Cluster Computing
https://doi.org/10.1007/s10586-024-04394-7

From static code analysis to visual models of microservice architecture

Tomas Cerny[1] · Amr S. Abdelfattah[2] · Jorge Yero[2] · Davide Taibi[3]

Received: 10 October 2023 / Revised: 16 January 2024 / Accepted: 26 February 2024

Abstract
Microservice architecture is the mainstream driver for cloud-native systems. It brings various benefits to the development process, such as enabling decentralized development and evolution of self-contained system parts, facilitating their selective scalability. However, new challenges emerge in such systems as the system-holistic quality assurance becomes difficult. It becomes hard to maintain the desired system architecture since many teams are involved in the development process and have greater autonomy. Without instruments and practices to coordinate teams and assess the system as a whole, the system is prone to architectural degradation. To face such challenges, various architectural aspects of the system should be accessible to practitioners. It would give them a better understanding of interconnections and dependencies among the microservice they manage and the context of the entire system. This manuscript provides the perspective on uncovering selected system architectural views using static code analysis. It demonstrates that holistic architectural views can be effectively derived from the system codebase(s), highlighting dependencies across microservices. Such new perspectives will aid practitioners in making informed decisions when intending to change and evolve the system. Moreover, with such a new instrument for system holistic assessment, we quickly realize that human experts must cope with another problem, the evergrowing scales of cloud-native systems. To elaborate on the topic, this manuscript examines how static analysis outcomes can be transformed into interactive architectural visualizations to assist practitioners in handling large-scale complexities.

Keywords Microservices · Software architecture reconstruction · Interactive visualization · Augmented reality · System-centric view · Decentralized systems

1 Introduction

Modern software systems that require scalability or aim at servicing a broad audience likely target cloud-native capabilities and thus utilize microservice architecture [1]. While existing infrastructure and development frameworks greatly simplify the design and development of such systems, there is a lack of instruments to analyze the system [2] to make informed decisions for the evolution of such systems or help developers understand the big picture of the overall system.

Architects typically prescribe a system design blueprint that is distributed across various development teams that implement microservices individually and let them interplay over closely coupled endpoints or messaging. The development teams are typically assigned to manage a few microservices exclusively [3] and establish an in-depth knowledge of these; however, they are unaware of other microservice details beyond recognized endpoints or

✉ Tomas Cerny
tcerny@arizona.edu

Amr S. Abdelfattah
amr_elsayed1@baylor.edu

Jorge Yero
jorge_yero1@baylor.edu

Davide Taibi
davide.taibi@oulu.fi

1 Systems and Indistrial Engineering, University of Arizona, 1127 East James E Rogers Way #111, Tucson, AZ 85721, USA

2 Computer Science, Baylor University, One Bear Pl, Waco, TX 76798, USA

3 University of Oulu, Pentti Kaiteran katu 1, 90570 Oulu, Finland

events and likely do not see implicit dependencies across microservices [4, 5].

Throughout the maintenance and evolution, there is no holistic system view [6, 7], which would be used by developers to understand the interconnections, similarities, and dependencies across microservices in the system. We would need to involve multiple teams to reconstruct a holistic system perspective to enable a broader system analysis. Moreover, since individual microservices change often, such a process would need to be performed periodically, which becomes expensive to perform manually. Without a system-centered perspective, there is no straightforward mechanism to verify if the prescribed architectural properties hold in the implemented system. The ability to derive a system-centered perspective of the currently developed system version would serve as an important instrument to facilitate quality assurance throughout system evolution.

Furthermore, in common market settings prioritizing new system feature development over quality maintenance, uninformed design decisions might greatly deteriorate system design. Indeed, new features are visible to customers, but quality maintenance is not something they value as much. Without a system-centered view to observe changes performed by individual teams, architects will easily lose track and let the system quality go downhill towards architectural degradation.

In previous works [8–13], we investigated inter-service dependencies, architectural properties, and the big picture details of systems based on microservices. Such information detail helps practitioners understand the system as a whole since they usually manage a particular system part (i.e., selected microservice). We investigated static analysis to reconstruct such information from systems codebase(s). Such a process can quickly reconstruct up-to-date documentation about the system. With advancements in this challenge, a visual perspective is necessary to engage practitioners. However, we quickly realized that to effectively present extracted information to practitioners, established visualization approaches do not fit the needs of cloud-native systems. Visual models for architectural views need to cope with a large volume of information in these systems to render the whole system, and established models have their limits.

To detail the perspective of deriving system-centered views. This manuscript addresses the following research questions related to the depicted challenges of collecting information and presenting them:

RQ1 Is it feasible to use static code analysis of individual microservices and to combine the results to determine the holistic detail and dependencies of the system?

RQ2 Given the state-of-the-art opportunities for architecture visualization, could static analysis produce inputs to apply such visualization for microservice systems?

This manuscript brings the perspective of static analysis of microservices to reconstruct the holistic system perspective. It details the intermediate representation of the system, which results from the static analysis, and illustrates how components and high-level code constructs extracted from individual microservices codebases could combine to derive the big system picture. Using such intermediate representation, this manuscript shows how conventional visual models can be derived for system documentation. Next, it elaborates on gaps and the specific needs for such models for cloud-native systems. Consequently, it illustrates how the same intermediate representation could feed alternative visual models that use advancements in visual domains such as interactive visualization, 3D models, and augmented reality.

The main contribution of this manuscript is the overall perspective of microservice system architecture reconstruction using static analysis with consequent information visualization aiding in software architecture analysis. Such a comprehensive perspective is missing in the literature because of gaps requiring diverse expertise, such as programming languages, cloud-native systems, software architecture, and visualizations. This work provides comprehensive details about the process pipeline and shares proof-of-concept open-source tools to aid with advancements in these efforts.

With microservice system architecture reconstruction in automated means, it becomes possible to utilize and present obtained system information through various visualization approaches. While established analysis tools use visual models proposed long before microservices became the industry mainstream, one might question whether recent advancements in data visualization might provide more effective models and means to analyze and understand the software architecture of such systems. This manuscript provides a broad overview of visual models that might be relevant to the needs of cloud-native systems. Based on promising visual models, it builds various proof of concept models intended to aid with architecture analysis (i.e., understanding microservice dependencies). It demonstrates that with the Software Architecture Reconstruction (SAR) process in place, researchers and experts on visualization gain new promising instruments to focus on constructing alternative visual models and assess them on realistic systems rather than being burdened by the construction of system models.

To demonstrate the process a large third-party system testbench was used show how the process can produce conventional models, but also derive models in augmented reality or three-dimensional space. Moreover, with such results from proof-of-concept tools, we discuss the practical impacts and implications of such model transitions. With the provided tools, we invite experts in visualizations to contribute to the microservice field with novel models, allowing practitioners to perform common tasks more effectively.

The organization of this article is as follows: Sect. 2 discusses the background and related work. Section 3 outlines the software architecture reconstruction process. This is followed by details on architectural views and intended visualizations. A Case Study in Sect. 5 elaborates on the approach to extract conventional architectural visualization from a sample system. Consequently, the alternative visual models are elaborated in Sect. 6. Answers to research questions and a discussion about our proofs-of-concept can be found in Sect. 7. The discussion elaborates on the presented knowledge that advances the discipline, the implications of the study, and its limitations. Finally, Sect. 8 draws conclusions and outlines future work.

2 Background and related work

Cloud-native systems are fueled by microservice architecture and a set of principles, standards, and guidelines [14]. These systems are typically large and complex or often seen as enterprises. As these systems evolve, it is easy to lose track of the documentation and the overall system view. When we aim to understand, document, and potentially improve an existing software system's architecture, we perform SAR [15, 16]. Such a process helps us to understand the current architecture, document the system, assess quality (i.e., detect anti-patterns or identify technical debt), plan evolutions, check compliance, or understand the costs and resources needed for maintenance and upgrades.

We typically model or visualize the system architecture to understand its properties and dependencies. However, software architecture means different things to different experts. Consequently, architecture is best described by various architectural views [17]. Considering the size and complexity of cloud-native systems, conventional visual models reach their limits, and we should research alternative opportunities for large data visualization that could apply to software architecture. The following text elaborates on these core perspectives.

A visual model of software architecture is a graphical representation or diagram that illustrates the structure, components, relationships, and key aspects of a software system's architecture. Visual modeling is a way for experts and novices to have a common understanding of otherwise complicated ideas. By using visual models complex ideas are not held to human limitations, allowing for greater complexity without a loss of comprehension. These visual models are used to communicate complex architectural concepts and designs to various stakeholders, including developers, architects, project managers, and other non-technical stakeholders. Visual models provide a clear and concise way to convey the high-level and low-level aspects of a software system's architecture.

2.1 Software architecture reconstruction (SAR)

Software architecture serves as the blueprint for systems by being the central focal point of the system's development and design. These systems must often be reconstructed to determine if they were built accurately, which constitutes the essence of the SAR process.

Various works have described the SAR process. An example of it is the concept offered by O'Brien's [15], who defined architecture reconstruction as "the process by which the architecture of an implemented system is obtained from the existing system". According to these authors, the results of this process has three main utilities: (1) the evaluation of the conformance of the as-built architecture to the as-documented architecture; (2) the reconstruction of the architecture descriptions for systems that are poorly documented, or for which documentation is not available; and (3) the analysis and understanding of the architecture of existing systems to satisfy new requirements and eliminate existing software deficiencies.

The SAR methodology facilitates the extraction of a representation of software architecture from entities such as formal documentation or, predominantly, the source code and runtime traces. It aids developers in gaining a clearer insight into the system in question and also plays a key role in other tasks, such as architecture verification, conformance checking, and trade-off analysis [16]. Additionally, SAR is particularly pertinent when addressing challenges associated with the software architecture degradation that occurs when a system architecture drifts away from the originally intended architecture due to changes in the codebase.

In the context of microservices, the SAR process has profound importance. It has the potential to derive the system-centric view and illustrate how the system works [16]. It can provide broad architectural description of the system [18]. There is a significant difference between monolith and decentralized systems like cloud-native microservices. Challenges at the cloud-native level might not exist with monoliths. For each microservice exists a self-contained codebase [19]. Different microservices are typically managed by different teams. Additionally, every microservice might use different design conventions, frameworks, have different library versions, or even operate on different platforms, making the SAR more challenging for such systems.

There are three categorizations of SAR methods based on how the analysis is performed [20]: *Manual* analysis, where a human inspects the system and handcrafts its representation; *static* analysis, where the view is constructed from pre-deployment artifacts; and *dynamic* or runtime analysis, where a tool creates a view while the system is running. It is important to highlight that even if the manual analysis does not use an automated tool, it is a crucial step in suggesting a method or confirming the outcomes.

Dynamic analysis gains advantages from accessing live data, including performance indicators and ongoing service interactions. For instance, tracing [21] can be used to uncover system communication paths [14] to illustrate explicit system dependencies. However, there is one major drawback. In order for these techniques to be effective, the system must be deployed and we need comprehensive interaction with the system either through complete test coverage or involve users, which might require production deployment. Through dynamic analysis, we can identify system endpoints. But without a thorough record of system interactions, we cannot capture a full view of the system. This limitation can sometimes be underestimated, especially when juxtaposed with static analysis. Dynamic analysis for this purpose would assume a comprehensive, up-to-date testing infrastructure with complete system coverage, including newly developed feature tests [22]. This is an unrealistic expectation, given quick microservice evolution turnarounds, delays in tests, and often time-related costs with testing. Using the production-level system to extract traces would not solve the problem since users might not use all system features. Furthermore, we would only know about potential issues when it might be too late, and it could take days after changes are introduced to the codebase.

Therefore, although it is possible to implement dynamic analysis for the purposes of this project, it will not be the method we focus on due to the limitations mentioned above. Instead, we narrow our SAR to static analysis.

2.1.1 Static software architecture reconstruction

Static analysis can be performed on a system before it is deployed, extracting information from existing artifacts that would otherwise have to be manually analyzed. In particular, analyzing a program's codebases has played a part in formal verification of a system's correctness [20, 23, 24]. It has been used in automatically generating test cases for a program, for example, by identifying points for performance analysis instrumentation [25] or by extracting and analyzing an abstract syntax tree to identify all execution paths that need to be tested [26]. Developers can also use static analysis to better understand a program at a higher level. For example, UML models can automatically be generated by static analysis for legacy systems to better understand how to maintain or replace them [27], and it is integral in identifying code clones [28–30]. Static code analysis has also been applied in the realm of microservices. It has been used to identify calls between microservices to generate security policy automatically [31]. Also, it has been used to analyze monolithic applications to recommend cuts for converting to microservices [32]. In generating a service dependency graph, Esparrachiari et al. [33] posit that source code analysis is not sufficient since the deployment environment may impact the actual dependencies, which the given deployed module has. However, our goal is different from theirs; we do not necessarily target every possible call in a system for dependency detection; rather, we find the calls that are part of the application's business logic and, for this purpose, the source code contains sufficient information. Pigazzini et al. [34] 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, they mainly focused on the identification of the anti-patterns. Rahman et al. [35] followed a similar approach to parsing the code but developed a tool named "MicroDepGraph"[1] to visualize the call graph between microservices.

Source code is not the only artifact available for static analysis. It is important to mention that many core artifacts like maven or docker files are typically included in the codebase and can be used for static analysis. Ibrahim et al. [36] use 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. This allows the creation of a centralized security concern for the system, but since it does not extend to source code, it cannot include security flaws in the programs deployed in the containers, only flaws with the images themselves. Another static source of information is in the API definitions. Mayer and Weinreich use API definitions generated by the Swagger tool as input to their architecture generation system. Still, their system is also dependent on runtime data extracted from calls between services [37].

