Microservice-Aware Static Analysis: Opportunities, Gaps, and Advancements

Tomas Cerny
Systems and Industrial Engineering, University of Arizona, Tucson, AZ, USA

Davide Taibi
Oulu University, Finland

Abstract

Microservice architecture is the mainstream to fuel cloud-native systems with small service sets developed and deployed independently. The independent nature of this modular architecture also leads to challenges and gaps practitioners did not face in system monoliths. One of the major challenges with decentralization and its independent microservices that are managed by separate teams is that the evolving system architecture easily deviates far from the original plans, and it becomes difficult to maintain. Literature often refers to this process as system architecture degradation. Especially in the context of microservices, available tools are limited. This article challenges the audience on how static analysis could contribute to microservice system development and management, particularly managing architectural degradation. It elaborates on challenges and needed changes in the traditional code analysis to better fit these systems. Consequently, it discusses implications for practitioners once robust static analysis tools become available.

2012 ACM Subject Classification Computer systems organization → Distributed architectures; Computer systems organization → Cloud computing; Software and its engineering → Automated static analysis

Keywords and phrases Microservice Architecture, Static Analysis, Reasoning, Decentralization

Digital Object Identifier 10.4230/OASIcs.Microservices.2020-2022.2

Funding *Tomas Cerny*: This material is based upon work supported by the National Science Foundation under Grant No. 2245287, and grant from Red Hat Research `https://research.redhat.com`.
Davide Taibi: This material is based upon work supported by the ADOMS Grant from Ulla Tuominen Foundation and a grant from the Academy of Finland (grant n. 349488 - MuFAno).

1 Introduction

Cloud-native systems are designed to take full advantage of the cloud infrastructure. The 12-factor app methodology [46] provides guidance on designing, building and deploying these systems. For instance, cloud-native systems have, as the foundation, the microservice architecture, utilize containers, and follow Continuous Integration and Delivery (CD/CI).

The microservice architectural style is a paradigm for developing systems as a suite of small, self-contained, and autonomous services communicating through a lightweight protocol. Each microservice has its own codebase with a separate configuration to facilitate its individual decentralized evolution. This, as a result, enables the separation of duties for roles like architects, developers, and DevOps. It also aligns well with Conways's law, stating that organizations should design systems to mirror their own communication structure.

While functionality gets well divided with microservices, overlaps across individual microservice' still exist. Each microservice has a bounded context within the overall system, and still, since microservices interact, the overlap is inevitable. Implications from these overlaps are reflected in the microservice codebase. Since the microservice codebase remains


© Tomas Cerny and Davide Taibi;
licensed under Creative Commons License CC-BY 4.0

self-contained, overlaps mean partially restated definitions, typically re-implemented in a particular framework version. This restatement can relate to data definitions of processed information, encapsulated knowledge, business logic, or other enforcement related to various policies (i.e., security, constraints, privacy, etc.). However, this overlap is uncontrolled throughout the system evolution, and there are fragile mechanisms to assess consistency errors. Thus, once any of these definitions change in the microservice codebase, there is no direct indication of the definition being restated elsewhere in other codebases.

In addition to functional decomposition, one would expect microservices to cope with the separation of concerns. While there are the same means as in any other component-based development, decentralization leads to the scattering of different concerns, lacking a single focal point. Such concern separation might be well-managed on a single codebase level, but it might get lost with the decentralization and the existence of multiple codebases. While infrastructure like centralized configuration servers and API gateway exists, i.e., to enable telemetry and tracing, these cannot be misused beyond their original purpose, leading to anti-patterns.

We typically aim to separate concerns in software systems to provide better readability and maintainability [29]. We can do a micro-management and design solid concern separation per each microservice. However, this would only relate to a single microservice, not the whole system. The question is whether we need to see a certain concern from the entire system perspective. Suppose we are architects; most likely, the answer is yes. For instance, to focus on system privacy concerns. To make informed decisions, developers must see dependent (i.e., interacting) microservices aligned across selected concerns. We must keep in mind with the self-contained microservice nature and the decentralized system perspective, each concern of a certain type is re-defined and encapsulated across microservices. This might be one of the greatest disappointments when migrating from monolith systems. As an example, the consequence of security assessment is that each microservice has to be analyzed individually. Then, the extracted knowledge must be combined ad-hoc, which is tedious, time-consuming, error-prone, and does not scale with agile development. Unfortunately, with microservice architecture, we must accept that the system must deal with "*scattered concerns*" [9].

In this article, we discuss how static analysis could contribute to solving the shortcomings of microservices-based systems. We emphasize how future tools should adapt to better fit these systems' specifics. We base our discussion on case studies and prototype tools we developed with our research teams.

In the remainder of this paper, we discuss the current approaches to assess cloud-native systems (Section 2). Next, Section 3 focuses on changes to static analysis tools to better align with cloud-native. Finally, Section 4 discusses the implications and impact on involved stakeholders once these tools become robust and available, while Section 5 concludes the paper.

