded context, where it operates on the subset of entity attributes it needs [45].

Both service and domain views give a view of the system architecture as-is, showing the communication between the services and the state of the domain entities in use. These views can be used as documentation for developers and DevOps. Architects can compare the current architecture against the planned system architecture and detect deviations. They can also use it as a first warning to detect if the architecture has drifted from the original plan.

A. *On service and domain view information*

To extract necessary system information to construct the service view, we require two things: first, to detect the endpoints of each service, and second, to detect the calls made from one service to another using these endpoints. Software frameworks often provide utilities for quickly defining these endpoints in code, and this has been utilized by projects like Swagger[7] for automated endpoint documentation. After endpoints are identified, Prophet inspects the microservice Abstract Syntax Trees (ASTs) for remote method calls. These can be recognized through common constructs such as REST templates, etc. Once the list of endpoints and calls is collected for each service, Prophet matches the calls to system endpoints based on the relative endpoint URL, the HTTP method, and parameters. The result is a call-graph representing the system, showing how the services communicate among themselves.

To extract system information to determine the domain view, we need to identify data entities, their properties, and their relationships. Data entities use frameworks utilities and can be identified similarly to endpoints. Having entities identified we can consider their properties and relevant data types. Identified property data types can reveal relationships the entities have with each other. These relationships have three different components, which we extract using code analysis: the types involved in the relationship (i.e., the entities that are on either side of the relationship), the multiplicity of the relationship, and the directionality of the relationship. Identifying the types is done based on the type names of the entities' fields, the multiplicity can be determined by whether or not the field is a collection, and its directionality can be determined by whether or not there is a corresponding field in both of the entities involved or in only one entity. Considering a single microservice codebase, we can derive a microservice bounded context.

Using the bounded contexts for all microservices, a combined canonical model for the entire system can be generated by merging the bounded contexts. Since the services should be operating on some of the same entities, the entities in each microservice can be merged by detecting if they have the same or similar names. Different services may have different purposes for the entities they share and so may retain different fields from each other. Fields with the same or similar names and the same data type are merged into a single field in the merged entity, while non-matching fields from all the source entities can simply be appended to the merged entity. The result represents the scope of all entities used in the system.

B. *Considered system samples*

To demonstrate visualization approaches for this manuscript and on a large, realistic system, we adopted two test benches. The Teacher Management System (TMS)[8] consists of three microservices, and the limited size allows us to embed complete SAR visualization examples in this article. For a demonstration of a complex case study, TrainTicket[9] [46] is used (originating from the ICSE conference). The TrainTicket was designed to emulate a real-world microservice system consisting of 41 microservices and over 60,000 lines of code.

[7]https://swagger.io

[8]https://github.com/cloudhubs/tms2020

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

It is written in Spring Boot, uses MongoDB as its database, and follows cloud-native practice with containers, routing, etc.

C. Conventional architectural visualization and its properties

The conventional approach to visualizing service and domain views operates in two-dimensional space. The service view represents microservices as nodes and particular service calls as edges. An example output of the result of this analysis on the TMS testbench is shown in Fig. 3.

The domain view has a perfect fit for the UML class diagram that represents the scope of all entities used in the system, as shown in Fig. 4 for the TMS system.

These results on the TMS system demonstrate a system-centric perspective extracted from the microservice codebase. Since this article focuses on visualization aspects, we next consider deficiencies and limits of obtained results.

The biggest shortcoming of the conventional two-dimensional graph representation is its visualization ability; it quickly runs into scaling problems. We discovered that the visualization breaks down when analyzing systems larger than a few microservices. A two-dimensional space only has so much area available to display a graph, which fills up quickly and becomes unintelligible. This limitation is not surprising; as the number of services in a system increases, the potential number of connections between them increases at a much faster rate. There is only so much space in a two-dimensional layout to arrange these connections, and thus the visualization becomes cluttered and unwieldy. We discovered this problem when analyzing larger systems; Fig. 5 shows service view output on the TrainTicket testbench (41 microservices), which becomes difficult to understand.