Another approach to pre-runtime SAR is to embed a source of information in the microservices as part of their development. For example, Salvadori et al. [38] propose creating semantic microservices that expose information

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

about their resources, allowing them to be automatically composed. In this way, a centralized view of microservice communication is always available. However, this method depends on using a fundamentally different approach to development and cannot be used to analyze existing codebases.

2.2 Architectural views

As mentioned at the beginning of this section, there exist different meanings for software architectures to different experts. Each expert might come with distinct viewpoint on the system [18]. Each viewpoint may govern one or more architectural views comprising a portion of an architecture description. Such architectural views [17, 18] capture certain system qualities or aspects. Selected views are elaborated by the 4+1 architectural view model [39] (that can be generalized to the N+1 model [40]) involving logical view, process view, development view, physical view, and scenarios, which do not have a visual format.

The ability to reconstruct effective architectural views of a system constitutes the foundation for successful SAR [41]. Specifically, in the case of SAR applied to microservices, Rademacher et al. [16] have considered four views (domain, technology, service, and operation) as their comprehensive perspectives.

Each of these views focuses on particular aspects within the system. And, they also interconnect with one another. For instance, the service view intersects with the domain view, detailing which data entities are linked to endpoints. Subsequently, the technology and domain views reveal where these data entities persist.

The process of construction of these views, according to Walker et al. [41] has a key point based on the fact that each view is an aggregation of smaller views, each illustrating a disparate microservice. In other words, a fully centralized perspective of the system's architecture can be constructed by aggregating views from each microservice that is seen as operating within its bounded context [42–44]. These views can be contrasted with the cloud-native approaches proposed by Carnell et al. [14] when building microservices. It is highlighted in chapter 3 of their book that in order to ensure proper microservice development there are different roles involved. These roles are intended to focus on different parts of the system.

Moreover, other perspectives can be involved, such as security audit. Expecting a single person to master each of these roles and their unique perspectives is impractical. Each role should have its own specialized architectural view and be able to evaluate the system both at the microservice level and holistically. These specialized views have been defined as *logical view*, *process view*, *deployment/physical view*, *data view*, *security view*, *implementation view*, *development view*, and *use case view* [45]. However, not all views can be easily extracted, such as the use case view.

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 to quickly understand how dedicated parts 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. Visual models can influence the efficiency with which practitioners analyze and understand the system view the model is displaying.

The following two different approaches employed these concepts for visual modeling of enterprise systems' architectures:

2.2.1 Traditional view modeling

Zhou et al. [46] sought common visualizations for enterprise architectures. Their study indicates that the most commonly used modeling languages for enterprise architectures are ArchiMate, followed by UML, Business Motivation Model (BMM), and BPMN, among others. The main emphasis of enterprise architectures is on business processes. ArchiMate has derived a number of concepts from UML but focuses mostly on services rather than objects, which is the case of UML. The advantage of ArchiMate is that it describes large systems, while UML does small modeling by showing greater details. However, both ArchiMate and UML operate in the 2D space.

Zhou et al. [46] highlight frameworks of the enterprise architecture practice. 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. [16]. However, we should note that the business architecture levels are not covered by Rademacher et al. [16]. This architecture involves motivation, organization, and asset mapping, which, although encoded in the system, primarily influence the implementation's motivation.

With regard to hierarchical approaches, the C4 model (Context, Containers, Components, and Code) is worth mentioning as a practical approach for software architecture modeling [47]. It is a hierarchical model consisting of four levels of abstraction, ranging from the high-level system context to individual code elements. It is used for a variety of system types, including microservices. While it does not prescribe a method of analysis, its key feature is important for analysis tools to follow, such that it allows varying levels of abstraction for the users to see. Such a hierarchical analysis is useful, especially for microservices, as it allows views of the system as a whole and inspection of individual services.

2.2.2 Alternative visualization for software architecture

Alternative visualization practices have emerged for software architectures [48, 49]. Shahin et al. [48] categorize alternative visualization as graph-based visualization involving graphs showing nodes and links similar to ontologies. This approach was the most common approach we identified in the microservice literature. Another approach is a notation-based visualization such as UML or SysML. However, matrix-based approaches also exist [50]. They can act as a complementary representation of a graph. Among others, metaphor-based visualization uses familiar physical world contexts (e.g., cities, islands, or landscapes).

Shahin et al. [48] noted that these visualizations often serve a specific purpose. They mentioned that most of the time, it is architecture recovery, which we call SAR in this article. The second most common purpose is architecture evolution, followed by architectural evaluation, impact analysis, general analysis, synthesis, implementation, and reuse.

The 3D space has also been assessed in the literature. To make the system more understandable using a visual metaphor, Virtual Reality (VR) and Augmented Reality (AR) methods have been explored for software architecture visualization. One example is the "software city"; software packages are represented as buildings and their dependencies as streets. Fittkau et al. [51] implemented the software city metaphor in virtual reality as shown in Fig. 1, while Steinbeck et al. [53] present a more advanced and scalable derivative called EvoStreets, which gives a better view of the software's hierarchical makeup.

Schreiber et al. [52] proposed a related visualization method more closely applicable to a microservice architecture, shown in Fig. 2. Their approach shows 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. While this approach has only been used on monolithic applications, the island metaphor is very suitable for displaying the relationships between independent modules in microservice architecture.

The problem of large data sets and visualizations has been recognized by Adrienko et al. [54]. Visualizations that represent individual entities suffer from overplotting. It has been proposed that multiple views be coordinated [55] to cope with such a problem, but it is rarely used in

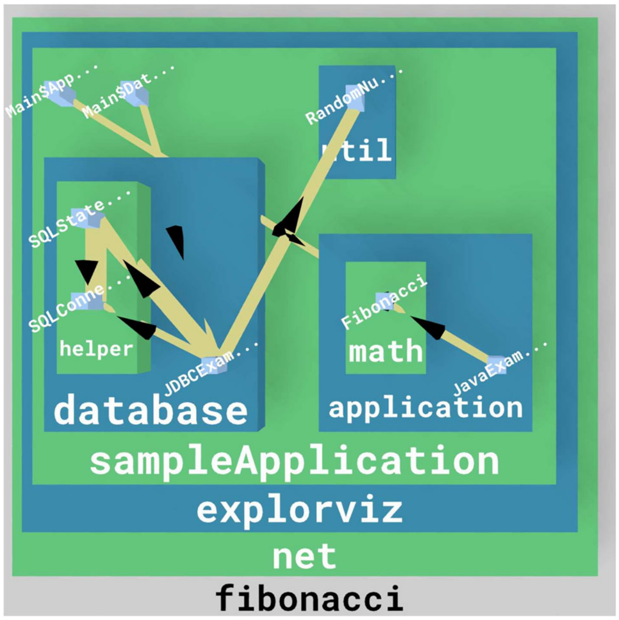


Fig. 1 The "software city" metaphor represents an application's package structure as a collection of buildings. In this visualization, packages are shown as flat green and blue boxes. The purple boxes on top symbolize the classes, and the yellow lines between them represent communications, with their width indicating how often calls are made. [51]

commercial systems. Such a principle could complement and greatly improve the C4 model [47]. A similar principle is also used in provenance tracking [56] where we move in one model in time and plot another as we move.

Large microservice-based systems are prime candidates for being visualized using VR/AR. The VR-EA tool from Oberhauser et al. [57] is an attempt to visualize larger enterprise applications. Instead of doing dynamic or static analysis to extract a model, VR-EA uses modeling tools as inputs to generate a 3D VR view in the virtual reality of business processes and their relationships with enterprise resources. This approach can provide a comprehensive view of the enterprise system, as it can show a large group of interconnected components. However, it depends on a set of models that must be custom-created to capture the relationships and complexities inside the large system, requiring manual creation of additional configuration and artifacts.

Virtual reality was used by Ma et al. [58] to monitor a distributed set of servers, visualizing each server as a physical machine in the same VR room. Although the monitoring capabilities were limited to system resource usage, the tool showed that physically disparate systems could be virtually collocated to provide a centralized view of a system.

More generally, large systems beyond software architecture have been explored in virtual reality. For example,

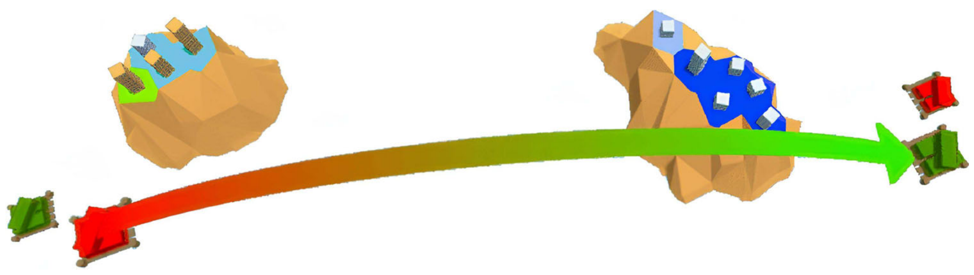


Fig. 2 Software components can be displayed using the "software island" metaphor, showing components and their inter-dependencies. Islands represent bundles, packages are represented as colored regions on the island, the classes are represented as buildings on the regions, the ports near the island represent the package exports and imports, the arrows between ports represent dependencies. The island on the right imports packages from the island on the left [52]

Toumpalidis et al. [59] used augmented reality to visualize data from IoT networks. A user could see a summary of a device's data overlaid on that device. This experiment showed that AR is useful for displaying and aggregating distributed data.

Three-dimensional visualizations can also be employed to visualize complex information relationships. Halpin et al. [60] use virtual reality to display the relationships between patent registrants as shown in Fig. 3, and Royston et al. [61] use a similar idea to display connections from social media sites. Neither of these approaches uses a specific visual metaphor; instead, they opt to display their contents as simple graphs in three-dimensional space. These examples have a common structure with microservice architectures, as microservices architecture can be viewed as a network of services communicating with each other based on semantic relationships.

Moreover, Moreno-Lumbreras et al. [62] have suggested using VR to visualize development metrics and analytics in three-dimensional space. Acquiring a broad range of aspects and views for this kind of visualization may be regarded as similar to architectural reconstruction. They analyzed and compared the comprehension of metrics in code reviews when aided by VR or 2D visualization. However, no results are available yet.

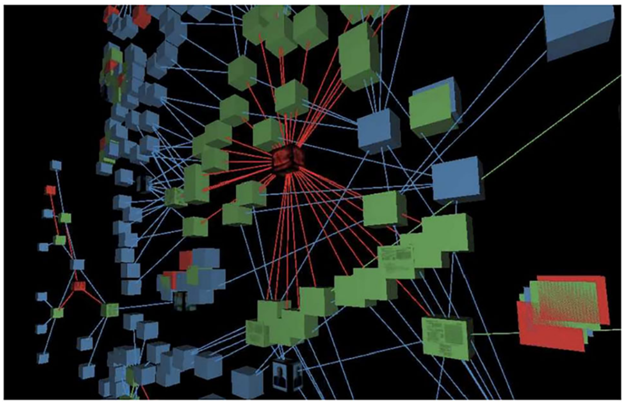


Fig. 3 Semantic information is a candidate for being displayed in large, three-dimensional graphs due to the natural connections between the elements [60]

The perspective of system evolution could be well addressed by dynamic graph visualizations. A dynamic graph visualization considers the challenge of representing the evolution of relationships between entities in readable, scalable, and effective diagrams. Becket al. [63] provide a broad survey. Apart from static graph visualizations with node-link and matrix representations, dynamic graphs can be represented as animated diagrams or as static charts based on a timeline. Such graphs can find a great fit with system evolution and provenance tracking. Yet the study noted the greatest challenge with visual scalability followed by extended data dimensions and interaction.

A broader introduction to visual graphs and metaphors and empirical user evaluations is provided by Burch et al. [64]. While recommendations exist for graph drawing to make it faster and more reliably explorable, many studies are still ongoing to uncover how well users perform specific tasks. The work guides the design and application of graph visualizations. Apart from other perspectives, it suggests interpretation, graph memorability, and graph creation. The study illustrates time-varying graphs, various representations, tracking, colo-codes, interactive features, aesthetics, metaphors, augmented reality, and other perspectives used across disciplines. Many of the presented advancements would greatly suit the microservice domain, and current tools greatly lack such advancements.

2.3 Microservice visualization in the industry

Microservice development comes from practitioners, and research tends to come later, so publications about microservices are still limited in a lot of areas. Thus, grey literature may hold valuable insights that academic literature simply cannot provide yet [6].

Amazon provides a solution called X-ray console.[2] The provided approach is a map visual representation that

[2] https://aws.amazon.com/xray/

consists of service nodes that serve requests, upstream client nodes that represent the origins of the requests, and downstream service nodes that represent web services and resources used by an application while processing a request (as depicted in Fig. 4). The X-ray console provides embedded views that enable the user to view service maps and traces of applications' requests.

SIMulate Interactive Actor Network VIsualiZation (Simianviz) created a simplified Netflix service interconnection based on http://netflix.github.io/. It provides an interactive visualization technique for the system.[3] Figure 5 shows the service graph representation of the system. It illustrates service dependencies in the whole system and enables the user to reconstruct the services communication graph to analyze different topologies. However, such a topology view is not particularly useful in debugging where a specific service is experiencing an issue.

3 Software architecture reconstruction of microservices

Microservice architectures essentially operate on a decentralized basis, and that is why some methodologies as Heroku's 12-factor app [14, 19, 34] recommend that each one of them be self-contained with its own codebase and database in order to improve evolution, scalability, and dependency management. However, being decentralized does not mean that they are isolated; in fact, the microservices interact using interfaces or message queues. Therefore, there is an interdependency among microservices, although generally a loose one, and given that these interactions occur via interfaces, a prominent dependency often lies in the endpoint names and the parameters that symbolize data or transfer entities.