2 Current Trends

Researchers often resort to applying dynamic system analysis to address various microservice challenges. The dynamic analysis can undoubtedly uncover service dependency graphs and bring them a more centric system view.

However, to uncover these artifacts, we need to invest in different efforts. First, the uncovered artifacts can only be as complete as the underlying systems tests or system interaction. This means that we could extract a complete graph if we had complete test coverage [18, 38]. However, complete test coverage is expensive, and it must adapt to system

evolution. Alternatively, we could use production system traffic, but even then, there is no guarantee of complete system coverage. Second, we do not want customers to identify, i.e., cyclic dependency in production, and should target system analysis before it ships to production. The dynamic analysis will need the system to run to perform interaction to uncover the previously mentioned artifacts, which is time-consuming. We would need a lot of computational power if we anticipate analyzing the system for every new code change (commit) in the codebase. It also takes time to perform the tests (especially with full coverage).

Dynamic analysis can be performed by system monitoring (i.e., with telemetry `https://opentelemetry.io`) or by centralized log tracing. Traces are produced through logging augmented with correlation identifiers, and log statements can represent what developers added to the system or instrumented to important system components (i.e., endpoints). The great advantage of this approach is its platform agnosticism.

However, one must recognize the necessity of additional extensions to microservices, and their infrastructure to integrate centralized logging and tracing [8]. For instance, correlation identifiers must be introduced, log centralization must be in place, and health checks must be provided for the most advanced reporting. The dynamic analysis led by telemetry can determine microservice dependencies from call-graphs [19, 43, 31], a heat map of how often are certain endpoints reached.

Nevertheless, the dynamic analysis has limitations. It cannot access details exclusive to codebases (such as which component is responsible for a given endpoint business logic, etc.) [18]. We must also consider the separation of duty relevant to telemetry. Different roles might have different needs. Developers might want to know if their change did not break microservice neighbors. However, DevOps manages telemetry or centralized logs with tracing, not Developers [4]. Such role division introduces indirection in reporting, multi-step interpretation, and latency between what has been developed and what has been identified.

The metaphor for dynamic and static analysis could be whether to use typed-safe or interpreted languages with no type-safety. Developers who manage an individual microservice codebase likely take advantage of quick code change checks. These are based on static analysis and are often part of integrated development environments, build files, or added to the CD/CI pipelines. However, the limit of these tools today is that they only relate to a single codebase. The emerging challenge is that successful new tools will need to operate across codebases and combine results with seeing the system as a whole rather than as separate pieces of the greater puzzle [21, 34].

When comparing static and dynamic analysis, we must understand that these instruments have two different targets. One can inform about the underlying structures and the white-box view; the other details how the system operates within a black-box view.

Naturally, there are overlaps. Both approaches can identify the system's endpoints or its microservices [18]. However, it is also the boundary of where the approach limits stand. Anything below endpoints is the goodwill of tracing instrumentation to access it in dynamic analysis [7]. However, there is a toll since code instrumentation to add additional logging has a performance impact. On the other hand, anything below endpoints is a native perspective for static analysis. On the contrary, dynamic analysis will reveal how users use the system and endpoints, which are more popular than others, etc.

We can observe that static analysis is rather in the control of developers, and dynamic analysis is more relevant to operations (i.e., DevOps engineers). Still, for other stakeholders, i.e., to perform a security assessment, we might need a combination of both. Ideally, both perspectives combine symbiotically, giving comprehensive system insights into its dynamics.

Still, none of the static or dynamic analyses could explain how developers organize or how the system changes over time. To answer such a perspective, Mining Software Repositories (MSR) can indicate how the system structure changes over time and does the organization around particular microservices changes. We can collect additional information related to version control messages, possibly linked to issues in ticketing systems. MSR often time connects with static analysis.

The primary input for static analysis is the system code (source, bytecode, or binary) [2]. In the most basic way, source code is parsed into an Abstract-Syntax Tree (AST) and then converted to other forms of graphs. These phrases are then traversed to perform defined verification or match various anti-patterns [44, 13]. The result of such parsing typically generates an intermediate representation (IR) or a model of the system in which the extracted information and structures are reasoned about.

Static analysis does not only consume code or code changes pushed by MSR. The cloud-native design typically involves build files and container configuration files in the repository, and these files can be easily analyzed to help determine topology [22, 39, 27] and involved technology and future static analysis approaches cannot omit these aspects.

3 Conventional Static Analysis versus Microservice Systems

As introduced previously, conventional static analysis performs on a single codebase. It determines dependencies across various internal structures using an abstraction that makes reasoning about a given system easier [2]. However, cloud-native systems are decentralized, possibly with a self-contained codebase per microservice. This difference makes it more challenging to deliver anticipated results to understand the system's dependencies or reason holistically since each codebase could employ a different framework, platform, or library version. As a result, it is necessary to consider static analysis per each codebase.