A problem of visualization may seem like a minor one, but it directly affects the view's intended purpose as an artifact to help a stakeholder quickly understand how microservices interact with each other in a large system and to allow them to visually identify potential problems with the architecture or to identify drift from the originally-intended architecture. As the graphs become cluttered, this kind of quick visual analysis becomes less feasible, as it takes more time to understand what the graph is displaying. A visualization solution based on two-dimensional diagrams simply does not scale well with the number of microservices in a system.

The related problem is that of navigating the displayed graphs. While a small system can be displayed on a single page without much issue, the output requires users to navigate larger graphs using the mouse scroll wheel and does not provide for zooming in or out, nor any other method of viewing multiple levels of abstraction, an important feature of microservice architectural analysis as seen in, e.g., the hierarchical C4 model [31]. This limited method of navigation creates a problem since there is no way to step back and get a broad view of the system, nor can the user quickly drill into a specific region of the microservice mesh. It can take time and effort to find the area of interest in the displayed graph, and it may not be as insightful if developers cannot easily relate what they are looking at to the rest of the system. Again, this directly impedes the original goal of the quick and intuitive analysis.

Another problem is that the information about each microservice's API is not easily accessible. The endpoints are only displayed on the edges that point to the node. The user must mentally reconstruct what the API looks like for a particular service by finding all of the incoming edges and identifying their labels. This is extra work for the user, which is also detrimental to the goal of quick visualization, and the difficulties with navigation as previously mentioned compound the task.

The final problem with the conventional visualization is its inability to display how the microservices interact with each other when servicing actual requests from users. Its visualization is completely static; the connections between services are there, but there is no information on how those connections are utilized. This also hinders the goal of providing at-a-glance visualization of a system; the static view of the connections provides only a partial picture.

To summarize, we identified these challenges:

1) Visualization ability: the method needs to scale better with system size than a two-dimensional, UML-based solution.
2) Comprehensibility: developers should be able to quickly comprehend the interaction of microservices in a system.
3) Navigation: the visualization should be easily navigable and enable traversing multiple levels of abstraction.
4) Interaction of services: a method is needed that can visualize how the services operate and interact with each other, beyond what is capable with UML-based diagrams (e.g., sequence diagrams).

D. A Microvision

The most applicable view for understanding the system-centric perspective and the system operation is the service view. With the 2D limitation in mind, we have adopted this view to explore the benefits of a three-dimensional visualization scheme. We utilize the AR medium, which is natively three-dimensional, and it lends itself to control schemes based on natural movement. We believe this combination holds potential for use with displaying and navigating complex systems such as microservices. The way we approach the visualization is by using a 3D graph operating in AR. Given we were able to automate the SAR process and reconstruct the service view in 2D, we use the same input for a 3D graph operating in AR.

The potential of using AR for software visualization has been recognized, and it has been used to visualize monolithic software systems in various ways as shown above in the Related Work section. However, it has not been previously applied to microservice systems. We aim to expand the existing 3D visualization techniques to a higher level of abstraction beyond a single piece of software to an entire distributed microservice system.

1) Designing 3D Visualization: For the *system-centric perspective*, the view needs to provide a *high-level system* visualization. In microservices, we intend to see their *interconnec-*

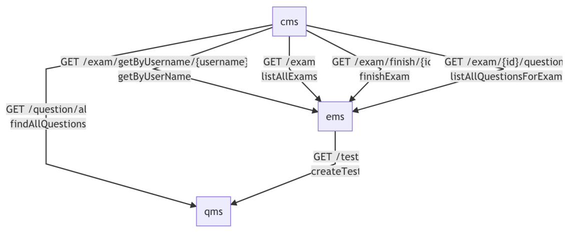


Fig. 3: Sample service view extracted from a the TMS benchmark.