In the realm of domain-driven development [1, 16], each module encompasses a bounded context [14], that offers a limited perspective of the system holistic data model, which is called context map. Frequently, these bounded contexts exhibit partial overlap with other modules via specific data entities and this overlap can serve as a component to derive the system-wide perspective. Additionally, interactions between services, like REST/RPC endpoint calls, provide another element to factor into our considerations. These two basic strategies are illustrated in Fig. 6.

3.1 Software architecture reconstruction process

The SAR process we have used involves a static-code analysis of each microservice's codebase. In this aspect, each one of those microservices follows the enterprise architecture standards of the layers of communications where it has a separation of component types in Controller, Service, and Data Repository. Moreover, this process is organized into four main phases: *extraction* phase, *construction* phase, *manipulation* phase and *analysis* phase. In order to better understand these phrases, next sections briefly describe their details.

3.1.1 Extraction phase

The extraction phase constitutes the initial phase of our process, where we use Abstract Syntax Trees (AST) analyzed from the code. As its name indicates an AST is a tree representation of the abstract syntactic structure of the code and is used as the base for other methods. We navigate this tree in order to extract call graphs (calling relationships between subroutines in a program) from the recognition of method calls. Methods at the top are potential endpoints; moreover, frameworks commonly supplement these endpoints with extra details (such as HTTP methods, constraints, etc.) that indicate the endpoint (i.e., in the form of annotations or external files). Upon identifying these endpoints, we evaluate their parameters and follow the calls down through the controllers, services, and repositories to the referenced and involved data entities. Then, it is possible to identify the types of these components by evaluating the corresponding attributes in the AST and the call graph originating from the endpoint. We can also detect inter-relationships among the entities within a microservice codebase and extract the foundational data structure. From a reverse perspective, we identify the data entities that specific endpoints interact with and by exploring the call graphs, we uncover associated constraints, clear policies, conditions, branching paths, and iterative loops. Specific focus is placed on the data entities because by evaluating their attributes and methods, we can discern interconnections among the entities within a particular microservice codebase, allowing us to unveil the foundational data framework.

3.1.2 Construction phase

The construction phase utilizes outcomes from the previous phase. Here, the microservice-specific details are transformed into a first level of intermedia representation. This stage utilizes components pinpointed during the exploration of call graphs. We enhance these identified components by integrating additional information that could be present at the component definition layer. For example, REST controller endpoints might stipulate access permissions [65, 66]. It is necessary to point out that paying attention to components corresponds to the microservice

[3] http://simianviz.surge.sh/netflix

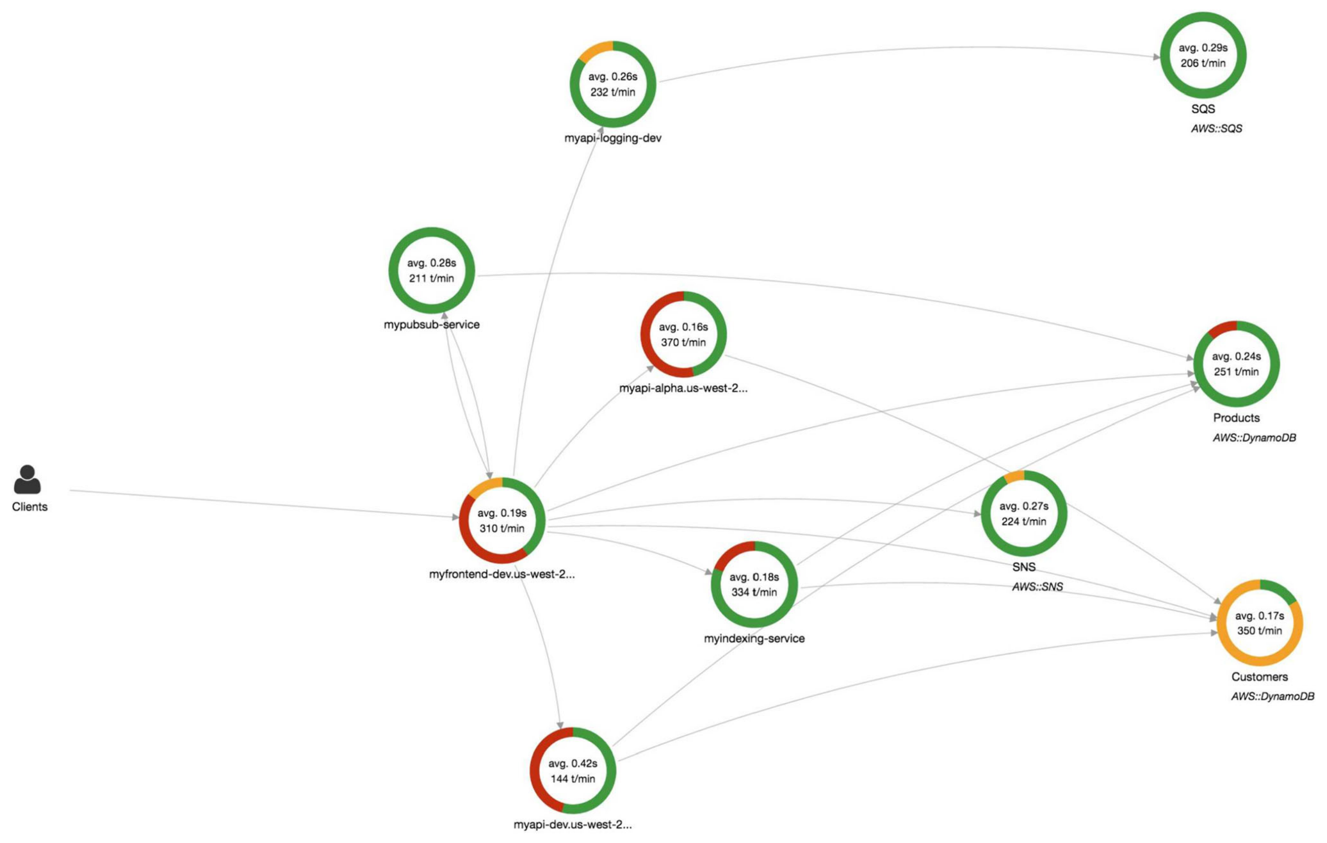


Fig. 4 Amazon X-ray console an industry tool for microservices visualization highlighting connected sectives and real time information from response times

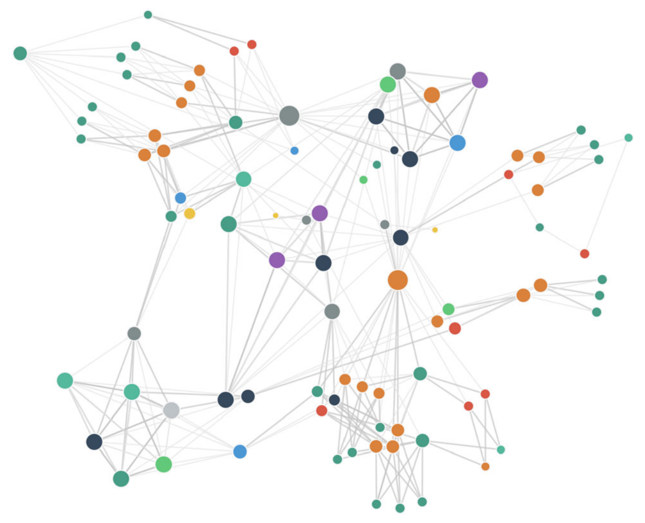


Fig. 5 Simianviz created a simplified Netflix service interconnection based on http://netflix.github.io/. It provides an interactive visualization technique and shows the service graph representation of the system

development practice [14], which means that the microservice will always process data and provide endpoints.

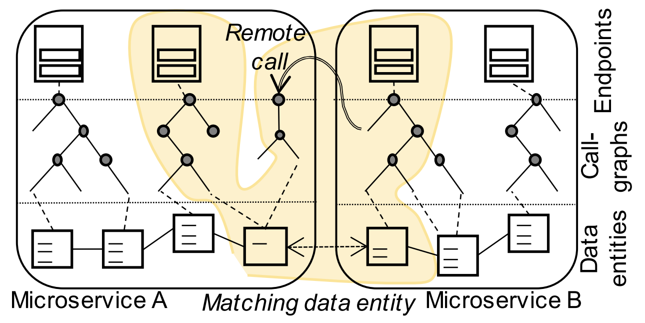


Fig. 6 Illustration of microservice dependencies used to combine microservices. Yellow area highlight indicates overlap across two microservives. Each microservice highlights 3-layered architecture with data entities, services and controller endpoints

The intermediate representation, which is constructed, is the Component Call Graph (CCG). It is created by adding additional properties to each component, like its type and its properties [67], with connections implied by calls. Figure 7 shows an example of the CCG. The type of each component is specified at the top between brackets. The properties that are extracted from each type are different, as can be seen in the cards attached to each component. The inclusion of these properties turns the call graph into a CCG, giving us a more profound insight into each

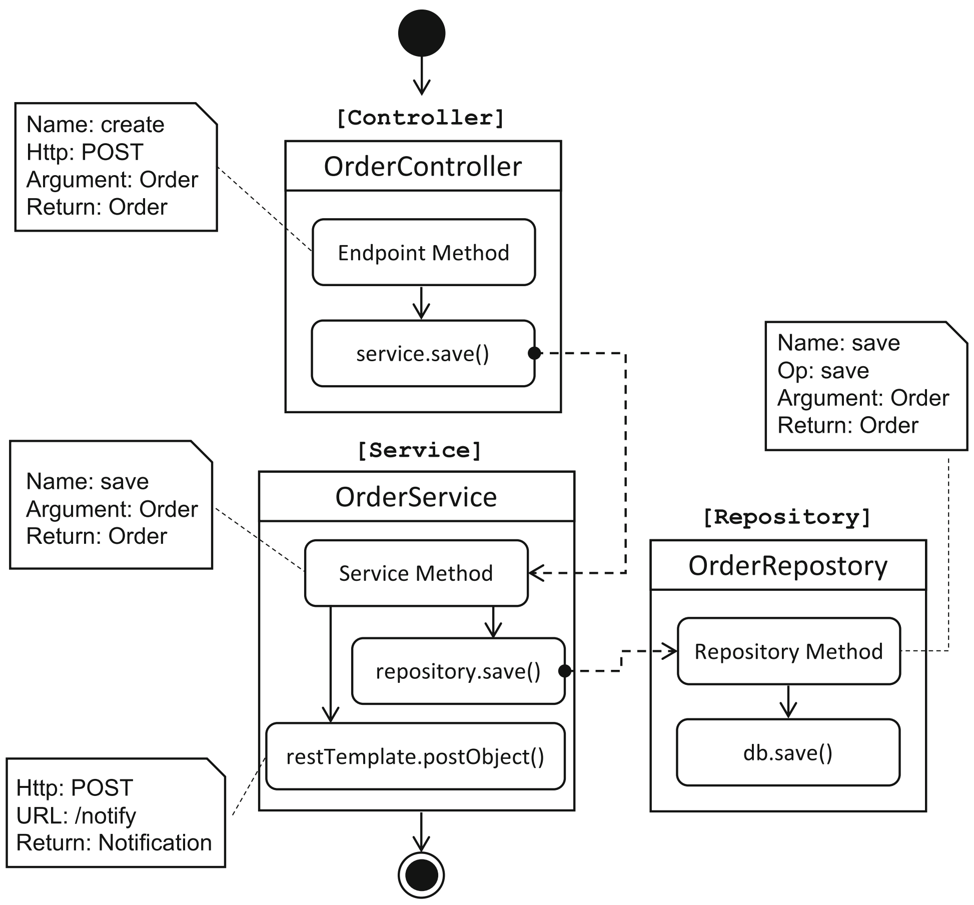


Fig. 7 Component Call Graph example indicating one slice from Fig. 6, illustrating how a controller endpoint connects to a service and respistory or remote call while working with an data entity 'Order'

component's role in the system for the upcoming phase. When we create individual CCGs for every endpoint, they collectively represent a single microservice. Similarly, as we identify components, we can also recognize external calls because they are executed via clearly defined interfaces or constructs [13]. These REST calls can then link to other external microservice endpoints based on a matching method signature, providing a broader view of the system's interdependencies.

Therefore, by gathering components and integrating their call sequences, types, and properties, we produce the CCG intermediate representation of the microservice in the form of JSON.

3.1.3 Manipulation phase

The manipulation phase, the third phase of our process, focuses on connecting several intermediate representations of microservices.

As previously described, our approach leans on two primary elements: the integration of overlapping data entities and interactions between microservice endpoints. This methodology is illustrated in Fig. 6. Yet, these techniques can be expanded upon, such as incorporating details from build and deployment scripts. Additionally, event-driven methods involving message brokers, like Kafka or Messaging Queues, can be assimilated into this phase. The merging process begins with entity matching, where we search for entities from different modules that show a subset match in properties, data types, and potentially names. We employed natural language processing techniques for this matching, specifically the Wu-Palmer algorithm [68]. Subsequently, by combining all the pertinent microservices, we extract the unified data model, also known as the context map. Through these matched entities, we further establish data and control dependencies.

A second element is based on interactions between services. Initially, we pinpoint all endpoints, their parameter types, and accompanying metadata [66]. We then locate the remote procedure calls embedded within these methods. After identifying them, we pair these entities, craft a comprehensive system service perspective, and enhance the unified model that emerged from the preceding approach.

The outcome of our manipulation phase is an integrated intermediate graph in the form of CCG that represents the entire system. This expanded viewpoint sets the stage for subsequent analysis, equipped with insights into the unified data model, dependencies between services, and a comprehensive list of system service endpoints. While it provides a holistic view, it also preserves details specific to each microservice. This includes its defined boundary with intersecting areas, technological data, and a cumulative index of technologies, categorized by layer, in the unified view. Finally, because each microservice encompasses information about its construction, deployment, and operation, this consolidated view can produce a graph illustrating interconnected deployments.