Multi-codebase is not the only challenge; individual analysis results do not combine linearly next to each other. Instead, they need careful interweaving in the scope where they overlap - across bounded contexts and interaction. By recognizing these connections, new tools could derive a virtual holistic perspective of the overall system with fine granularity of inter-microservice dependencies.

A good tactic is necessary to overcome the above challenges in the context of polyglot systems. Since many platforms can be used, it is unavoidable to employ multiple platform parsers. The result of all such efforts should be in the form of a unified intermediate representation. This will also enable intermediate representation interweaving that does not need to deal with microservice platform heterogeneity.

In our research and prototyping[45, 6][1] and [28][2], we focused on microservice middleware, and the detection communication patterns between services [35, 42, 41] and on metrics to detect coupling based on the interaction between microservices [1] detected with static analysis [33].

Furthermore, we observed that most microservices would be developed using particular platform frameworks that introduce components [14, 37]. For example, consider Spring, Java Enterprise, C#, and Django. Even if components would not be employed, a good programming convention would be established following separating concerns on the codebase level. With a focus on such practice, we determined that low-level code analysis might

[1] `https://cloudhubs.ecs.baylor.edu/prophet/`
[2] MicroDepGraph `https://github.com/clowee/MicroDepGraph`

Table 1 Static analysis vs. Microservices: Cultural clash.

	Conventional static analysis	Microservice traits
Code perspective	Plain 'low-level' code expected with low-level intermediate representation	Enterprise standards and components are the main constructs
Codebase	Limited to a single codebase; linear result combination does not work for microservices given their dependencies	Decentralized with decentralized codebases per microservice
Heterogeneity	Language-specific (monoglot)	Heterogeneous system parts (polyglots)

be unnecessary, but still, this is the conventional approach. Instead, one should focus on components like data entities, repositories, services, and controllers. In addition, the internal call-graphs across components and involved high-level structures should be detected (i.e., remote-procedure calls, REST-call, event registration, etc.). The result would be in the form of an intermediate representation that forms a component call-graph.

We summarize the gaps for conventional static analysis when placed into a contract with microservices in Table 1. We propose that microservice-aware static analysis may operate with polyglot systems built with heterogeneous platforms; it must recognize high-level structures and components and properly combine results across analyzed codebases.

4 Proposed Methodology for Microservice-aware Static Analysis

The key decision for static analysis is to choose the proper system intermediate representation. We chose a component call-graph since many platforms use components.

With an emphasis on operating with an intermediate representation that forms a component call-graph, we consider the utility of conventional code parsers to determine AST to detect the system structure. Based on common component types across different frameworks, it is possible to detect components, their properties, specifics, and connections. However, this often drags the approach to become platform-specific in its nature.

Nevertheless, working with AST or its converted graphs like control-flow graphs enables us to determine the anticipated intermediate representation of the component call-graph per each microservice.

We have initially assessed this approach on Spring and Java Enterprise platforms on two system benchmarks [47, 10] with success.

In our follow-up work [37][3], we intended to generalize the process across platforms. As a result, we proposed that the AST be extended to be a superset across multiple languages, which leads to a Language-Agnostic AST (LAAST). Some rules can be added to convert constructs across platforms (i.e., defer operator or switch into if/else).

Using LAAST, it is fairly simple to build or customize pattern-matching agents to detect components or higher-level structures. Thus, a common set can be established for conventional framework components. Still, the developer can customize these matches for naming conventions and apply custom callback to populate the component call-graph intermediate representation with a given component properties.

[3] `https://github.com/cloudhubs/source-code-parser`

We have tested this follow-up prototype with success on the previous testbeds [47, 10] and on C++ [20], manually validating precision and recall for component detection above 95% [37].

With the component call-graph intermediate representation of each microservice, we can consider interweaving them. The bottom-up approach is to join involved data models in given bounded contexts. The horizontal approach is to predict possible inter-service communication. This can be accomplished by detecting remote calls to certain endpoints, identifying relative paths, HTTP types, and parameters, and matching them to endpoint signatures. We recognize that only dynamic analysis can recognize inter-service communication with perfect precision. Still, if we solely consider static analysis, this results in the best approximation, and static analysis is about approximation. However, other interactions based on events and brokers can also be considered.

To interweave microservices, first, overlaps with data entities should be identified. Using LAAST matching agents, we can identify entities and reason about their interconnections to derive a data model per each bounded context. For instance, we can use similarity from natural-language processing, Wu-Palmer algorithm [23] to determine potential matches in entities across microservice intermediate representations. Placing identified entities in an overlay helps us connect intermediate representations together and build a context map.

However, other techniques can be used. The next strategy targets cross-service interaction. We consider possible remote calls between microservices. Similarly, we operate on LAAST to identify endpoints, relative paths, HTTP types and parameters, and similarly remote calls within services, which we match based on HTTP types, relative paths, and parameters. We can determine additional dependencies across microservices that strengthen the previously assembled overlay with this route. Performing this across all microservices, we determine the holistic system component call-graph intermediate representation, which corresponds to the latest state of the system.