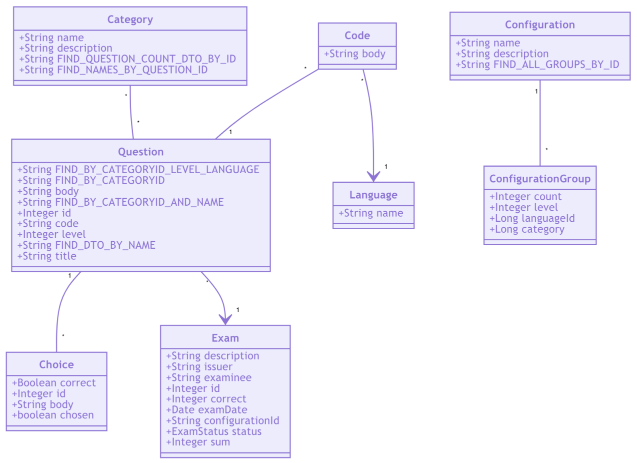


Fig. 4: Domain view derived from the TMS benchmark. These entities are aggregate definitions from partial entities in each microservice's bounded context.

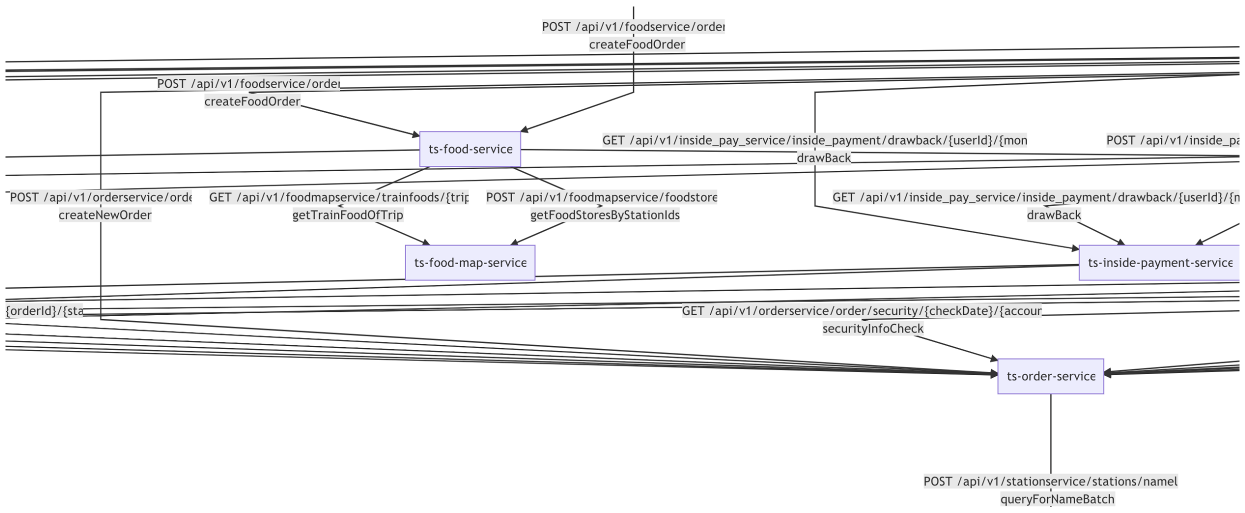

POST /api/v1/foodservice/order
createFoodOrder
POST /api/v1/foodservice/order
createFoodOrder
ts-food-service
GET /api/v1/inside_pay_service/inside_payment/drawback/{userId}/{mon
drawBack
POST /api/v1/inside_pa
POST /api/v1/orderservice/orde
createNewOrder
GET /api/v1/foodmapservice/trainfoods/{trip
getTrainFoodOfTrip
POST /api/v1/foodmapservice/foodstore
getFoodStoresByStationIds
GET /api/v1/inside_pay_service/inside_payment/drawback/{userId}/{m
drawBack
ts-food-map-service
ts-inside-payment-service
{orderId}/{sta
GET /api/v1/orderservice/order/security/{checkDate}/{accour
securityInfoCheck
ts-order-service
POST /api/v1/stationservice/stations/namel
queryForNameBatch

Fig. 5: The service view from a large microservice testbench TrainTicket [46]. Connections between services become difficult to decipher as the system size grows.

tion; however, the view cannot get cluttered as more services are introduced. Quickly understanding the high-level structure should be prioritized in all system-centric perspectives.