3.1.4 Analysis phase

The analysis phase constitutes the last step of the SAR process and involves drawing conclusions about the entire system. In essence, any form of deduction could operate with the intermediate system representation. For example, we could detect design smells or security policy violations [66, 69]. If we want to reason about the system, we want to do it with the information extracted from the most current version. This most current version resides in the codebase and can be extracted automatically using static code analysis in a continuous integration pipeline.

Figure 8 shows the whole process. Once the system's intermediate representation is constructed, we can move to architecture visualization. Having an intermediate representation that contains necessary information from the microservice system unlocks endless visualization possibilities. According to the needs of DevOps, developers, architectures, stakeholders, or others, a visualization based on the same JSON intermediate representation can be created.

A holistic visualization of the system is ideal for reporting, aiding navigation, and enhancing understanding. In this aspect, the present article narrows its focus to our newly introduced architectural visualization process, which we will delve into in the subsequent section.

4 Software architecture reconstruction and microservice visualization

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 [69, 70] (i.e., poor design choices and anti-patterns), or consistency errors. This manuscript considers deriving conventional architectural visualization and questions whether alternative visualization could dedelivered for microservices that could improve their architectural analysis.

The SAR process may result in multiple architectural views. However, limiting our attention to a few views for a proof of concept is sufficient. Thus, we should consider the most suitable views for microservices that support a system-centric perspective.

Mayer and Weinreich [71] 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. Rademacher et al. [16] suggests focusing on the **service view** as it defines the microservice's APIs and the inter-service calls between them. Furthermore, this view is also relevant for developers seeking to understand the system's operation. Besides this suggestion, they also focused on the **domain view**, since it defines the domain model used by the microservice system. It is 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. Instead, each service operates on its own bounded context, which operates on the subset of entity attributes it needs [72].

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

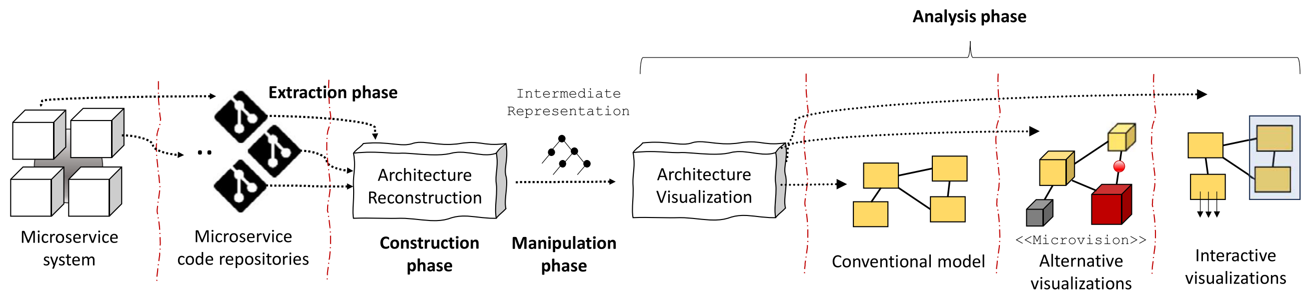

Analysis phase
Extraction phase
Intermediate Representation
Architecture Reconstruction
Architecture Visualization
Microservice system
Microservice code repositories
Construction phase
Manipulation phase
Conventional model
<<Microvision>>
Alternative visualizations
Interactive visualizations

Fig. 8 Construction process of our approach presented in this manuscript indicates indivisuakl inputs and phases to derive intermediate representation of the system and confect it into a visualization

current architecture against the planned system architecture and detect deviations. They can also use it as a tool to assess warnings to detect if the architecture has drifted from the original plan [73] and help them to make informed decisions to mitigate evolution issues.

Given the above reasons, this work will focus specifically on domain and service views. Alshuqayran et al. [74] identified that communication and integration is the most microservice challenge in the literature. The domain and service views aid in identifying the list of microservices that can be called from a specific one. Also, they assist in identifying domain classes that are exchanged among microservices during inter-service calls, which can be instrumental in addressing communication and integration challenges. Additionally, Gortney et al. [10] found that the most commonly extracted model through dynamic practices is the service dependency graph, and this graph can be directly extracted from the service view.

Given the same intermediate representation of a microservice system can be used for arbitrary view, we will use these two views to illustrate different visual models. Similar to the static analysis approach detailed in Sect. 3, we implement various proof of concept visualization tools that use our intermediate representation.

4.1 Practitioner needs in microservice architecture analysis

Bushong et al. [2] performed a systematic mapping study on microservice analysis and architecture evolution, highlighting practitioners' intents to improve system qualities or detection of issues. With system evolution, inadequate solutions can be applied to the system due to the rush and market demands, which might introduce technical debt and, eventually, architectural degradation where the actual system architecture deviates from the prescribed one. Microservice architecture analysis can be performed to assess various system qualities, including security or performance. Change impact analysis or root cause analysis is another common task.

No matter the viewpoint (Sect. 2.2), practitioners need quick access to information to make informed decisions. They need a white-box approach to analyze the architecture. The lowest level might be manual code review; however, models bring great simplification to the assessment process and abstraction. Static visual models can bring the abstraction [63, 64]; however, the quantity of information for microservices might be a concern as known across fields [54, 55], leading to clutter and difficult navigation. In our work, we focus on the service view and domain view, and we detail the common purposes and expectations from such views.

A service view provides a high-level overview of the various microservices that make up the entire system. This view highlights the relationships, interactions, and dependencies between different microservices. The service view in microservices architecture helps practitioners understand the modular and distributed nature of the system. It should illustrate the key components and their connections within the microservices ecosystem. In such a view, practitioners find information to enhance design, manage, and optimize the system. They need to identify microservices and their boundaries, perform dependency analysis (i.e., change impact, error propagation), and find potential points of failure such as too many dependencies, cycles, etc. A large amount of service dependencies may cause performance issues or worsen resilience, scalability, and fault tolerance. So, the management of dependencies across microservices is a fundamental task to make informed decisions. They might also need to manage the cardinality of dependencies or identify potential bottleneck microservices.

Well-suited service view visualization should enable practitioners to understand the system overview with an abstraction that gives a navigable access to individual microservices to observe their endpoints, interconnection and dependencies. Thus, it should be easy to find and identify microservice, assess its connections, and inspect cardinality of dependencies. Since microservice connections could easily clutter, a distinction like color codes could help the observability. Sucessful view would thus offer navigation capabilities such as zooming and searching within the graphical system modal allows for swift recognition of properties. This facilitates easy access to essential details without causing distractions during identification.

The domain view focuses on understanding and modeling the business domain within which microservices operate. Practitioners perform various tasks related to view to ensure that the architecture aligns with business requirements to support business processes effectively. Microservices operate with their own business domains. However, to plan evolution, it is important to understand the decompositions in the centered view, especially since microservices domains partially overlap, forming dependencies. Practitioners need access to individual microservice domain models, see overlaps, and have access to the holistic perspective. It is important to highlight that the source code has limited detail of the domain view, and thus, it only provides approximation through individual microservice data models or the overall context map of the system (merge models through overlaps).

Appropriate visualization of the domain view would likely stand on established models that emerged from UML, such as class diagrams that show data entities, their attributes, and associations. However, such a visual model has spacing limits, and thus, one large model might not give the desired perspectives. Alternatively, a more fitting approach might involve an interactive view that reveals data entity names and provides attribute details when necessary. However, it remains essential to explicitly represent the relationships between the entities in some way for effective presentation. Moreover, subviews might fit the needs of practitioners for selected microservices which could bind with the service view as suggested by related work on multiple coordinated views [54, 55].

5 Case study part 1: deriving conventional visualization

To evaluate the proposed SAR methodology, we executed the process to generate service and domain views for microservices-based systems. Additionally, the implementation provides a rudimentary visual representation of these architectural views.

The objective of this study is to furnish a detailed guide to implementing the introduced SAR process for constructing architectural views. Additionally, it involves implementing conventional models to visualize the extracted views and evaluating the usability and challenges associated with their application.

5.1 Software architecture reconstruction implementation

We implemented a prototype tool Prophet.[4] Prophet performs static code analysis on Java-based source code, identifying the component-based constructs intrinsic to Spring Boot and Enterprise Java.

To extract the necessary system information to construct the service view, we need two things in general: 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. In our case, Prophet identifies endpoints, detects remote method calls in the microservice code, and provides them with access to the CCG. Prophet recognizes common constructs such as REST templates, etc. Once the endpoints and calls are 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 interact.

Moreover, the visualization approaches leverage the suggested CCG intermediate representation, established through the SAR process using a REST API. Beyond visualization, this intermediate representation serves to facilitate system analysis. For instance, it enables the identification of design flaws or security policy violations, as demonstrated in previous studies [66, 69].

To extract system information to determine the domain view, Prohet identifies data entities, their properties, and their relationships. Data entities use framework 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 each microservice, merging the bounded contexts can generate a combined canonical model for the entire system. 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 be appended to the merged entity. The result represents the scope of all entities used in the system.

5.2 Considered system benchmarks

To demonstrate visualization approaches for the purpose of this manuscript and on a large, realistic system, we adopted two test benches. The Teacher Management System (TMS)[5] consists of three microservices, and the limited size allows us to embed complete SAR visualization examples in this article. For demonstration of a real-world system, TrainTicket[6][75] is used (originating from the ICSE conference). The TrainTicket was designed to

[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://github.com/cloudhubs/tms2020

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

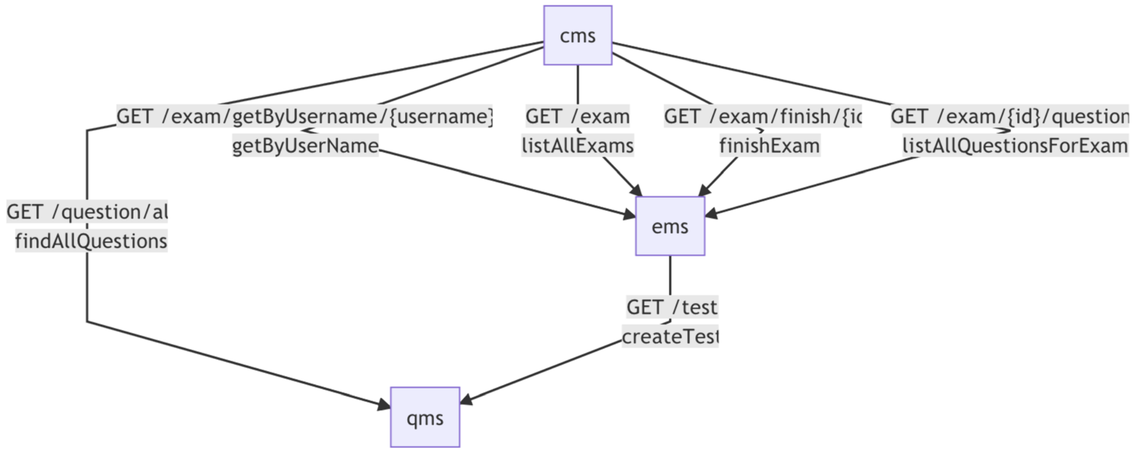


Fig. 9 Sample service view extracted from the TMS benchmark. It indicates connection of the there microservices

emulate a real-world microservice system consisting of 41 microservices and over 60,000 lines of code. It is written in Spring Boot, uses MongoDB as its database, and follows cloud-native practice with containers, routing, etc. The frequent use of the train-ticket microservice in literature [76, 77] highlights its importance. It serves as a key benchmark to test and evaluate new methods, including the one we have introduced in this study.

5.3 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. 9. It can be observed on Figs. 4 and 5 that represent industry tools.

The domain view is a perfect fit for the UML class diagram that represents the scope of all entities used in the system, as shown in Fig. 10 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.

