

Static analysis tools in the era of cloud-native systems

Tomas Cerny and Davide Taibi

¹ Computer Science, Baylor University, Waco, Texas, USA
`tomas_cerny@baylor.edu`

² Tampere University, Tampere, Finland, EU
`davide.taibi@tuni.fi`

Abstract

Microservices 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. The intended system design might deviate far from what is eventually produced and maintained as the architecture tends to degrade over time. This paper 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 and discusses implications for practitioners once robust static analysis tools become available.

1 Introduction

Microservice architectural style is a paradigm to develop systems as a suite of small, self-contained, and autonomous services, communicating through a lightweight protocol. Each microservice has own codebase with a separated configuration to facilitate evolution. This, as a result, enables the separation of duty for roles like architects, developers, and DevOps.

What is, however, less perfect is the separation of concerns. Such separation is likely to be well managed on a single codebase level, but it might get lost with the decentralization and existence of multiple codebases.

Overlaps across microservices with their bounded contexts are inevitable as microservices interact. Implications from these overlaps are reflected in the microservice codebase. Since microservice codebase remains self-contained, overlaps means partially restated definitions, typically reimplemented 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 changes in the microservice codebase, there is no direct indication of the definition being restated elsewhere in other codebases.

We typically want to separate concerns to provide better readability and maintainability. We can do a micro-management and design solid concern separation per each microservice. However, this would only relate to a single microservice and not the whole system. The question to ask is whether we need to see a certain concern per the entire system perspective. Suppose we are architects, most likely yes. Even for developers, it would be nice to see concerns aligned across interacting microservices. However, 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 for security assessment is that each microservice has to be analyzed individually, and then extracted knowledge must be combined ad-hoc, which is tedious, error-prone, and does not scale with agile development.

In this work, we discuss how static analysis could contribute to solve the shortcomings of microservices-based systems. We emphasize how the future tools should adapt to better fit

these systems’ specifics. We base our discussion on a set of prototype tools that 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 point to dynamic system analysis as one possible direction to address decentralization challenges. Dynamic analysis, typically driven by telemetry (i.e. <https://opentelemetry.io>) is often considered as the answer to the system-centric perspective, derived from traces of user interaction or simulated tests. The great advantage of this approach is its platform agnosticism. Still, many additional steps must be done for microservices to be integrated and support this [3]. For instance, correlation ID must be introduced, log centralization must be in place, and health checks must be provided for most advanced reporting. The dynamic analysis led by telemetry can determine microservice dependencies from call-graphs [7, 17, 12], a heat map of how often are certain endpoints reached. However, dynamic analysis does not have access to details only available in codebases. Besides, we must consider the separation of duty relevant to telemetry. It is not developers who manage telemetry, but DevOps which introduces indirection, multi-step interpretation, and latency between what has been developed and what has been found consequently from the operating system. The argumentation could be similar to whether we should use typed-safe or interpreted languages with no type-safety. Developers typically take advantage of quick code and workspace checks that are based on static analysis. These are often part of their integrated development environments, build files, or added to the CD/CI pipelines. However, these tools only relate to a single codebase. The emerging challenge is that successful new tools have to operate across codebases and combine results with seeing the system as a whole rather than as separated pieces of the puzzle.

When comparing static and dynamic analysis, we must understand that these have two different targets. One can tell us what the underlying structures and the white-box view are; the other gives us detail about how the system is used and performs and provides the black-box system overview. Both approaches can identify entry points to the system or to its microservices, and that is where we can see the overlap. However, it is also the boundary of where the approach limits stand. Anything below entry points can be assessed by static analysis, and whatever happens above is the target for dynamic analysis. However, we see that static analysis is rather in control of developers and the dynamic analysis is more relevant to DevOps. Still, for another stakeholder, i.e., to perform a security assessment, we might need a combination of both. Above this, the system can also utilize a hybrid approach to understand system insights and dynamics. Such an approach involves code instrumentation which adds additional logging or measures into the executed code. However, this approach has a performance impact and is often applied in profiling and performance optimization.

Mining Software Repositories (MSR) can indicate how the system structure changes over time. We can collect additional information related to version control messages, possibly linked to issues in ticketing systems. However, we must also assume one important thing; the static analysis does not only consume code or code changes. 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 [9, 16, 11] and involved technology.