In addition, the user should easily *interacts* with the view and *navigates* through the microservice system both at a high level and a lower level of detail centered around a few services. The high-level view should be easily understandable, and upon drilling down into a lower-level view, the user should be able to identify details about individual services and how they relate to their neighbors.

In this case, the high-level view refers to the *overall structure of the system*, and the low-level view refers to information about *individual services* and their immediate neighbors. The less time it takes to go from high to low level of detail, the easier it is to understand the system and the roles the individual services play in it.

Given our ambitious AR microservice visualization plan, we have developed a Microvision proof-of-concept tool delivering the service view. Microvision consists of two broad components to achieve its functionality: the main *graph display, and the API viewer*. The following section describes the design of these components and details the rationale behind the components.

Graph display: The overall display of the microservice system shows an abstract 3D graph view of the system projected in AR. The goal of the base graph is to give a quick view of the system, its services, and its connections. Each microservice is represented as a node, and an edge exists between nodes if a call exists between two microservices. Fig. 6 shows our implementation example. The nodes are distributed such that there is no overcrowding in one particular area of the graph. To help visualize how the connections work, a node can be selected to highlight it and its neighbors. In this case, the neighbors highlighted are only those that are called at some point from the selected node. This helps clarify the node's role among nearby nodes.

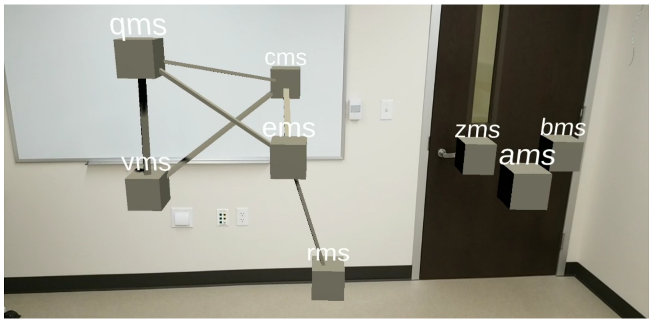


Fig. 6: Showing 3D graph of services and their connections.

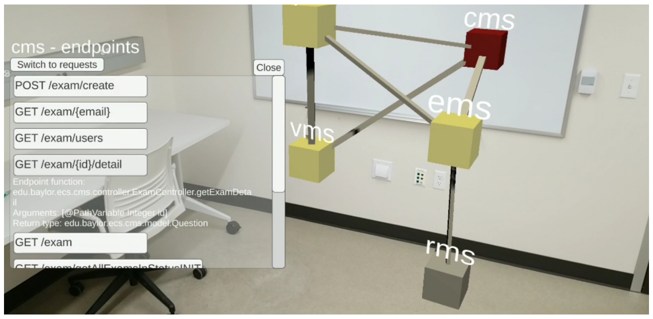
The rationale for choosing a graph display is simple. A graph is the most natural way of conceptualizing how microservices work, and two-dimensional graphs are consistently encountered in other microservice analysis tools.

The three-dimensional display addresses both of these drawbacks. First, adding the third dimension increases the available area to display the services and reduces the amount of overlap their connections have. This decluttering also makes it more straightforward to analyze the architecture and identify potential architectural problems or areas of drift from the original architecture. Second, displaying the graph in an AR environment allows us to implement a natural navigation scheme: the user simply moves their device through the graph. Since a 3D graph has an innate spatial logic to it, it is intuitive to navigate by natural motion, even for a large graph. This natural movement also allows for quickly switching between a broad overall view of the system or a closer look at a particular group of services simply by moving closer or farther.