5.4 Challenges in conventional architectural visualization

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. 11 shows service view output on the TrainTicket testbench (41 microservices), which becomes difficult to understand.[7]

Visualization directly affects the view's intended purpose as an artifact to help a stakeholder to quickly understand how microservices interact 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 [47]. 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

[7] TrainTicket service view full image is available at https://zenodo.org

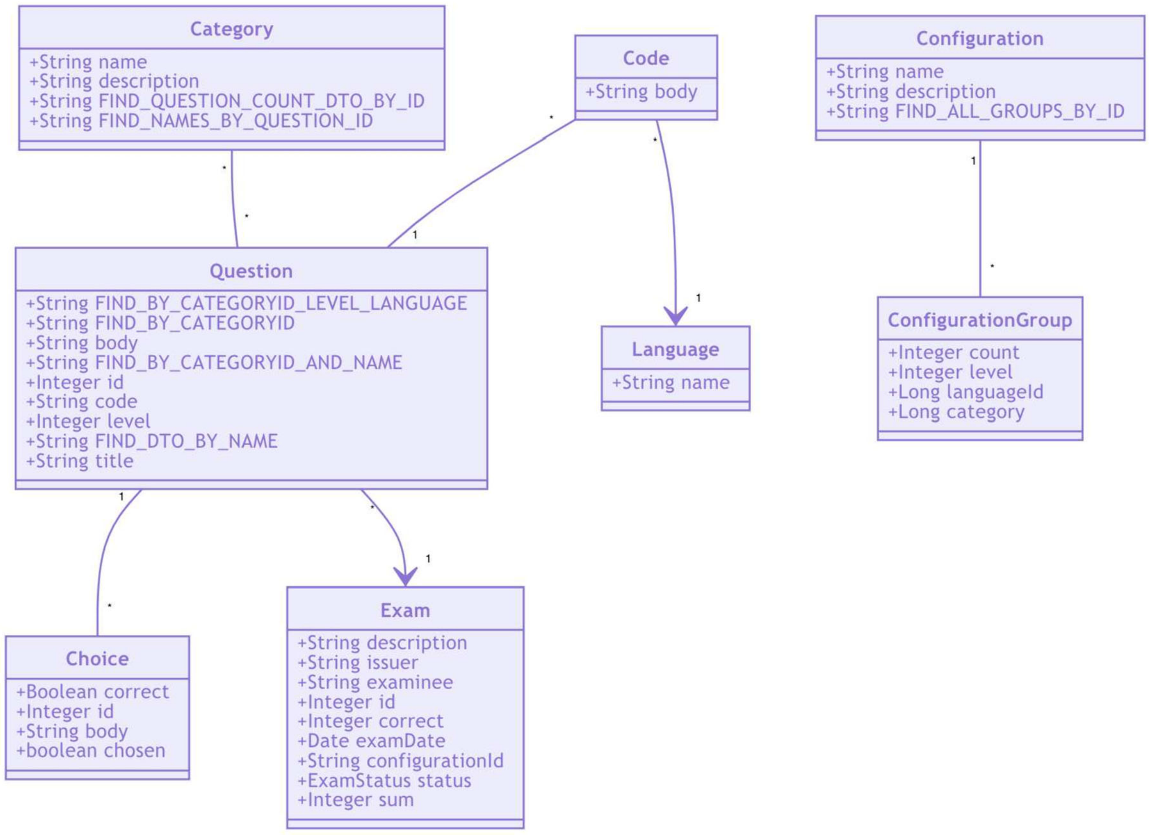


Fig. 10 Domain view derived from the TMS benchmark. These entities are aggregate definitions from partial entities in each microservice's bounded context

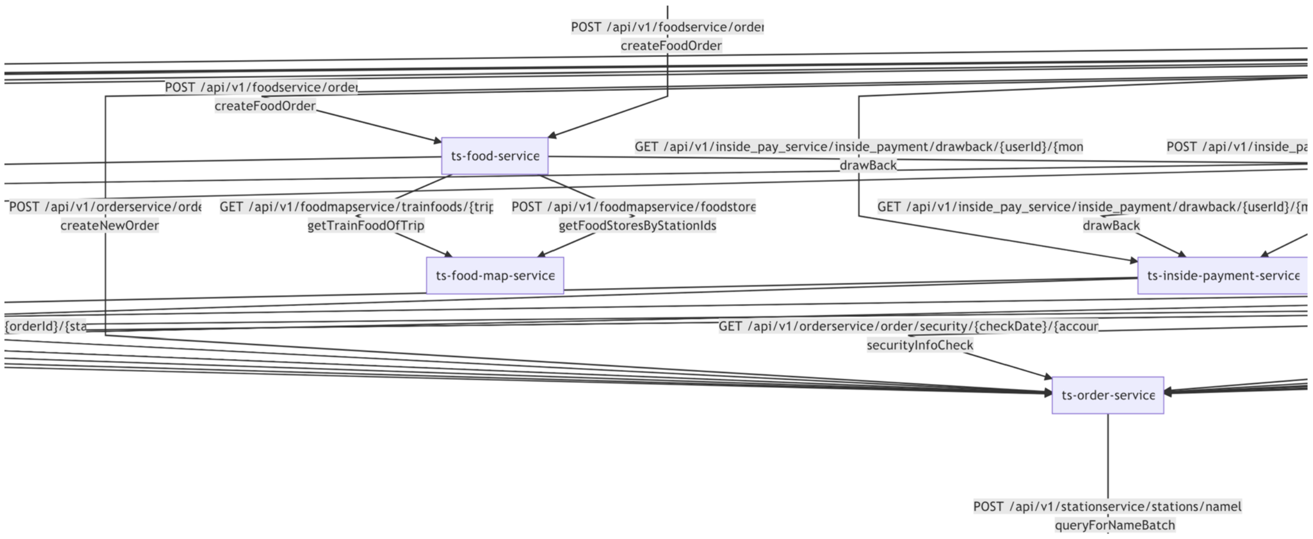

POST /api/v1/foodservice/order
createFoodOrder
POST /api/v1/foodservice/order
createFoodOrder
ts-food-service
GET /api/v1/inside_pay_service/inside_payment/drawback/{userId}/{mon
drawBack
POST /api/v1/inside_pa
POST /api/v1/orderservice/orde
createNewOrder
GET /api/v1/foodmapservice/trainfoods/{trip
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. 11 The service view from a large microservice testbench TrainTicket [75]. Connections between services become difficult to decipher as the system size grows

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.

On the other hand, they appreciated the system overview along with an interactive display of endpoints in services and more easy-to-follow links across services.

6 Case study part 2: deriving alternative visualizations

Conventional visualization uses static graphs represented by rendered two-dimensional space as established models sourced from UML or SysML. This poses multiple challenges to visualization ability since we need an approach that scales better with system size. Alternative approaches should better cope with the quantity of information in microservices. Exiting surveys proposed various approaches [63, 64] to visualize large graphs, and this domain is no different. Apart from abstraction and scalability to large microservices comprising ten or hundreds of microservices, interactivity should be provided to navigate through the information and details. An effective solution should provide readable diagrams that make it faster and more reliably explorable information; in our case, it might be dependencies across microservices, their cardinalities, and bottleneck services. To support the search for properties, the visualization should be easily navigable and should be able to traverse multiple levels of abstraction.

Given our SAR infrastructure that extracts necessary system information to derive service and data views. We can experiment with and evaluate alternative visual models. We first considered an augmented reality visual model for the service view, as it brings alternative space, navigation, and interaction means. Consequently, with feedback from the augmented reality prototype, we looked into a more traditional approach using a web-based solution with interactive models that emerged from data science and visualization of large graphs. We describe these two in the following subsections.

6.1 Service view in augmented reality with a 3D model

The service view is the most applicable view for understanding the system-centric perspective and the system operation. In previous work [11], we adopted this view to explore the benefits of a three-dimensional visual scheme considering the conventional visual model. For that objective, an AR medium was used because it is natively three-dimensional and lends itself to control schemes based on natural movement. This combination holds potential for use in displaying and navigating complex systems such as microservices. We approached the visualization by using a 3D graph functioning within AR. Since we had automated the SAR process (Sect. 2.1) and recreated the service view in 2D, we used the same data for a 3D graph in AR.

The use of AR for software visualization is well-acknowledged, and it’s been employed to illustrate monolithic software systems in different manners. We expanded the existing 3D visualization techniques to a higher level of abstraction beyond a single piece of software to an entire distributed microservice system.

For the *system-centric perspective*, it is necessary to present a *high-level system* visualization. In the context of microservices, the objective is to visualize their interrelationships. However, it’s crucial to maintain clarity in the view, especially as the number of services increases. Quickly understanding the high-level structure should be prioritized in all system-centric perspectives. Furthermore, it is essential that users should be able to *interact* with the view and *navigate* through the microservice system both at a high level and a lower level of detail centered around a few services. In this context, the high-level view corresponds to the *overall structure of the system*, while the low-level view is focused on *individual services* and their immediate neighbors. Minimizing the transition time from a high to a low level of detail facilitates the understanding of the system and the roles the individual services play in it.

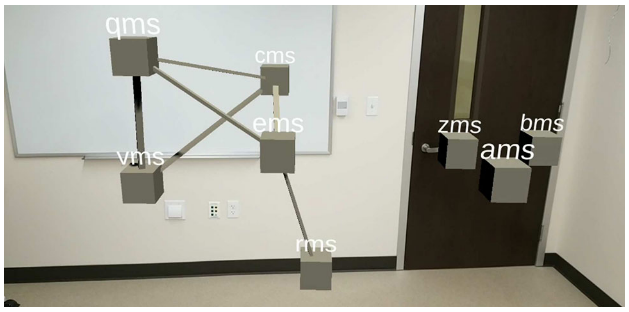
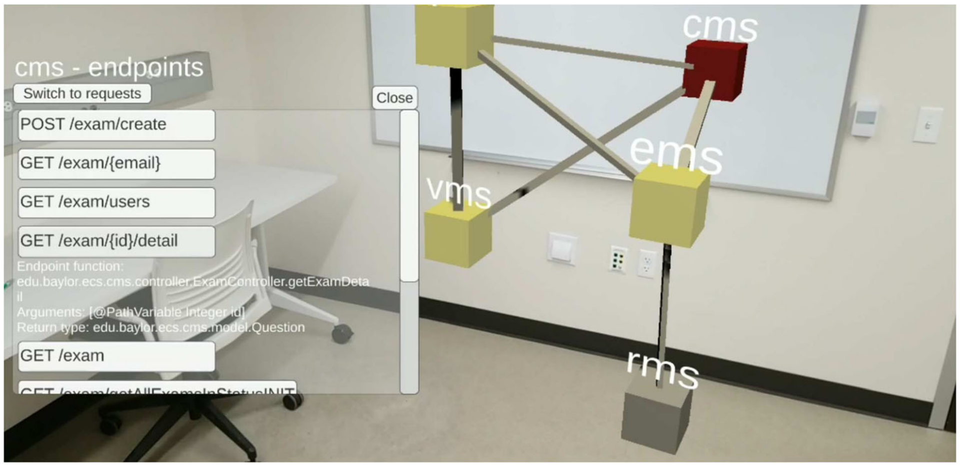


Fig. 12 Microvision showing a 3D graph of microservices and their connections



Endpoint function:
edu.baylor.ecs.cms.controller.ExamController.getExamDeta
il
Arguments: [@PathVariable Integer id]
Return type: edu.baylor.ecs.cms.model.Question

Fig. 13 The context menu shows a selected service API endpoints, in this case, the “cms” service highlighted in red

Based on our AR microservice visualization strategies, we formulated a proof-of-concept tool named Microvision, dedicated to presenting the service view. Microvision is fundamentally structured around two primary components: the main *graph display, and the API viewer*.

The **Graph display** projected in AR presents an abstract 3D graph view of the microservice system allowing to have a quick view not only of the system but also its services and connections. As shown in Fig. 12 each microservice is represented as a node and exists an edge from one node to another if there is a call between two microservices. Nodes are spread out to avoid clustering in any part of the graph. By selecting a node, its connections are highlighted, showing only those neighbors it interacts with. This emphasizes the selected node’s relationship with surrounding nodes.

The **API view** is designed for a detailed view, showing the endpoints of a chosen microservice. In Fig. 13, when a microservice is selected, an API box appears listing its endpoints. This list is not shown initially to avoid overwhelming users with excess data. Endpoints are marked by their path and HTTP method. Tapping an endpoint reveals more details, like the method behind it and its parameters and return type.

Based on the elements presented above, it is possible to propose that our proof-of-concept, Microvision,[8] overcome the limitations of the conventional 2D visualization. We developed a 3D visualization based on AR that offers better scaling with the number of services than a 2D diagram. In addition, we demonstrated navigation and control through the reconstructed microservice system architecture because the graph is displayed in AR and is easily traversed by natural movement.

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

6.1.1 Assessment and challenges

To assess the AR model approach’s advantages in a practical setting, we used a realistic microservice system benchmark and applied our static analysis to obtain an intermediate representation of the system. We extracted the full system and also considered a subset to make it represent a smaller system sample. From the results, we first generated a sample conventional 2D model involving established practices, and then we derived an AR model for our prototype. To receive extensive feedback on AR model benefits and limits requires interaction with practitioners on sample use cases relevant to the system understandability or system architectural specificity. Thus, we intended to conduct a controlled experiment involving microservice practitioners. However, such a study requires broad details and explanations beyond the scope of this manuscript. Thus, we developed a protocol and performed the study separately. We provide comprehensive details of this study involving 20 practitioners through a separate publication [78] and share the important outcomes from such a study in the context of this manuscript.

When we assess decentralized system architecture, we typically need to recognize dependencies across microservices but also service cardinality (the degree of dependency between microservices) or bottlenecks (most dependant microservices among the whole system). Our study showed that the AR model enables novice practitioners to perform the detection of microservice dependencies, as well as experts do it. Such tasks are common when we make system changes or change impact analyses intending to understand involved parts of the system. Our study also showed that the AR model, given it operates in 3D, fits large systems better so that tasks can be performed more effectively; however, there is no large impact on the small system. Following the same for the experience level significance of practitioners, the analysis showed that the conventional 2D tool requires more experienced participants in order to detect the dependency of a system. With regard to other tasks, such as the identification of service cardinality and bottlenecks, our study showed that both conventional and AR models performed equivalently.

The study showed that conventional 2D models are not more applicable to common architecture analysis tasks than what AR models can offer. Participants also responded that the conventional approach is less easy to use or understand and more relevant to a larger system. The AR model also enables faster extraction of information. However, the conventional model seems to be better suited to finding all the information needed to make a task conclusion.

Practitioners have provided useful feedback on the AR model prototype for the service dependency graph. Such feedback can help researchers to build more robust tools supporting such visualization. They indicated that the difficulty with AR is that they must move a lot around the graph, and they considered it impractical that microservices have fixed locations that cannot be relocated. They lacked a zooming feature and indicated that color codes could improve the perception, especially when colors could be customized. They found it hard to search, given no search feature was provided. Another suggestion was to consider the orientation or direction of dependencies. We must also consider that an additional device is necessary to navigate the AR model. While effective, it is not practical as developers expect integration with their development environment. Interactivity adds great value, but users expect common browsing features like search, which are not implicit in alternative models. Interaction with a team becomes hard as everyone has their own view, and similarly, information transfer is difficult given the disjoint between development and interaction environment.

In support, the participants suggested that the AR model provides simpler dependency tracking and improves their understanding of the system by showing an overview picture of the system and clear dependencies between services. However, it is crucial to emphasize that for Augmented Reality (AR) to attain success, it must integrate numerous features and standard expectations based on feedback. For example, users expect a fast and interactive view with efficient search and filter capabilities. In the next section, we are going to describe an alternative solution to the mentioned limitation based on an interactive visualization.