In addition to the above, we can also account for configuration files in the codebase. This is especially relevant to container descriptors that are part of cloud-native codebases.

We have tested the proposed interweaving on the previously mentioned testbeds [47, 10, 20] and assessed the results manually, with few associations missing in the resulting context map and few unidentified connections in the inter-service interaction [6]. The missing connections were all due to ambiguity caused by choosing from multiple potential URLs at endpoints, which we did not optimize our prototype for, expecting each endpoint to match a single URL.

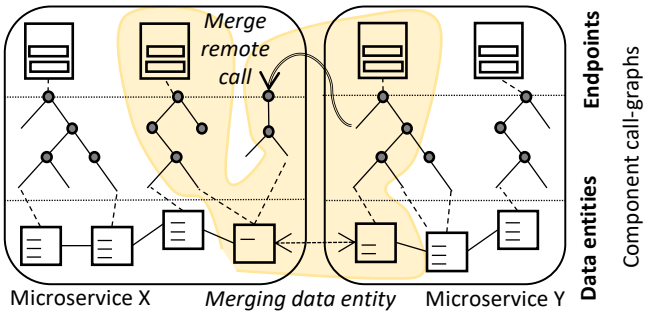

Figure 1 Example interweaving of two microservices X and Y based on remote calls and similar data entities.

Table 2 Microservices-aware static analysis: procedural steps.

Stage	Microservice-aware static analysis	Realization
Step 1	Recognize components in code along with high-level constructs (i.e., remote calls, component interaction)	Use AST or other graphs
Step 2	Establish microservice intermediate representations	Consider component call-graph
Step 3 (optional)	Unification across platforms	Consider language agnostic AST
Step 4	Connect individual microservice intermediate representations (see Figure 1)	• Data entity overlaps • Inter-service calls (remote calls) • Use container descriptors

To summarize the methodology process, we highlight important steps in Table 2. We also illustrate two mentioned interweaving techniques in the context of component call-graphs from two different microservices. This is captured in Figure 1.

5 Discussion on Implications and Challenges

The primary motivation behind the static analysis is automated reasoning and reports [2], as well as system architecture reconstruction [45]. With the ability to operate across the holistic system or multiple microservices, developers (as opposed to DevOps) could gain new aid to quickly understand the impact of their changes [3]. Or get quick feedback on newly introduced anti-patterns [13, 44] and lowered quality metrics. Table 3 lists selected challenge areas.

Considering different concerns, one could be assessing whether the system complies with various organizational policies. Currently, analysts need to review the codebase to determine compliance. Having a holistic system intermediate representation might become easier to assess. Similarly to consistency checking, certain policies could be evaluated.

One related venue for discussion and research is the consistency of business logic. Analyzing business logic is difficult from code, even though we know that service components are to be encapsulated and we can track control and data flows. This opens the question of consistency checking across microservices, which is certainly attractive. However, this also has to consider that modern frameworks add rules via method interception. For instance, frameworks like Drools can greatly reduce management efforts for business rules; however, these again apply to a single codebase. Integrating a configuration server in cloud-native methodology could open a non-intrusive path for centralizing such rules that are now scattered across the system. It would simply utilize principles of generative programming to accomplish this.

Considering the above problem areas (business logic consistency, policies) or other examples like security and privacy, the root cause of problems and dependencies is the manifestation of scattered concerns. Static analysis can extract information about selected concerns from each microservice and put them next to each other to centralize the perspective. This can be accomplished, for instance, by using the component call-graph as the system representation and augmenting its perspectives related to given concerns by targeted code analysis.

Table 3 Selected of challenge areas for static analysis in microservices.

Automated vs. human expert reasoning – anti-patterns, metrics, reports, etc.
Scattered concerns – compliance and consistency in policies, business logic, privacy, security, etc.
Formal methods/verification – using information reconstructed from the code.
Software Architecture Reconstruction – how to perform, which sources to include.
Holistic perspective – how to derive it, which perspectives to consider.
Human-centered perspective – what system information to display in given perspectives to support expert reasoning about the system.
System aspect visualization – augmented/virtual reality, 3D models, etc.
System evolution assessment – many self-contained moving parts must be considered.
Evolution modeling – conversion of intermediate representations system models with instruments designed to speculate on evolution.

Another important avenue for research is the derivation of the system's architectural perspectives that would centralize concerns that are now scattered in decentralized codebases of microservices. For instance, architects might have the motivation to analyze the system's context map or canonical data model, or analysts need to have a single focal point for security assessment and understand all business constraints applied in the system. All these perspectives are necessary to make informed decisions reflecting the current system state, which is currently very difficult to obtain from cloud-native systems.