The other alternative considered is to display the content of the system using a visual metaphor, such as "software cities" or similar approaches encountered in prior work [34], [35], [47]. We ultimately rejected this approach in favor of an abstract approach, since learning a new visual metaphor would take extra time and training, and the idea of how microservices connect is served well by an abstract graph representation. Furthermore, these visual metaphors usually restrict the space in which the services can be displayed; for example, the software city metaphor requires a two-dimensional layout of the system in question, with the vertical dimension being used to display information about the content of the software packages. For a large system of microservices, this space would be better utilized by simply displaying more services in a smaller area.

API view: The API view component is responsible for displaying the endpoints that make up the API of a selected microservice. The design goals for this component were to concisely display the relevant API for a particular microservice without cluttering the overall graph. For this component, a simple pop-up box that contains the list of endpoints was chosen. Fig. 7 shows our implementation example.

Upon selecting a microservice in the graph, the API view box will pop up and display the list of endpoints. Each endpoint is identified by the endpoint path and HTTP method. The endpoints can be expanded by tapping them, providing



POST /exam/create
GET /exam/{email}
GET /exam/users
GET /exam/{id}/detail
Endpoint function:
edu.baylor.ecs.cms.controller.ExamController.getExamDeta
il
Arguments: [@PathVariable Integer id]
Return type: edu.baylor.ecs.cms.model.Question
GET /exam

Fig. 7: The context menu shows a selected services API endpoints, in this case the "cms" service highlighted in red.

further information such as the actual method that implements the endpoint and its parameters and return type.

This component is intended to provide details for developers to drill down into. The graph itself provides both a broad overview and details of the services through a contextual window panel. We chose to show specifically the API because that is a microservice's best indicator of its role within the system. It is not by default shown within the graph itself, as that would add too much information that may not be immediately relevant to the user.

The primary alternative considered for this component is to annotate the node in the graph itself with the information. This way, the information could be positioned spatially in relation to the nodes it applies. This alternative was rejected because it would clutter the graph unnecessarily. The graph already contains the nodes, their connections, and the names of the microservices annotated onto the nodes. Adding extra text to the graph would make the existing elements more difficult to read and interpret. Furthermore, the convention of tapping an element to display an informational pop-up already exists, and since we highlight the selected node, we do not lose any clarity.

2) Microvision Approach Summary: Our proof-of-concept, Microvision[10], addresses the shortcomings of 2D visualization. We demonstrated 3D visualization based on AR to reduce the cluttered nature of the conventional visualization for microservices. In addition, we demonstrated navigation and control through the reconstructed microservice system architecture. Specifically, we addressed the challenges:

1) Visualization ability: we have developed a 3D visualization that offers better scaling with the number of services than a 2D diagram.
2) Comprehensibility: the 3D structure can be viewed for a high-level system overview.
3) Navigation: the graph is displayed in AR and is easily traversed by natural movement. Multiple levels of abstraction are viewed naturally within the graph itself due to the 3D overview.

Still, the interaction perspective can be further developed with simulated endpoint interaction to better address dependencies and tracing.

[10]Its code is available at GitHub https://github.com/cloudhubs/microvision

V. Small Evaluation Study

A small-scale pilot study in terms of the number of participants and the number of microservices is included. It was conducted to evaluate the feasibility of the Microvision approach by answering practical developer questions and analyzing the architecture of a microservice system. We conducted the trial user study with graduate student volunteers performing various analysis tasks on a real-world microservice system using Microvision and giving feedback on their experience.

Our evaluation was conducted with a group of six graduate computer science students (five males, one female). All of the participants had experience with software development, and four participants had prior experience with microservices.