6.2 Web-based interactive 2D and 3D models

The interactive visualization allows the user to manage, select, customize, and section the information contained in the architectural views according to their needs and objectives. By enabling users to engage directly with the visual display, information becomes markedly more accessible. For instance, if a user wishes to navigate to a particular microservice or entity and is familiar with its name, searching becomes expedient compared to scanning through entities sequentially. Manipulating the visual graph-whether by repositioning a node relative to others or rotating the entire view-enhances the presentation of information for individual users. Isolating and emphasizing a particular segment of the view can be especially beneficial for certain users.

We implemented a prototype[9] that is able to visualize the service view of a system based on the intermediate representation extracted during SAR. Moreover, it is capable of representing the visualization using a graph in two formats: 2D and 3D. Figures 14 and 15 show examples of these formats of interactive visualization of the service view respectively.

The prototype is designed to enhance user interaction and visualization. It grants users a range of actions for seamless navigation and information retrieval, detailed as follows:

1. **Zoom Functionality** Users can effortlessly zoom in and out of the graph for a more detailed or broad view.
2. **Node Relocation** Users can relocate a node, representing a microservice, to a different place on the graph using the drag-and-drop feature.
3. **Microservice Search** An efficient search tool allows users to quickly find a microservice by its name.
4. **Microservice Filtering** Users can filter by a specific microservice, with the system highlighting only its neighboring microservices.
5. **Detailed Microservice Information** By clicking on a microservice, users can view comprehensive details including both its dependencies and dependent microservices.
6. **Connection Information** Clicking on a connection between microservices reveals detailed information about that specific link.
7. **Diverse Feature Visualization** To aid visualization, the prototype presents features in various forms:
   - **Node Representation** Nodes are depicted in diverse shapes to denote their type - service, database, or API.
   - **Connection Highlight** Hovering over a node automatically highlights its connections, offering users a clear and immediate visual cue.

By integrating these features, the prototype not only ensures a more intuitive and efficient user experience in managing and navigating microservices and their

[9] https://github.com/cloudhubs/ArchitectureVisualizationPOC

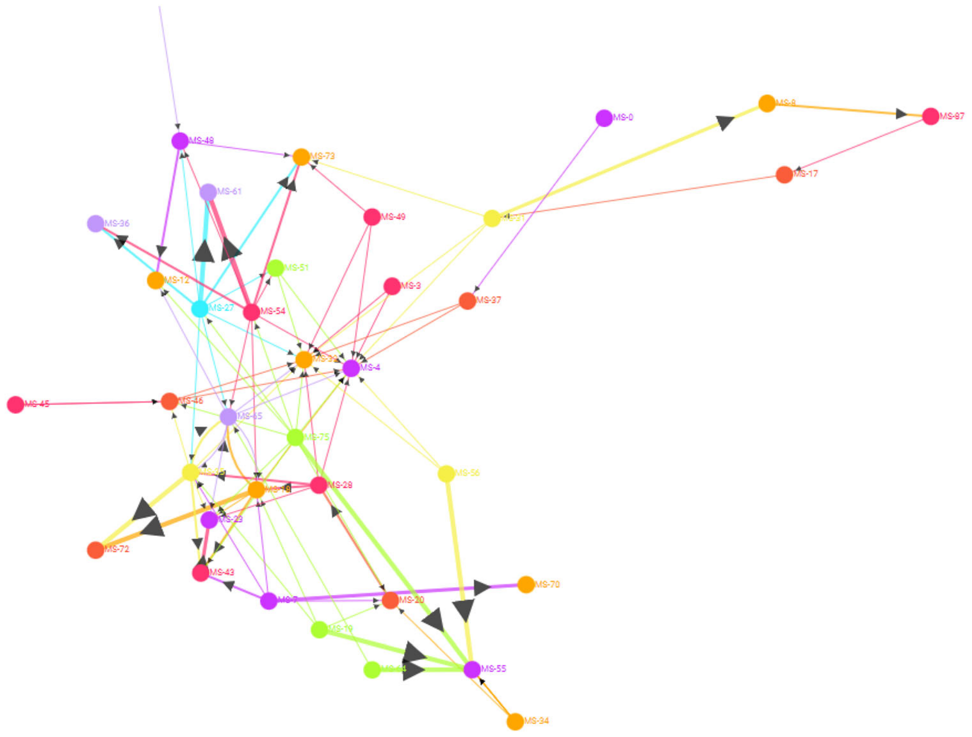


Fig. 14 2D service view interactive visualization showing TrainTicket benchmark with anonymized microservice names

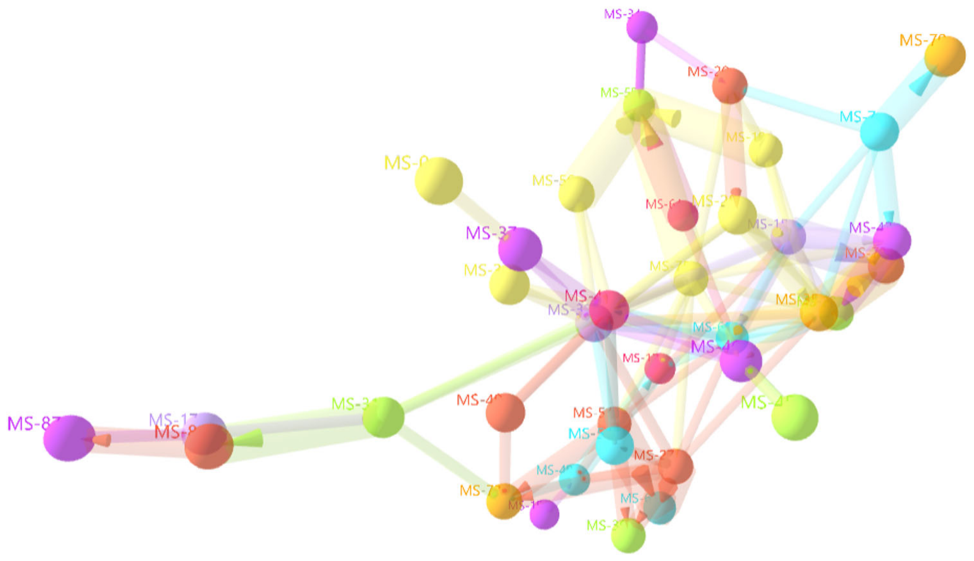


Fig. 15 3D service view interactive visualization showing TrainTicket benchmark with anonymized microservice names

interconnections but also effectively overcomes the limitations of the AR mechanism to search and filter mentioned in the previous section.

6.2.1 Preliminary assessment and challenges

The introduction of these web-based models is derived from the need to address challenges posed by both AR and traditional visualization approaches. These web-based models offer practical features that address the requirements identified in previous studies for practitioners.

Moreover, we are conducting a study and share our preliminary evaluation results of these web-based interactive models. In a user study, we considered a similar setting to the previous assessment with the same microservice system benchmark and its intermediate representation of the system. This allowed us to populate both prototypes. In total, 24 practitioners were given two system sizes to assess randomly (complete and a subset system) for 2D and 3D tasks with anonymized microservice names and asked to identify various dependencies.

Practitioners were able to accomplish tasks with similar correctness with both models, while the 2D solution enabled them to accomplish tasks faster. Their perception was to recommend the 2D solution, possibly given their familiarity with 2D models.

Practitioners highlighted the benefits of possible node rearrangement as a very positive usability feature. However, they also mentioned that with the rearrangement, other nodes repositioned, which was not appreciated. Practitioners noted the benefits of search, filtering, and tracking. The on-demand access to information was positively rated. However, the color codes of nodes and dependencies would sometimes end up with light colors that were hard to read; they pointed out that zooming in a node detail could be disorienting. Also, they observed that zooming and panning were not necessary for their tasks. They highlighted that arrows of dependencies should be more significant than small.

6.3 Other considerations on visualization

In other instances, a different approach of a context map visualization prototype was implemented.[10] The scope of the context might be too broad, and a sub-context map might be a better approach to limit the necessary detail. The idea of this approach is that the user selects a subset of microservices she/he is interested in, and details only related to these services are rendered. As an example, consider that an architect calls in five teams that need to meet and discuss system extension, and they select five relevant microservices to render their details. The sub-context view then renders the entities of the five selected microservices and combines them to get a relevant perspective. This aids experts in grasping the interrelations between selected microservices, their selected neighbors, and the intertwining of data models with dependencies. Such a feature proves advantageous, especially in scenarios where the given system encompasses a vast array of entities and microservices. A representative example of a sub-context map can be observed in Fig. 16.

Using such models renders more convenience for additional extensions that can augment the user experience or attention to detail. Detecting anti-patterns is a known approach to mitigate architectural degradation [79]. Using the service view of a large system to identify anti-patterns manually would be a difficult task. At the same time, using an automated approach at the microservice level, it might still be difficult to explain what the anti-pattern relates to. In order to expedite this task, we implemented a proof-of-concept[11] that extends the interactive 3D visualization with

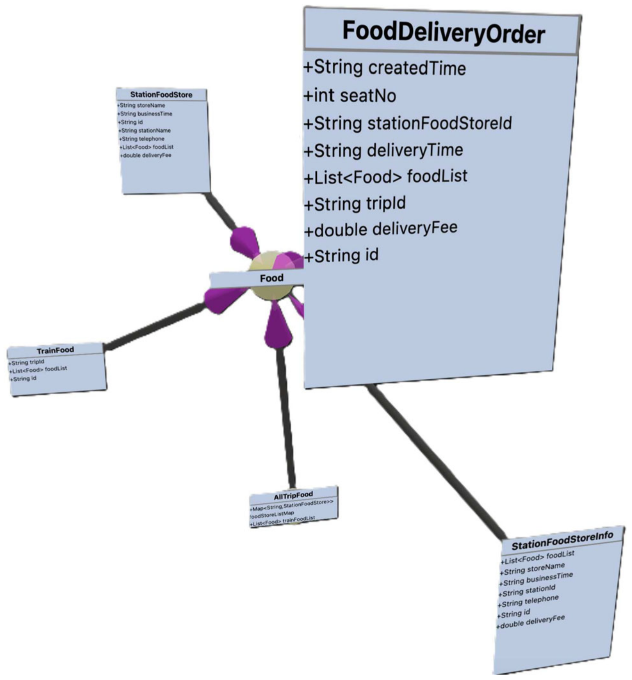


Fig. 16 Sub-context map for Train-ticket benchmark showing selected data entities

the possibility of highlighting a subsection of the service view in which anti-patterns that occur according to user-defined thresholds. High-coupling microservices, cyclic dependencies in the inter-service call, and bottleneck microservices can be identified. Figure 17 shows an example of cycle dependency anti-pattern highlighted.

7 Discussion and research questions answers

The development of the research has been guided by two main research questions presented at the beginning of this article. These questions are a result of the limitations that we find in the state of the art regarding the analysis of microservice architectures. Our results are nothing more than a research attempt to solve these limitations.

In our study, we utilized two architectural views reconstructed from the microservice system codebase using static analysis. These views provide a system-centric view and serve as up-to-date and aligned documentation facilitating observation of system evolution, addressing major challenges indicated by Bogner et al. [6]. It is important to note that architectural views are not the only valuable product of the research project. The reconstruction process produces a system-holistic intermediate representation suitable for broader system reasoning. This representation can facilitate the integration of decentralized microservices, enable verification, or serve as a communication

[10] https://github.com/cloudhubs/graal_mvp

[11] https://github.com/cloudhubs/mvp

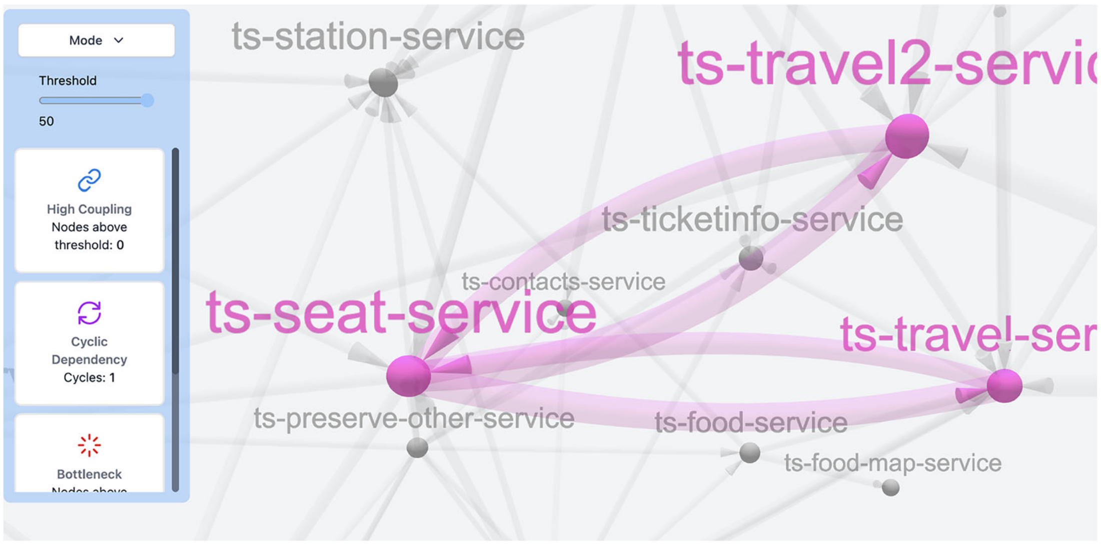


Fig. 17 3D service view interactive visualization with anti-pattern detection. In this case a cyclic dependency is highlighted

medium between stakeholders such as architects, developers, and DevOps.

Our research employs the SAR approach along with a basic prototype to demonstrate the ability to use static analysis to attain a comprehensive and updated view of a system and its documentation effectively **addressing RQ1**.