Still, the primary input for static analysis is the system code. It is typically parsed into an

Abstract-Syntax Tree (AST). Various tools then analyze the tree to perform defined verification or match patterns. The AST can also be used to generate an intermediate representation (IR) or a model in which the system information is reasoned.

3 Static Analysis for Microservices-Based systems

Conventional static analysis performs on a single codebase. It determines dependencies across various internal structures with a central focal point. However, cloud-native systems are decentralized with a self-contained codebase per microservice. This difference makes it more challenging to deliver anticipated results since each codebase could employ a different framework, platform, or library version. Thus, we must consider static analysis per each codebase.

Multi-codebase is not the only challenge; we cannot just align the analysis results linearly next to each other. Instead, we need to interweave them in the scope where they overlap - across bounded contexts. If we accomplish this, we can derive a virtual holistic perspective of the system with fine granularity of system internal dependencies.

To overcome the above challenges, we need good tactics. Since many platforms can be used, it is unavoidable to employ multiple platform parsers. The result of all such efforts should be in a unified IR. This will also enable IR interweaving that does not need to deal with heterogeneity.

In our research and prototyping[19, 2]¹, we focused on microservice middleware. Furthermore, we observed that most microservices would be developed using particular platform frameworks that introduce components [5, 15]. Among examples, consider Spring or Java Enterprise. Even if components are not employed, a good programming convention will be established following the separation of concerns on the codebase level. With a focus on such practice, we determined that low-level code analysis can be unnecessary. Instead, we can focus on components like data entities, repositories, services, and controllers. In addition, the internal call-graphs and involved high-level structures should be detected (i.e., remote-procedure calls, REST-call, event registration, etc.). With this priority, we can use AST to detect the structure and determine an intermediate representation from these interconnected components augmented with methods, properties, and additional details. We have as assessed this approach on Spring and Java Enterprise on two system benchmarks [20, 4].

In our follow-up work [15]², we intended for generalization and proposed that the AST is extended to be a superset across languages, which we call Language Agnostic AST (LAAST). 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 matchers for naming conventions and apply custom callback to populate the IR with a given component. We have tested this prototype success on the previous testbeds [20, 4] and added one in C++ [8], manually validating precision and recall for component detection above 95% [15]. Still, more experiments for other platforms remain.

With the IR of each microservice, we need to interweave them. The bottom-up approach is to join involved data models. 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, but if we solely consider static analysis, this results with the best approximation. However, other interactions based on events and brokers can be taken into account as well.

¹<https://cloudhubs.ecs.baylor.edu/prophet/>

²<https://github.com/cloudhubs/source-code-parser>

To interweave microservices, first, we identify overlap with data entities. Using LAAST matching agents we identify entities. We analyze these entities to derive a data model per each bounded context. Next, we use similarity from natural-language processing, Wu-Palmer algorithm [10] to determine potential matches in entities across microservice IRs. Placing identified entities in an overlay helps us connect IRs together and build a context map. However, other techniques can be used. Next, 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 IR, which corresponds to the latest state of the system. We have tested this interweaving on the previously mentioned testbeds [20, 4, 8] and assessed the results manually, with few associations missing in the resulting context map and few unidentified connections in the inter-service interaction [2]. The missing connections were all due to ambiguity caused by choosing from multiple potential URLs at endpoints which we did not design to our prototype, expecting each endpoint to match a single URL. In preliminary work, we also detect event interaction through message brokers, typical for cloud-native systems.

4 Discussion on Implications

Using our prototype tools, we managed to assess benefits, limitations, and implications from static analysis over cloud-native system with broader detail.

The primary motivation behind the static analysis is automated reasoning and reports. With the ability to operate across the holistic system or across multiple microservices, developers (as opposed to DevOps) gain new aid to understanding the impact of their changes. For instance, if we track interconnection that disappears with an update, something might be wrong with the update. This would greatly help conformance/consistency checking, 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.

It becomes easier to assess whether the system complies with various organizational policies with holistic system IR. Analysts might need to assess the codebase to determine compliance. Similarly to consistency checking, certain policies could be assessed. One 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. This, however, has to take into account rules added by method interception. Frameworks like Drools can greatly reduce management efforts for business rules; however, these again apply to a single codebase. An integration with configuration server in cloud-native methodology could open a path for centralizing rules.