The participants were given a system consisting of 16 microservices to analyze. The system in question was a TrainTicket testbench [46] handling train ticket reservations. We prepared a set of nine evaluation tasks relating to this system for the participants to complete. Three questions were prepared relating to individual services and their connections in the system, and six were prepared relating to user requests and how the system handled them. The tasks required the participants to use Microvision to identify different aspects of the microservice APIs and their connections to each other. The tasks are given in Table I.

General questions
Which service has the most connections?
Of those connections, how many calls from that service to another?
How many services have only a single connection?
Request #1
How many services are involved in this request?
What is the last call in the call chain? Include the service and endpoint.
Suppose the ts-ticketinfo-service changes the arguments required for its controller method, queryForStationId. Will this change affect this request?
Request #2
What data type is the argument passed to the second endpoint?
What is the controller method that handles the initial request?
Suppose the ts-travel-plan service could be made to skip the route-plan-service and call the ts-travel-service directly. Would this change make the ts-route-plan-service obsolete in the system?

TABLE I: Tasks completed by participants in the evaluation.

We also prepared a 5-question satisfaction survey regarding participant experience with the application. The first three questions were given on a 5-point Likert scale and measured the participants' satisfaction with the use of Microvision on the specified tasks. The next two questions asked about the participants' perceived usefulness of Microvision's features and asked for suggestions for new features. The feedback survey and the participant responses are given in Table II.

The evaluation was split into three segments. In the first segment, the participant was briefed on the features of Microvision and given directions on how to operate it. This segment lasted a maximum of ten minutes. In the second segment, the participants were given 15 minutes to complete the evaluation tasks, beginning with the three tasks relating to individual services, followed by the six tasks relating to user requests. The tasks were given one at a time, with immediate feedback as to whether the participant answered correctly or not. In the final segment, the participants were given the feedback survey, which they could complete in any amount of time they chose.

Satisfaction feedback (5-point Likert scale)	
Was Microvision helpful to you when completing the tasks?	4 strongly agree, 2 agree
Was Microvision intuitive to use?	4 strongly agree, 2 agree
Given the option, would you use Microvision again?	4 strongly agree, 2 agree
General feedback	
Which features, if any, did you find helpful or useful when using Microvision?	3D graph visualization: 6/6; API viewer: 5/6
Do you have any other comments or suggestions regarding Microvision?	Graph scaling and individual node movement suggested

TABLE II: Feedback questions posed to participants.

1) Evaluation Results: All six of the participants completed the evaluation tasks with 100% accuracy within the allotted time frame. Furthermore, all the participants either agreed or strongly agreed to the first three qualitative feedback questions. Regarding the features, all participants said the 3D graph visualization was useful, and five out of the six participants said the API viewer was useful. These results show that Microvision is a promising direction to further research for microservice system analysis.

VI. Conclusions

This research was motivated by recurrent microservice system challenges regarding missing system-centric views. It analyzed cloud-native systems using static analysis to demonstrate it is feasible to derive a system-centric perspective of these systems. It also elaborated on alternative visualization directions using three-dimensional space to render the system's service view in AR. We implemented a Microvision tool that uses the intermediate representation of the system built by the static analysis tool Prophet, which is capable of multi-codebase analysis for microservices. We assessed the approach on two system testbench with a short evaluation study to better understand the practical implications and potential impacts of such a visualization. In future work, we plan to perform a large user study for which we have already obtained IRB approval. The SAR process will also be generalized to a platform-agnostic approach as initiated in [48]. We will also assess more architectural views and alternative 3D models.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 1854049, grant from Red Hat Research, and Ulla Tuominen (Shapit).

References

[1] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, "Industry practices and challenges for the evolvability assurance of microservices," *Empirical Software Engineering*, vol. 26, no. 5, p. 104, 2021.

[2] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, "The pains and gains of microservices: A systematic grey literature review," *Journal of Systems and Software*, vol. 146, pp. 215–232, 2018.

[3] J. Carnell and I. H. Sánchez, *Spring microservices in action*. Manning Publications Co., 2021.