In the extraction phase of the SAR methodology, information is gathered from each microservice using static code analysis. Following this, the construction phase compiles the extracted information and integrates additional data from the component definition layer. Various Component Call Graphs (CCGs) are created at this stage, each containing detailed information about individual microservice components. External calls connected to other microservice endpoints are identified based on their method signatures, providing initial insights into system dependencies.

The subsequent manipulation phase consolidates all CCGs into a single graph that represents the entire system. This phase unifies all contexts into a comprehensive data model, offering an exhaustive view of the system and its interconnected elements.

Our tool seamlessly moves through these phases, generating a detailed JSON intermediate representation of the system. For real-world validation, a third-party system benchmark was applied to our developed prototype tool. The results allowed the creation of diverse visualizations that clearly depicted the complete details of the system and its dependencies.

In order to **answer RQ2**, we implemented proof-of-concept tools for SAR of microservice systems and demonstrated that various visual models can be derived related to the state-of-the-art opportunities for architecture visualization. These views might be subject to further analysis with regard to efficiency in serving practitioners in their tasks. With any successful SAR process, no matter the method, whether performed manually, statically, or dynamically, in the context of microservices, the next logical question to ask would relate to appropriate visualization formats.

While the domain of large data visualization holds vast potential, practitioners often have specific expectations. We intended to use the intermediate representation formed in the prior phase as input for diverse visualization tools. Numerous visualizations representing the system from unique perspectives are available, each serving different purposes and users and each having distinct benefits and limitations.

It’s crucial to develop visualizations that aptly represent the system’s diverse aspects for various users involved in the microservice architecture, such as customers, stakeholders, DevOps, and developers, among others. Another significant research area is defining a standard for the intermediate representation of a microservice architecture. This task is challenging as this representation must encompass extensive information about the system to support the creation of detailed and informative visualizations.

Promising direction to the microservice needs to bring the approach of multiple coordinated views [54, 55]. In our case, the service view could be interconnected with the domain view. For instance, the selections of microservices in the service view could directly plot the subcontext map in the domain view, and likely, more opportunities exist when moving in time, which is well suited for visual models incorporating evolution (i.e., provenance tracking [56])

Addressing these challenges and establishing a widely accepted standard will greatly enhance the efficiency and clarity of visualizations, allowing for a more unified and focused approach to visualizing microservice architecture.

Specific models might be designed for a particular quality or system perspective. Architectural views might need to be migrated into multi-modal and contextual perspectives, letting the user navigate into the necessary detail or select a specific perspective. At this point, we did not assess any dynamic perspective in which the system operates; however, hypothesizing another dimension—the dimension of time—in the architectural visualization is inevitable. An architectural view that includes time will be able to show the system evolution changes, deviation from the original goals, and the changing characteristics of the dynamic system.

Various important questions remain concerning the visualization of code changes, performance statistics, or economic impacts. All these questions will drive further innovation. Even though we did not address these challenges, we believe that our work provides the fundamental cornerstone to start the discussion at the academic and industrial level since better diagnostics will facilitate system maintenance and sustainability and make broad spectra of system assessments much easier aka security analysis or root cause analysis.

7.1 Limitation

In discussing the findings of our research, it's essential to acknowledge the limitations that accompanied the study. Recognizing these limitations not only provides transparency but also guides future research endeavors by highlighting areas for further exploration and improvement. In this section, we will delineate the specific limitations encountered during our research.

In the first place, our prototype tool is restricted to microservices written in Java using the Spring Boot framework. It will be recommended that its capability expand to support various programming languages. Additionally, our tool cannot detect event-driven communication between microservices, like connections using JMS, Kafka, among others, which are widely used.

Our proof-of-concept is also limited to generating only two architectural views of the system - the service view and the domain view. The visualizations tested were solely based on these two views. Enabling the representation and visualization of diverse views would offer a more multi-faceted analysis of the system.

Furthermore, we considered microservice dependencies, which often change with evolution. Thus, their observability and management are important to prevent bottlenecks and thus mitigate architectural degradation. However, the study did not examine the broader evolution aspect of microservice architecture, a crucial element in identifying software architecture degradation. It's essential to infuse time into our prototype, allowing it to represent the system's evolution.

We tested various visualizations based on an intermediate representation from static code analysis. However, by developing a tool that performs dynamic analysis while maintaining the same JSON intermediate representation structure, all existing visualizations could be seamlessly utilized. This enhancement would allow for a more flexible and extensive analysis of the system.

Lastly, while we compared 2D, 3D, and AR visualizations, we did not explore visualizations utilizing virtual reality, leaving room for further exploration in this domain.

8 Conclusions

Cloud-native systems are currently used by a broad number of companies. Yet, they all must deal with the same challenges that remain open for such systems. The challenge considered in this work is related to the missing system-centric view. We elaborated on it in the context of SAR performed on the decentralization nature of these systems using static analysis. While such a process has various goals, we target system visualization as an instrument to engage practitioners in a "beyond their microservice" view to help with an illustration of microservice dependencies in the overall system.

We illustrate that static analysis can aid with the extraction of various architectural views and bring the benefits of never-outdated documentation and quick instruments aiding informed decisions. The specifics of cloud-native systems were elaborated on in the context of architectural perspective visualization and illustrated using data from established third-party system testbench. Multiple tool prototypes were generated and offered to the community in an open-source format for extensions and experimentation. The tools provide more insights into the needs for successful visualizations, such as coping with large numbers of microservices or interactive features while not omitting practitioners' expectations (i.e., search features or integration with their workstation),

With the illustrations in this work, we motivate practitioners in alternative methods that could aid in interpreting microservice system details, not requiring complex assessment of the source code or system execution to generate traces. However, it must be noted that we only provide a proof of concept, and production-ready solutions will need to deal with platform heterogeneities.

In future work, we aim to create a detailed intermediate representation of the system that includes time for evolution analysis. We will also explore generating this representation through dynamic analysis and compare its visualization results. Plans include integrating other dependencies, such as Event-Driven (i.e., messaging systems), and further experimenting with hierarchical and interactive visualization.

Author contributions Tomas Cerny: Formal analysis and investigation, Methodology, Supervision, Validation, Writing- Original draft preparation, Writing- Revision. Amr S. Abdelfattah: Resources, Formal analysis, and investigation, Data curation, Validation, Writing- Original draft preparation, Writing-Revision. Jorge Yero: Resources, Conceptualization, Methodology, Software, Data curation, Formal analysis and investigation, Writing- Original draft preparation. Davide Taibi: Validation, Writing- Revision.

Funding This material is based upon work supported by the National Science Foundation under Grant No. 2245287 and a grant from Red Hat Research.

Data availability We have shared our open-source tools Prophet, our tool that is able to extract the intermediate representation can be found at GitHub https://github.com/cloudhubs/prophet-utils,https://github.com/cloudhubs/prophet-utils-app,https://github.com/cloudhubs/prophet. The source code of the different visualization approaches utilized can be found at https://github.com/cloudhubs/mvp, https://github.com/cloudhubs/graal_mvp, https://github.com/cloudhubs/ArchitectureVisualizationPOC, https://github.com/cloudhubs/microvision, and https://github.com/cloudhubs/prophet-web.

Declarations

Conflict of interest On behalf of all authors, the corresponding author states that there is no Conflict of interest.

Research Involving Human and /or Animals Not applicable

Informed Consent Not applicable

References

1. Cerny, T., Donahoo, M.J., Trnka, M.: Contextual understanding of microservice architecture: current and future directions. SIGAPP Appl. Comput. Rev. **17**(4), 29–45 (2018). https://doi.org/10.1145/3183628.3183631
2. Bushong, V., Abdelfattah, A.S., Maruf, A.A., Das, D., Lehman, A., Jaroszewski, E., Coffey, M., Cerny, T., Frajtak, K., Tisnovsky, P., et al.: On microservice analysis and architecture evolution: a systematic mapping study. Appl. Sci. **11**(17), 7856 (2021)
3. Amoroso d'Aragona, D., Li, X., Cerny, T., Janes, A., Lenarduzzi, V., Taibi, D.: One microservice per developer: is this the trend in OSS? In: Papadopoulos, G.A., Rademacher, F., Soldani, J. (eds.) Service-Oriented and Cloud Computing, pp. 19–34. Springer, Cham (2023)
4. Richardson, C.: Pattern: microservices architecture (2014). http://microservices.io/patterns/microservices.html
5. Abdelfattah, A.S., Cerny, T.: The microservice dependency matrix. In: Papadopoulos, G.A., Rademacher, F., Soldani, J. (eds.) Service-Oriented and Cloud Computing, pp. 276–288. Springer, Cham (2023)
6. Bogner, J., Fritzsch, J., Wagner, S., Zimmermann, A.: Industry practices and challenges for the evolvability assurance of microservices. Empir. Softw. Eng. **26**(5), 104 (2021). https://doi.org/10.1007/s10664-021-09999-9
7. Soldani, J., Tamburri, D.A., Van Den Heuvel, W.-J.: The pains and gains of microservices: a systematic grey literature review. J. Syst. Softw. **146**, 215–232 (2018). https://doi.org/10.1016/j.jss.2018.09.082
8. Parker, G., Kim, S., Maruf, A.A., Cerny, T., Frajtak, K., Tisnovsky, P., Taibi, D.: Visualizing anti-patterns in microservices at runtime: a systematic mapping study. IEEE Access **11**, 4434–4442 (2023). https://doi.org/10.1109/ACCESS.2023.3236165
9. Cerny, T., Abdelfattah, A.S., Bushong, V., Al Maruf, A., Taibi, D.: Microservice architecture reconstruction and visualization techniques: a review. In: 2022 IEEE International Conference on Service-Oriented System Engineering (SOSE), pp. 39–48 (2022). IEEE
10. Gortney, M.E., Harris, P.E., Cerny, T., Al Maruf, A., Bures, M., Taibi, D., Tisnovsky, P.: Visualizing microservice architecture in the dynamic perspective: a systematic mapping study. IEEE Access (2022)
11. Cerny, T., Abdelfattah, A.S., Bushong, V., Al Maruf, A., Taibi, D.: Microvision: Static analysis-based approach to visualizing microservices in augmented reality. In: 2022 IEEE International Conference on Service-Oriented System Engineering (SOSE), pp. 49–58 (2022). IEEE
12. Bushong, V., Das, D., Al Maruf, A., Cerny, T.: Using static analysis to address microservice architecture reconstruction. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1199–1201 (2021). IEEE
13. Schiewe, M., Curtis, J.B., Bushong, V., Cerny, T.: Advancing static code analysis with language-agnostic component identification. IEEE Access (2022). https://doi.org/10.1109/ACCESS.2022.3160485
14. Carnell, J., Sánchez, I.H.: Spring Microservices in Action, p. 448. Manning Publications Co., 2nd ed. Shelter Island, NY, USA (2021). https://www.manning.com/books/spring-microservices-in-action-second-edition
15. O'Brien, L., Stoermer, C., Verhoef, C.: Software architecture reconstruction: practice needs and current approaches. Technical report, Carnegie Mellon University (2002). https://doi.org/10.1184/R1/6583982.v1
16. Rademacher, F., Sachweh, S., Zündorf, A.: A modeling method for systematic architecture reconstruction of microservice-based software systems. In: Nurcan, S., Reinhartz-Berger, I., Soffer, P., Zdravkovic, J. (eds.) Enterprise, Business-Process and Information Systems Modeling, pp. 311–326. Springer, Cham (2020)
17. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Softw. Eng. Notes **17**(4), 40–52 (1992). https://doi.org/10.1145/141874.141884
18. Software, systems and enterprise—architecture description. Standard, International Organization for Standardization, Geneva, CH (2022)
19. Wiggins, A.: The twelve-factor app (2017). https://12factor.net/. Accessed 10 Feb 2021
20. Abdelfattah, A.S., Cerny, T.: Roadmap to reasoning in microservice systems: a rapid review. Appl. Sci. **13**(3), 1838 (2023)
21. Al Maruf, A., Bakhtin, A., Cerny, T., Taibi, D.: Using microservice telemetry data for system dynamic analysis. In: 2022 IEEE International Conference on Service-Oriented System