Another important avenue for research is the centralization of concerns that are now scattered in the decentralized codebases of microservices. For instance, motivation to access the context map or canonical data model, or to have a single focal point for security assessment, understand all business constraints applied in the system. All these perspectives are needed to make wise design decisions, and all these are very difficult to gain from cloud-native systems.

Despite the above details, we need visual representation to properly articulate the concerns, point out the consistency, and sketch the system architectural perspective. A lot of work has been done in this context, recognizing that architecture can be described through various views [13]. The process is known as Software Architecture Reconstruction (SAR) [14] can indeed



Figure 1: Our AR prototype for SAR and communication simulation.

be accomplished through static analysis [19]. Many researchers address this for microservices either through manual code and documentation assessment [14] or through dynamic analysis, which provides only a subset of what is necessary. SAR fits well to the process we described, with the reasoning being recognized as one outcome and visual representation as another.

We have researched SAR in cloud-native and used reasoning to detect access policy consistency errors [6] or to detect bad smells [18]. We have also researched visualization of microservices [2] with work in progress on three-dimensional rendering using augmented reality as sketches in Figure 1 for one of the systems testbeds.

A large set of challenges was identified by practitioners in a large study on microservice evolvability [1]. Missing system-centric perspectives, inter-service dependencies, coordination between decentralized teams, and challenges with outdated documentation, were the most frequently mentioned issues. They also mention challenges related to microservice integration, API breaking changes, etc. We strongly believe all these could be leveraged by introducing robust static analysis tools for cloud-native systems.

This article does not anticipate that static analysis is superior to dynamic analysis. It is simply meant for different goals and challenges, and clearly, a broad research opportunity exists for combined static and dynamic analysis.

5 Conclusion

In this article, we discuss static analysis in the context of 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 systems. We have listed major obstacles preventing static analysis from operating on the holistic system and 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 that can 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.

6 Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1854049, a grant from Red Hat Research <https://research.redhat.com>, and a Grant from Ulla Tuominen Foundation.

References

- [1] J. Bogner, J. Fritzsche, S. Wagner, and A. Zimmermann. Industry practices and challenges for the evolvability assurance of microservices. *Empirical Software Engineering*, 26(5):104, 2021.
- [2] 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.
- [3] John Carnell and Illary Huaylupo Sánchez. *Spring microservices in action*. Manning Publications Co., 2nd ed. Shelter Island, NY, USA, 2021.
- [4] Tomas Cerny. Microservice Testbed for Texas Teacher Examination, 2020. <https://github.com/cloudhubs/tms2020>, last accessed 1/2/2022.
- [5] Tomas Cerny, Jan Svacina, Dipta Das, Vincent Bushong, Miroslav Bures, Pavel Tisnovsky, Karel Frajtek, Dongwan Shin, and Jun Huang. On code analysis opportunities and challenges for enterprise systems and microservices. *IEEE Access*, 8:159449–159470, 2020.
- [6] 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.
- [7] Silvia Esparrachiari, Tanya Reilly, and Ashleigh Rentz. Tracking and controlling microservice dependencies. *Queue*, 16(4):10:44–10:65, August 2018.
- [8] 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. Zaruvinisky, 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.
- [9] 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.
- [10] 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.
- [11] A. Ibrahim, S. Bozhinoski, and A. Pretschner. Attack graph generation for microservice architecture. In *Symposium on Applied Computing*, 2019.
- [12] 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.
- [13] Liam O’Brien, Christoph Stoermer, and Chris Verhoef. Software architecture reconstruction: Practice needs and current approaches. Technical report, Carnegie Mellon University, 01 2002.
- [14] 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.
- [15] Micah Schiewe, Jacob Curtis, Vincent Bushong, and Tomas Cerny. Advancing static code analysis with language-agnostic component identification. *IEEE Access*, 10:30743–30761, 2022.
- [16] 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.
- [17] 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.
- [18] A. Walker, D. Das, and T. Cerny. Automated code-smell detection in microservices through static analysis: A case study. *Applied Sciences*, 10(21), 2020.
- [19] A. Walker, I. Laird, and T. Cerny. On automatic software architecture reconstruction of microservice applications. *Information Science and Applications*, 2021.
- [20] 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.