[4] T. Cerny and D. Taibi, "Static analysis tools in the era of cloud-native systems," in *4th International Conference on Microservices*, 2022.

[5] A. Walker, I. Laird, and T. Cerny, "On automatic software architecture reconstruction of microservice applications," *Information Science and Applications: Proceedings of ICISA 2020*, vol. 739, p. 223, 2021.

[6] V. Bushong., D. Das., and T. Cerny., "Reconstructing the holistic architecture of microservice systems using static analysis," in *Int. Conference on Cloud Computing and Services Science -*, 2022, pp. 149–157.

[7] D. Das, A. Walker, V. Bushong, J. Svacina, T. Cerny, and V. Matyas, "On automated rbac assessment by constructing a centralized perspective for microservice mesh," *PeerJ Computer Science*, vol. 7, p. e376, 2021.

[8] T. Cerny, A. Abdelfattah, V. Bushong, A. A. Maruf, and D. Taibi, "Microservice architecture reconstruction and visualization techniques: A review," in *2022 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, 2022.

[9] L. O'Brien, C. Stoermer, and C. Verhoef, "Software architecture reconstruction: Practice needs and current approaches," Carnegie Mellon University, Tech. Rep., 01 2002.

[10] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, p. 40–52, 1992.

[11] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012.

[12] F. Rademacher, S. Sachweh, and A. Zündorf, "A modeling method for systematic architecture reconstruction of microservice-based software systems," in *Enterprise, Business-Process and Information Systems Modeling*. Springer International Publishing, 2020, pp. 311–326.

[13] A. Al Maruf, A. Bakhtin, T. Cerny, and D. Taibi, "Using microservice telemetry data for system dynamic analysis," in *2022 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, 2022.

[14] B. Mayer and R. Weinreich, "An approach to extract the architecture of microservice-based software systems," in *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, 2018, pp. 21–30.

[15] K. A. Torkura, M. I. Sukmana, and C. Meinel, "Integrating continuous security assessments in microservices and cloud native applications," in *Int. Conf. on Utility and Cloud Computing*, 2017, pp. 171–180.

[16] G. Granchelli, M. Cardarelli, P. D. Francesco, I. Malavolta, L. Iovino, and A. D. Salle, "Towards recovering the software architecture of microservice-based systems," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 46–53.

[17] A. Chlipala, "The bedrock structured programming system: Combining generative metaprogramming and hoare logic in an extensible program verifier," *SIGPLAN Not.*, vol. 48, no. 9, pp. 391–402, Sep. 2013.

[18] E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla, "Verification of java bytecode using analysis and transformation of logic programs," in *Practical Aspects of Declarative Languages*, 2007, pp. 124–139.

[19] X. Li, Y. Chen, and Z. Lin, "Towards automated inter-service authorization for microservice applications," in *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, 2019, pp. 3–5.

[20] S. Eski and F. Buzluca, "An automatic extraction approach: Transition to microservices architecture from monolithic application," in *International Conference on Agile Software Development: Companion*, 2018.

[21] D. Taibi and K. Systä, "From monolithic systems to microservices: A decomposition framework based on process mining," in *Proceedings of the 9th International Conference on Cloud Computing and Services Science*, 2019, pp. 153–164.

[22] U. Azadi, F. A. Fontana, and D. Taibi, "Architectural smells detected by tools: A catalogue proposal," in *Proceedings of the Scientific Workshop Proceedings of XP2016*, ser. XP '16 Workshops, 2019.

[23] S. Esparrachiari, T. Reilly, and A. Rentz, "Tracking and controlling microservice dependencies," *ACM Queue*, vol. 16, pp. 44–65, 2018.

[24] I. Pigazzini, F. A. Fontana, V. Lenarduzzi, and D. Taibi, "Towards microservice smells detection," in *Proceedings of the 3rd International Conference on Technical Debt*, ser. TechDebt '20, 2020, p. 92–97.

[25] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE Software*, vol. 35, no. 3, pp. 56–62, 2018.