Results from the static analysis can be accompanied by models for formal verification to aid with trade-off analysis, system evolution planning, or behavior prediction to guarantee correctness. For instance, static analysis can extract a system's intermediate representation and convert it to an architectural description language [30] to help human experts speculatively extend the system via abstract models. Abstractions could also involve choreographic programming to ensure correct concurrency [15, 16]

Despite the above details related to reasoning, the human-centered view should exist and consider visual representation to properly articulate and interpret various concerns, but human experts [3]. In such views, experts could determine anticipated consistency and sketch the system's architectural perspective. Much work has been done in this context, recognizing that architecture can be described through various views [32]. The process is known as Software Architecture Reconstruction (SAR) [36]. As demonstrated in previous works, it can be accomplished through static analysis [45]. Still, researchers address this for microservices through tedious manual code review, possibly outdated documentation assessment [36], or dynamic analysis, which provides only a subset of information, structure, and detail to what is necessary. Software architecture reconstruction fits well with the static analysis process we experimented with.

The typical output of software architecture reconstruction is a system model for answering questions or just reasoning. Reasoning can be automated, such as consistency violation detection or anti-pattern detection [13, 44]. Alternatively, we can also combine these to visually represent certain systems' architectural viewpoints [26].

With regard to system evolution, if we track service interconnection that disappears with an update, something might be wrong with the update, and the developer might be notified about such change impact. This could greatly improve conformance/consistency checking across microservice evolution, which is currently very fragile due to horizontal separation of duty where distinct development teams manage different microservice codebases. However, specific strategies to do so remain to be addressed.

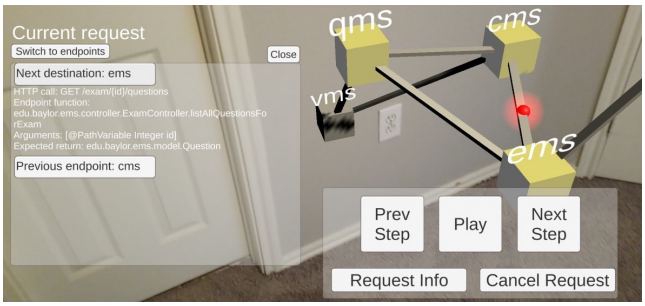

Figure 2 Our AR prototype for SAR and communication simulation.

In the context of challenges for microservices, a broad study by Borgner et al. [5] identified a large set of additional problems that practitioners face with microservices. There is a promising potential for microservice-aware static analysis to address these problems. In the study, the most frequently mentioned issues were missing system-centric perspectives, inter-service dependencies, coordination between decentralized teams, and challenges with outdated documentation. They also mention challenges related to microservice integration, API-breaking changes, etc. We strongly believe all these challenges could be leveraged by introducing robust static analysis tools for cloud-native systems.

6 Experimental Evaluation

We have implemented our methodology in various prototype tools and assessed benefits, limitations, and implications from the Microservice-aware static analysis in cloud-native systems with broader detail.

For instance, we have approached software architecture reconstruction in microservice systems [6, 45] and managed to derive four architectural viewpoints that present the decentralized system as if it was a virtual monolith. This has the potential to realign current static analysis tools to operate on the holistic system.

Figure 3 Our interactive two-dimensional service dependency graph visualization.

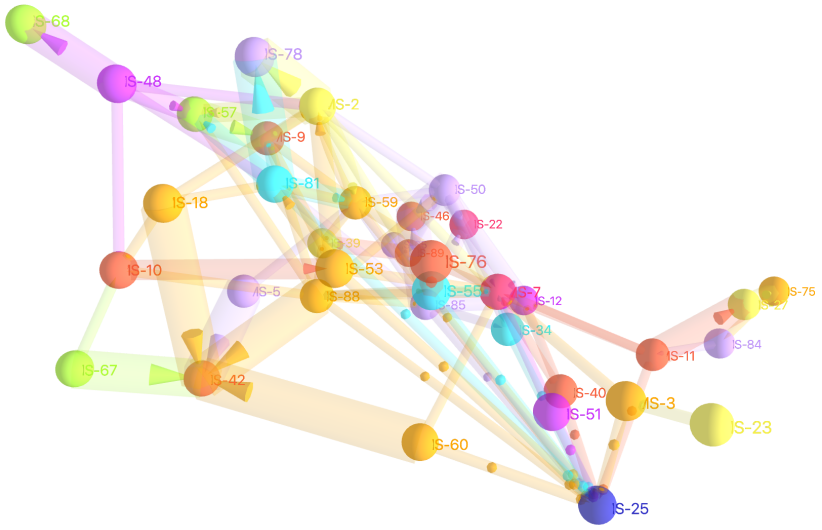

Figure 4 Our interactive three-dimensional service dependency graph visualization.

In addition, we used our model for automated reasoning to detect access policy consistency errors [17] across endpoints. For instance, we might consider access rights from a single microservice context, but when they interplay, they might consider different access rights leading to inconsistencies and possibly vulnerability.