Engineering (SOSE), pp. 29–38 (2022). https://doi.org/10.1109/SOSE55356.2022.00010
22. Elsayed, A., Cerny, T., Salazar, J.Y., Lehman, A., Hunter, J., Bickham, A., Taibi, D.: End-to-end test coverage metrics in microservice systems: an automated approach (2023). arXiv:2308.09257
23. Chlipala, A.: The bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. SIGPLAN Not. **48**(9), 391–402 (2013). https://doi.org/10.1145/2544174.2500592
24. Albert, E., Gómez-Zamalloa, M., Hubert, L., Puebla, G.: Verification of java bytecode using analysis and transformation of logic programs. In: Hanus, M. (ed.) Practical Aspects of Declarative Languages, pp. 124–139. Springer, Berlin (2007)
25. Cho, H.: Using metaprogramming to implement a testing framework. In: Proceedings of the 47th Annual Southeast Regional Conference. ACM-SE 47, pp. 55–1552. ACM, New York, NY, USA (2009). https://doi.org/10.1145/1566445.1566519
26. Tonella, P.: Evolutionary testing of classes. SIGSOFT Softw. Eng. Notes **29**(4), 119–128 (2004). https://doi.org/10.1145/1013886.1007528
27. Papotti, P.E., Prado, A.F., Souza, W.L.: Reducing time and effort in legacy systems reengineering to MDD using metaprogramming. In: Proceedings of the 2012 ACM Research in Applied Computation Symposium. RACS '12, pp. 348–355. Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2401603.2401681
28. Keivanloo, I., Roy, C.K., Rilling, J.: Sebyte: Scalable clone and similarity search for bytecode. Sci. Comput. Program. **95**, 426–444 (2014). https://doi.org/10.1016/j.scico.2013.10.006 . Special Issue on Software Clones (IWSC'12)
29. Keivanloo, I., Roy, C.K., Rilling, J.: Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning. In: Proceedings of the 6th International Workshop on Software Clones. IWSC '12, pp. 36–42. IEEE Press, Piscataway, NJ, USA (2012). http://dl.acm.org/citation.cfm?id=2664398.2664404
30. Rattan, D., Bhatia, R., Singh, M.: Software clone detection: a systematic review. Inf. Softw. Technol. **55**(7), 1165–1199 (2013). https://doi.org/10.1016/j.infsof.2013.01.008
31. Li, X., Chen, Y., Lin, Z.: Towards automated inter-service authorization for microservice applications. In: Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos. SIGCOMM Posters and Demos '19, pp. 3–5. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3342280.3342288
32. Eski, S., Buzluca, F.: An automatic extraction approach: transition to microservices architecture from monolithic application. In: Proceedings of the 19th International Conference on Agile Software Development: Companion. XP '18. Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3234152.3234195
33. Esparrachiari, S., Reilly, T., Rentz, A.: Tracking and controlling microservice dependencies. Queue **16**(4), 10–441065 (2018). https://doi.org/10.1145/3277539.3277541
34. Pigazzini, I., Fontana, F.A., Lenarduzzi, V., Taibi, D.: Towards microservice smells detection. In: Proceedings of the 3rd International Conference on Technical Debt. TechDebt '20, pp. 92–97. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3387906.3388625
35. Rahman, M.I., Panichella, S., Taibi, D.: A curated dataset of microservices-based systems. In: SSSME-2019 (2019)
36. Ibrahim, A., Bozhinoski, S., Pretschner, A.: Attack graph generation for microservice architecture. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. SAC '19, pp. 1235–1242. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3297280.3297401
37. Mayer, B., Weinreich, R.: An approach to extract the architecture of microservice-based software systems. In: 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), pp. 21–30 (2018)
38. Salvadori, I., Huf, A., Mello, R.D.S., Siqueira, F.: Publishing linked data through semantic microservices composition. In: Proceedings of the 18th International Conference on Information Integration and Web-Based Applications and Services. iiWAS '16, pp. 443–452. Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/3011141.3011155
39. Kruchten, P.: Architectural blueprints: the 4+1 view model of software architecture (1995). CoRR arXiv:2006.04975
40. Kruchten, P.: The 4+1 view model of architecture. IEEE Softw. **12**(6), 42–50 (1995). https://doi.org/10.1109/52.469759
41. Walker, A., Laird, I., Cerny, T.: On automatic software architecture reconstruction of microservice applications. In: Information Science and Applications: Proceedings of ICISA 2020, vol. 739, p. 223 (2021)
42. Fowler, M.: Bounded context (2014). https://martinfowler.com/bliki/BoundedContext.html. Accessed 10 Feb 2021
43. Evans, E., Evans, E.J.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, Boston (2004)
44. Vernon, V.: Implementing Domain-Driven Design. Addison-Wesley, Boston (2013)
45. Sun, C.-A.: A multi-view architectural model and its description and construction. In: 2010 International Conference on Computational Intelligence and Software Engineering, pp. 1–5 (2010). https://doi.org/10.1109/CISE.2010.5676834
46. Zhou, Z., Zhi, Q., Morisaki, S., Yamamoto, S.: A systematic literature review on enterprise architecture visualization methodologies. IEEE Access **8**, 96404–96427 (2020). https://doi.org/10.1109/ACCESS.2020.2995850
47. Vázquez-Ingelmo, A., García-Holgado, A., García-Peñalvo, F.J.: C4 model in a software engineering subject to ease the comprehension of uml and the software. In: 2020 IEEE Global Engineering Education Conference (EDUCON), pp. 919–924 (2020). https://doi.org/10.1109/EDUCON45650.2020.9125335
48. Shahin, M., Liang, P., Babar, M.A.: A systematic review of software architecture visualization techniques. J. Syst. Softw. **94**, 161–185 (2014)
49. Wettel, R., Lanza, M.: Visually localizing design problems with disharmony maps. In: Proceedings of the 4th ACM Symposium on Software Visualization. In: SoftVis '08, pp. 155–164. Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1409720.1409745
50. Abdelfattah, A.S., Cerny, T.: The Microservice Dependency Matrix. Springer, Cham (2023)
51. Fittkau, F., Krause, A., Hasselbring, W.: Exploring software cities in virtual reality. In: 2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT), pp. 130–134 (2015)
52. Schreiber, A., Nafeie, L., Baranowski, A., Seipel, P., Misiak, M.: Visualization of software architectures in virtual reality and augmented reality. In: 2019 IEEE Aerospace Conference, pp. 1–12 (2019)
53. Steinbeck, M., Koschke, R., Rüdel, M.O.: How evostreets are observed in three-dimensional and virtual reality environments. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 332–343 (2020)
54. Andrienko, G., Andrienko, N.: Coordinated multiple views: a critical view. In: Fifth International Conference on Coordinated

and Multiple Views in Exploratory Visualization (CMV 2007), pp. 72–74 (2007). https://doi.org/10.1109/CMV.2007.4
55. Boukhelifa, N., Roberts, J.C., Rodgers, P.J.: A coordination model for exploratory multiview visualization. In: Proceedings International Conference on Coordinated and Multiple Views in Exploratory Visualization - CMV 2003 -, pp. 76–85 (2003). https://doi.org/10.1109/CMV.2003.1215005
56. Burgess, K., Hart, D., Elsayed, A., Cerny, T., Bures, M., Tisnovsky, P.: Visualizing architectural evolution via provenance tracking: a systematic review. In: Proceedings of the Conference on Research in Adaptive and Convergent Systems, pp. 83–91 (2022)
57. Oberhauser, R., Pogolski, C.: VR-EA: virtual reality visualization of enterprise architecture models with ArchiMate and BPMN. In: Shishkov, B. (ed.) Business Modeling and Software Design vol. 356, pp. 170–187. Springer, Cham (2019). Series Title: Lecture Notes in Business Information Processing
58. Ma, Z., Bai, Y.: A distributed system monitoring tool with virtual reality. In: Proceedings of the 2nd International Conference on Computer Science and Application Engineering. CSAE '18. Association for Computing Machinery, New York, NY, USA (2018)
59. Toumpalidis, I., Cheliotis, K., Roumpani, F., Smith, A.: VR binoculars: an immersive visualization framework for IoT data streams. In: Proceedings of the IEEE Living in the Internet of Things: Cybersecurity of the IoT (2017)
60. Halpin, H., Zielinski, D.J., Brady, R., Kelly, G.: Exploring semantic social networks using virtual reality. In: Sheth, A., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T., Thirunarayan, K. (eds.) The Semantic Web—ISWC 2008, pp. 599–614. Springer, Berlin (2008)
61. Royston, S., DeFanti, C., Perlin, K.: A collaborative untethered virtual reality environment for interactive social network visualization (2016). CoRR arXiv:1604.08239
62. Moreno-Lumbreras, D., Robles, G., Izquierdo-Cortázar, D., Gonzalez-Barahona, J.M.: To VR or not to VR: Is virtual reality suitable to understand software development metrics? (2021). arxiv:2109.13768
63. Beck, F., Burch, M., Diehl, S., Weiskopf, D.: A taxonomy and survey of dynamic graph visualization. In: Computer Graphics Forum, vol. 36, pp. 133–159 (2017). Wiley, New York
64. Burch, M., Huang, W., Wakefield, M., Purchase, H.C., Weiskopf, D., Hua, J.: The state of the art in empirical user evaluation of graph visualizations. IEEE Access **9**, 4173–4198 (2020)
65. Hopkins, W.: JSR 375: JavaTM EE Security API (2009).https://jcp.org/en/jsr/detail?id=375
66. Das, D., Walker, A., Bushong, V., Svacina, J., Cerny, T., Matyas, V.: On automated RBAC assessment by constructing a centralized perspective for microservice mesh. PeerJ Comput. Sci. **7**, 376 (2021)
67. Abdelfattah, A.S., Rodriguez, A., Walker, A., Cerny, T.: Detecting semantic clones in microservices using components. SN Comput. Sci. **4**(5), 470 (2023)
68. Han, L., Kashyap, L.: A., Finin, T., Mayfield, J., Weese, J.: 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, pp. 44–52. Association for Computational Linguistics, Atlanta, Georgia, USA (2013)
69. Walker, A., Das, D., Cerny, T.: Automated code-smell detection in microservices through static analysis: a case study. Appl. Sci. (2020). https://doi.org/10.3390/app10217800
70. Taibi, D., Lenarduzzi, V., Pahl, C.: Architectural patterns for microservices: a systematic mapping study. In: CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018 (2018). SciTePress
71. Mayer, B., Weinreich, R.: A dashboard for microservice monitoring and management. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pp. 66–69 (2017). https://doi.org/10.1109/ICSAW.2017.44
72. Černý, T., Donahoo, M., Trnka, M.: Contextual understanding of microservice architecture: current and future directions. ACM SIGAPP Appl. Comput. Rev. **17**, 29–45 (2018)
73. Baabad, A., Zulzalil, H.B., Hassan, S., Baharom, S.B.: Software architecture degradation in open source software: a systematic literature review. IEEE Access **8**, 173681–173709 (2020). https://doi.org/10.1109/ACCESS.2020.3024671
74. Alshuqayran, N., Ali, N., Evans, R.: A systematic mapping study in microservice architecture. In: 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA), pp. 44–51 (2016). https://doi.org/10.1109/SOCA.2016.15
75. Zhou, X., Peng, X., Xie, T., Sun, J., Xu, C., Ji, C., Zhao, W.: Benchmarking microservice systems for software engineering research. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, pp. 323–324 (2018)
76. Li, B., Peng, X., Xiang, Q., Wang, H., Xie, T., Sun, J., Liu, X.: Enjoy your observability: an industrial survey of microservice tracing and analysis. Empir. Softw. Eng. **27** (2022) https://doi.org/10.1007/s10664-021-10063-9
77. Zhou, X., Peng, X., Xie, T., Sun, J., Li, W., Ji, C., Ding, D.: Delta debugging microservice systems. In: Huchard, M., Kästner, C., Fraser, G. (eds.) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018, pp. 802–807. ACM, ??? (2018). https://doi.org/10.1145/3238147.3240730
78. Abdelfattah, A.S., Cerny, T., Taibi, D., Vegas, S.: Comparing 2d and augmented reality visualizations for microservice system understandability: a controlled experiment. In: 2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC), pp. 135–145 (2023). https://doi.org/10.1109/ICPC58990.2023.00028
79. Cerny, T., Abdelfattah, A.S., Maruf, A.A., Janes, A., Taibi, D.: Catalog and detection techniques of microservice anti-patterns and bad smells: a tertiary study. J. Syst. Softw. **206**, 111829 (2023). https://doi.org/10.1016/j.jss.2023.111829

Tomas Cerny is an Associate Professor of Systems and Industrial Engineering at the University of Arizona. His area of research is software engineering, cloud systems, and code analysis. He received his Master's, and Ph.D. degrees from the Faculty of Electrical Engineering at the Czech Technical University in Prague, and an M.S. degree from Baylor University. He started his academic career in 2009 at the Czech Technical University, FEE, from where he transferred to Baylor University in 2017 and to the University of Arizona in 2023. Dr. Cerny served 15+ years as the lead developer of the International Collegiate Programming Contest Management System. He authored over 160 peer-reviewed publications, mostly related to code analysis and microservices. Among his awards are best papers at Microservices 2022/2023, IEEE SOSE 2022, Closer 2022, LXNLP 2022, the Outstanding Service Award ACM SIGAPP 2018 and 2015, and the 2011 ICPC Joseph S. DeBlasi Outstanding Contribution Award. He serves on the committee of various conferences including program or conference chairs at ACM SAC, Microservices, SANER, ISD, SOSE, ACM RACS, and ESOCC.

Amr S. Abdelfattah earned his Ph.D. in Computer Science in 2024 from Baylor University in the USA, focusing his research on Software Engineering, Mobile Cloud Computing, and Microservice Architecture. He is an esteemed member of both the scientific research honor society, Sigma Xi, and the academic excellence honor society, Upsilon Pi Epsilon. He completed his Bachelor's and Master's degrees in Computer and Information Science at Ain Shams University in Egypt, before pursuing his Master's and Ph.D. degrees in Computer Science at Baylor University in the USA. With over nine years of experience as a mobile technical lead for renowned international companies, he has honed his expertise in the tech industry. He is also a certified Professional Scrum Master, demonstrating his commitment to ongoing professional development. Driven by a desire to make meaningful contributions to the field, He aims to establish himself as a leader in the tech industry while continuing to excel as an educator and researcher.

Jorge Yero received the B.S. degree in La Universidad de La Habana, Cuba, in 2020. Currently, he is pursuing his Ph.D. at Baylor University, USA. Between 2022 and 2023, he was a Research Assistant with the Department of Computer Science at Baylor University. Experience in Software Engineering, DevOps, and Business intelligence roles in industry. His research interests include code analysis, microservice architecture, NLP and individual recognition.

Davide Taibi is a full Professor at the University of Oulu (Finland), where he heads the M3S Cloud research group. His research is mainly focused on Empirical Software Engineering applied to cloud-native systems, with a special focus on the migration from monolithic to cloud-native applications. He is investigating processes, and techniques for developing Cloud Native applications, identifying cloud-native specific patterns and anti-patterns. He has been a member of the International Software Engineering Network (ISERN) since 2018. Before moving to Finland, he was been Assistant Professor at the Free University of Bozen/Bolzano (2015-2017), a post-doctoral research fellow at the Technical University of Kaiserslautern and Fraunhofer Institute for Experimental Software Engineering - IESE (2013–2014), and research fellow at the University of Insubria (2007–2011).