[26] D. Taibi, V. Lenarduzzi, and C. Pahl, *Microservices Anti-patterns: A Taxonomy*. Springer International Publishing, 2020, pp. 111–128.

[27] M. Rahman and D. Taibi, "A curated dataset of microservices-based systems," in *Joint Proceedings of the Summer School on Software Maintenance and Evolution*. CEUR-WS, September 2019.

[28] A. Ibrahim, S. Bozhinoski, and A. Pretschner, "Attack graph generation for microservice architecture," in *ACM/SIGAPP Symposium on Applied Computing*, 2019, pp. 1235–1242.

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

[30] Z. Zhou, Q. Zhi, S. Morisaki, and S. Yamamoto, "A systematic literature review on enterprise architecture visualization methodologies," *IEEE Access*, vol. 8, pp. 96 404–96 427, 2020.

[31] A. Vázquez-Ingelmo, A. García-Holgado, and F. J. García-Peñalvo, "C4 model in a software engineering subject to ease the comprehension of uml and the software," in *2020 IEEE Global Engineering Education Conference (EDUCON)*, 2020, pp. 919–924.

[32] M. Shahin, P. Liang, and M. A. Babar, "A systematic review of software architecture visualization techniques," *J. Syst. Softw.*, vol. 94, pp. 161–185, 2014.

[33] R. Wettel and M. Lanza, "Visually localizing design problems with disharmony maps," in *ACM Symposium on Software Visualization*, ser. SoftVis '08, 2008, p. 155–164.

[34] 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.

[35] M. Steinbeck, R. Koschke, and M. O. Rüdel, "How evostreets are observed in three-dimensional and virtual reality environments," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 332–343.

[36] Z. Ma and Y. Bai, "A distributed system monitoring tool with virtual reality," in *International Conference on Computer Science and Application Engineering*, ser. CSAE '18, 2018.

[37] R. Oberhauser and C. Pogolski, "VR-EA: Virtual Reality Visualization of Enterprise Architecture Models with ArchiMate and BPMN," in *Business Modeling and Software Design*, 2019, pp. 170–187.

[38] A. Wiggins, "The twelve-factor app," 2017, (Accessed on 10/02/2021). [Online]. Available: https://12factor.net/

[39] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: Current and future directions," *SIGAPP Appl. Comput. Rev.*, vol. 17, no. 4, pp. 29–45, Jan. 2018.

[40] W. Hopkins, "JSR 375: JavaTM EE security API," November 2009. [Online]. Available: https://jcp.org/en/jsr/detail?id=375

[41] L. Han, A. L. Kashyap, T. Finin, J. Mayfield, and J. Weese, "UMBC_EBIQUITY-CORE: Semantic textual similarity systems," in *Second Joint Conference on Lexical and Computational Semantics (*SEM), Volume 1: Proceedings of the Main Conference and the Shared Task: Semantic Textual Similarity*. Atlanta, Georgia, USA: Association for Computational Linguistics, Jun. 2013, pp. 44–52.

[42] 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, 2020.

[43] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural patterns for microservices: A systematic mapping study," in *Proceedings of the 8th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, INSTICC. SciTePress, 2018, pp. 221–232.

[44] B. Mayer and R. Weinreich, "A dashboard for microservice monitoring and management," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 66–69.

[45] T. Černý, M. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: current and future directions," *ACM SIGAPP Applied Computing Review*, vol. 17, pp. 29–45, 01 2018.

[46] 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*, 2018.

[47] A. Schreiber, L. Nafeie, A. Baranowski, P. Seipel, and M. Misiak, "Visualization of software architectures in virtual reality and augmented reality," in *2019 IEEE Aerospace Conference*, 2019, pp. 1–12.

[48] M. Schiewe, J. B. Curtis, V. Bushong, and T. Cerny, "Advancing static code analysis with language-agnostic component identification," *IEEE Access*, 2022.