Furthermore, we used our component call-graph intermediate representation along with the architectural viewpoints to detect eleven microservice-specific bad smells [44].

Moreover, the component call-graph intermediate representation has proven well-suited as a model for semantic clone detection across system endpoints [40], which is important when different teams reinvent the same functionality, not knowing about co-existing endpoints.

The granularity of components is suitable for developers in development frameworks. It fits well with the granularity of graph nodes, which makes the visual connection between models and code easy to comprehend.

In addition, we have also researched visualization of systems architecture based on microservices [6]. We have considered that established visualization techniques are too "static" when it comes to developer needs. Thus, we focused on interactivity. Furthermore, we proposed that conventional techniques visualize and model system architecture needs to be reconsidered given the space constraints needed by microservices [11]. Large or even medium-size microservices systems rendered in conventional visualizations require too much space to display relevant information. Thus, in addition to interactivity, we considered three-dimensional spatial visualization.

We share our proof of concepts called Microvision [12, 11] in Figure 2 highlighting one of the systems testbeds. In a large user study involving experts and novices [3], we identified that the benefits of three-dimensional spatial visualization come from mid-size systems and enable novices to identify architectural properties and anomalies as quickly as experts.

However, we also started investigating more web-friendly models that can render in two and three dimensions [24]. In the context of anti-pattern or smell detection, we also considered their visualization in these models [25]. Figures 3 and 4 illustrate our new visual models for two and three-dimensional service dependency graphs, and Figure 5 previews a cyclic dependency anti-pattern highlight in the service dependency graph [26].

Figure 5 Highlighting detected anti-patterns in the visualization.

In summary, microservice-aware static analysis has great potential to address current gaps and challenges with microservices. This article does not anticipate that static analysis is superior to dynamic analysis. It is meant for different goals and challenges; clearly, a broad research opportunity exists for combined analysis.

7 Conclusion

This article discusses static analysis in the context of microservices and cloud-native design. While static analysis is recognized for many benefits, it has not been widely adopted and used for challenges faced in cloud-native system development. We have listed major obstacles preventing static analysis from operating on the holistic system. We point to our experiments that attempted to interweave intermediate representations of microservices to enable such operation. We believe the scientific and industrial community should put more effort into developing robust tools to help developers better face system evolution and maintenance tasks. In future work, we plan to continue our research in this direction of combining decentralized systems. We will also continue to develop prototypes across languages, demonstrating the ability to assess heterogeneous systems to provide a single focal point when assessing certain information and concerns.

References

1 Amr S Abdelfattah and Tomas Cerny. The microservice dependency matrix. *CoRR*, abs/2309.02804, 2023. doi:10.48550/ARXIV.2309.02804.

2 Amr S. Abdelfattah and Tomas Cerny. Roadmap to reasoning in microservice systems: A rapid review. *Applied Sciences*, 13(3), 2023.

3 Amr S Abdelfattah, Tomas Cerny, Davide Taibi, and Sira Vegas. Comparing 2d and augmented reality visualizations for microservice system understandability: A controlled experiment. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*, pages 135–145, 2023. doi:10.1109/ICPC58990.2023.00028.

4 Abdullah Al Maruf, Alexander Bakhtin, Tomas Cerny, and Davide Taibi. Using microservice telemetry data for system dynamic analysis. In *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 29–38. IEEE, 2022. doi:10.1109/SOSE55356.2022.00010.

5 J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann. Industry practices and challenges for the evolvability assurance of microservices. *Empirical Software Engineering*, 26(5):104, 2021. doi:10.1007/S10664-021-09999-9.

6 V. Bushong, D. Das, and T. Cerny. Reconstructing the holistic architecture of microservice systems using static analysis. In *Int. Conf. on Cloud Computing and Services Science (CLOSER)*, 2022. doi:10.5220/0011032100003200.

7 Vincent Bushong, Diptal Das, and Tomas Cerny. Reconstructing the holistic architecture of microservice systems using static analysis. In *Proceedings of the 12th International Conference on Cloud Computing and Services Science - CLOSER*, pages 149–157, 2022. doi:10.5220/0011032100003200.

8 John Carnell and Illary Huaylupo Sánchez. *Spring microservices in action*. Manning Publications Co., 2nd ed. Shelter Island, NY, USA, 2021. URL: https://www.manning.com/books/spring-microservices-in-action-second-edition.

9 Tomas Cerny. Aspect-oriented challenges in system integration with microservices, soa and iot. *Enterprise Information Systems*, 13(4):467–489, 2019. doi:10.1080/17517575.2018.1462406.

10 Tomas Cerny. Microservice Testbed for Texas Teacher Examination, 2020. https://github.com/cloudhubs/tms2020, last accessed 1/2/2022. URL: https://github.com/cloudhubs/tms2020.

11 Tomas Cerny, Amr S. Abdelfattah, Vincent Bushong, Abdullah Al Maruf, and Davide Taibi. Microservice architecture reconstruction and visualization techniques: A review. In *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 39–48, 2022. doi:10.1109/SOSE55356.2022.00011.

12 Tomas Cerny, Amr S. Abdelfattah, Vincent Bushong, Abdullah Al Maruf, and Davide Taibi. Microvision: Static analysis-based approach to visualizing microservices in augmented reality. In *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 49–58, 2022. doi:10.1109/SOSE55356.2022.00012.

13 Tomas Cerny, Amr S. Abdelfattah, Abdullah Al Maruf, Andrea Janes, and Davide Taibi. Catalog and detection techniques of microservice anti-patterns and bad smells: A tertiary study. *Journal of Systems and Software*, page 111829, 2023. doi:10.1016/J.JSS.2023.111829.

14 Tomas Cerny, Jan Svacina, Dipta Das, Vincent Bushong, Miroslav Bures, Pavel Tisnovsky, Karel Frajtak, Dongwan Shin, and Jun Huang. On code analysis opportunities and challenges for enterprise systems and microservices. *IEEE Access*, 8:159449–159470, 2020. doi:10.1109/ACCESS.2020.3019985.

15 Luís Cruz-Filipe and Fabrizio Montesi. Procedural choreographic programming. In *Formal Techniques for Distributed Objects, Components, and Systems: 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings 37*, pages 92–107. Springer, 2017. doi:10.1007/978-3-319-60225-7_7.

16 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. A formal theory of choreographic programming. *Journal of Automated Reasoning*, 67(2):21, 2023. doi:10.1007/S10817-023-09665-3.

17 D. Das, A. Walker, V. Bushong, J. Svacina, T. Cerny, and V. Matyas. On automated rbac assessment by constructing a centralized perspective for microservice mesh. *PeerJ Computer Science*, 7:e376, 2021. doi:10.7717/PEERJ-CS.376.

18 Amr Elsayed, Tomas Cerny, Jorge Yero Salazar, Austin Lehman, Joshua Hunter, Ashley Bickham, and Davide Taibi. End-to-end test coverage metrics in microservice systems: An automated approach. *CoRR*, abs/2308.09257, 2023. doi:10.48550/ARXIV.2308.09257.

19 Silvia Esparrachiari, Tanya Reilly, and Ashleigh Rentz. Tracking and controlling microservice dependencies. *Queue*, 16(4):10:44–10:65, aug 2018. doi:10.1145/3277539.3277541.

20 Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2019. doi:10.1145/3297858.3304013.

21 Mia E Gortney, Patrick E Harris, Tomas Cerny, Abdullah Al Maruf, Miroslav Bures, Davide Taibi, and Pavel Tisnovsky. Visualizing microservice architecture in the dynamic perspective: A systematic mapping study. *IEEE Access*, 2022. doi:10.1109/ACCESS.2022.3221130.

22 G. Granchelli, M. Cardarelli, P. D. Francesco, I. Malavolta, L. Iovino, and A. D. Salle. Towards recovering the software architecture of microservice-based systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 46–53, 2017. doi:10.1109/ICSAW.2017.48.

23 L. Han, A. L. Kashyap, T. Finin, J. Mayfield, and J. Weese. UMBC_EBIQUITY-CORE: Semantic textual similarity systems. In *Conf. on Lexical and Computational Semantics*, 2013. URL: https://aclanthology.org/S13-1005/.

24 P. Harris, M. Gortney, A.S. Abdelfattah, and Tomas Cerny. Designing a system-centered view to microservices using service dependency graphs: Elaborating on 2d and 3d visualization. In *The Recent Advances in Transdisciplinary Data Science: First Southwest Data Science Conference, SDSC 2022, Waco, TX, USA, March 25–26, 2022, Revised Selected Papers*, 2023.

25 A. Huizinga, G. Parker, A.S. Abdelfattah, X Li, T. Cerny, and D. Taibi. Detecting microservice anti-patterns using interactive service call graphs: Effort assessment. In *The Recent Advances in Transdisciplinary Data Science: First Southwest Data Science Conference, SDSC 2022, Waco, TX, USA, March 25–26, 2022, Revised Selected Papers*, 2023.

26 Austin Huizinga, Hossain Chy, Md Showkat, Harris Patrick, Parker Garrett, Mia Gortney, Cerny Tomas, Dario Amoroso d'Aragona, and Taibi Davide. Microprospect: Visualizing microservice system architecture evolution and anti-patterns. In *Software Architecture. ECSA 2023 Tracks and Workshops*, 2023.

27 A. Ibrahim, S. Bozhinoski, and A. Pretschner. Attack graph generation for microservice architecture. In *Symposium on Applied Computing*, 2019. doi:10.1145/3297280.3297401.

28 Mohammad Rahman Imranur, Sebastiano Panichella, and Davide Taibi. A curated dataset of microservices-based systems. In *Joint Proceedings of the Summer School on Software Maintenance and Evolution*. CEUR-WS, sep 2019. URL: http://arxiv.org/abs/1909.03249.

29 Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97—Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings 11*, pages 220–242. Springer, 1997. doi:10.1007/BFB0053381.

30 Luka Lelovic, Michael Mathews, Amr Abdelfattah, and Tomas Cerny. Microservices architecture language for describing service view. In *13th International Conference on Cloud Computing and Services Science (CLOSER 2023)*, 2023. doi:10.5220/0011850200003488.

31 S. Ma, C. Fan, Y. Chuang, W. Lee, S. Lee, and N. Hsueh. Using service dependency graph to analyze and test microservices. In *42nd Annual Computer Software and Applications Conf.*, 2018. doi:10.1109/COMPSAC.2018.10207.

32 Liam O'Brien, Christoph Stoermer, and Chris Verhoef. Software architecture reconstruction: Practice needs and current approaches. Technical report, Carnegie Mellon University, jan 2002. doi:10.1184/R1/6583982.v1.

33 Sebastiano Panichella, Mohammad Rahman Imranur, and Davide Taibi. Structural coupling for microservices. In *11th International Conference on Cloud Computing and Services Science*, apr 2021. doi:10.5220/0010481902800287.

34 Garrett Parker, Samuel Kim, Abdullah Al Maruf, Tomas Cerny, Karel Frajtak, Pavel Tisnovsky, and Davide Taibi. Visualizing anti-patterns in microservices at runtime: A systematic mapping study. *IEEE Access*, 2023. doi:10.1109/ACCESS.2023.3236165.

35 Ilaria Pigazzini, Francesca Arcelli Fontana, Valentina Lenarduzzi, and Davide Taibi. Towards microservice smells detection. In *Proceedings of the 3rd International Conference on Technical Debt*, TechDebt '20, pages 92–97, New York, NY, USA, 2020. doi:10.1145/3387906.3388625.

36 F. Rademacher, S. Sachweh, and A. Zündorf. A modeling method for systematic architecture reconstruction of microservice-based software systems. In *Enterprise, Business-Process and Information Systems Modeling*. Springer International Publishing, 2020. doi:10.1007/978-3-030-49418-6_21.

37 Micah Schiewe, Jacob Curtis, Vincent Bushong, and Tomas Cerny. Advancing static code analysis with language-agnostic component identification. *IEEE Access*, 10:30743–30761, 2022. doi:10.1109/ACCESS.2022.3160485.

38 Sheldon Smith, Ethan Robinson, Timmy Frederiksen, Trae Stevens, Tomas Cerny, Miroslav Bures, and Davide Taibi. Benchmarks for end-to-end microservices testing. *CoRR*, abs/2306.05895, 2023. doi:10.48550/ARXIV.2306.05895.

39 J. Soldani, G. Muntoni, D. Neri, and A. Brogi. The ntosca toolchain: Mining, analyzing, and refactoring microservice-based architectures. *Sw. Practice and Experience*, 51(7):1591–1621, 2021. doi:10.1002/SPE.2974.

40 Jan Svacina, Vincent Bushong, Dipta Das, and Tomás Cerný. Semantic code clone detection method for distributed enterprise systems. In Maarten van Steen, Donald Ferguson, and Claus Pahl, editors, *Proceedings of the 12th International Conference on Cloud Computing and Services Science, CLOSER 2022„ Online Streaming, April 27-29, 2022*, pages 27–37. SCITEPRESS, 2022. doi:10.5220/0011032200003200.

41 Davide Taibi and Kari Systä. A decomposition and metric-based evaluation framework for microservices. In *Cloud Computing and Services Science*, pages 133–149, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-49432-2_7.

42 Davide Taibi and Kari Systä. From monolithic systems to microservices: A decomposition framework based on process mining. In *Proceedings of the 9th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, pages 153–164. INSTICC, SciTePress, 2019. doi:10.5220/0007755901530164.

43 J Thalheim, A Rodrigues, I.E. Akkus, P Bhatotia, Rm Chen, B. Viswanath, L. Jiao, and C. Fetzer. Sieve: Actionable insights from monitored metrics in distributed systems. In *Middleware*, 2017. doi:10.1145/3135974.3135977.

44 A. Walker, D. Das, and T. Cerny. Automated code-smell detection in microservices through static analysis: A case study. *Applied Sciences*, 10(21), 2020. doi:10.3390/app10217800.

45 A. Walker, I. Laird, and T. Cerny. On automatic software architecture reconstruction of microservice applications. *Information Science and Applications*, 2021.

46 Adam Wiggins. The twelve-factor app, 2017. https://12factor.net/, last accessed 1/2/2022. URL: https://12factor.net/.

47 X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao. Benchmarking microservice systems for software engineering research. In *40th Int. Conf.Software Engineering: Comp.*, 2018. doi:10.1145/3183440.3194991.