On Automatic Software Architecture Reconstruction of Microservice Applications

Andrew Walker, Ian Laird, and Tomas Cerny

Baylor University, One Bear Place, Waco, Texas, 76706
tomas_cerny@baylor.edu

Abstract. The adoption of Microservice Architecture (MSA) is rapidly becoming standard for modern software development. However, the added benefits of using a distributed architecture, including reliability and scalability, come with a cost in increasing the system's complexity. One way developers attempt to mitigate the effects of an overly complicated system is through Systematic Architecture Reconstruction (SAR), which creates a high-level overview of the system concerns. This is typically done manually, which takes a great amount of effort from the developers. This paper proposes a method for automatically completing SAR of an MSA application through code analysis and demonstrating it on a case study on an existing microservice benchmark application.

Keywords: Microservice · Software Architecture Reconstruction · Code Analysis · Static Analysis · Information Extraction · Compliance

1 Introduction

Microservice Architecture (MSA) offers many benefits for modern software development. Primarily these include benefits in scalability and reliability. With a distributed development model, different developers' teams can work separately on different modules of the application. While this offers rapid development benefits, it also allows errors to slip into the application through discrepancies between modules. Another side effect of this distributed development is an increase in the complexity of the application.

One way to mitigate these unintended errors is through Software Architecture Reconstruction (SAR). The construction of a simplified overview of the application can help developers understand the application's full scope, even beyond their modules. One study [18] underlines that SAR is key to architecture verification, conformance checking, and trade-off analysis.

Many methods have been proposed for SAR, even for distributed systems, but none can fully automate the domain, technology, service, and operation views. In this paper, we propose a novel automation method applied to an existing distributed system SAR methodology. We then verify our automation against an existing microservice benchmark system.

The paper is organized as follows. The SAR and static-code analysis background is given in Section 2 and SAR state of the art in Section 3. Our automatic SAR method for microservice apps is described in Section 4. Section 5 tests our method against an existing testbed application. Section 6 concludes the paper.

2 Background

This section presents background on the SAR process as a whole, the methodology that serves as the source of our automation goals, and lastly, static-code analysis, which is the method we use to automate the SAR process.

2.1 Systematic Architecture Reconstruction

SAR has historically been defined with four distinct phases. These are as follows:

1. *Extraction*: This phase collects all of the artifacts needed during the next three phases of SAR. The artifacts collected are relevant to the "views" that are being constructed. A "view" is defined as a set of related artifacts that cover a concern of the architecture of the system.
2. *Construction*: This phase creates a canonical representation of the views and usually stores them in some form, like a database.
3. *Manipulation*: This phase combines the views to allow for the answering of more complicated questions in the next phase. How the views are combined is relevant to the specific application being reconstructed.
4. *Analysis*: This phase answers questions about a system given the overall views of the architecture constructed in the previous phases. There are almost infinite questions that can be asked, covering multitudes of domains, including networking, security, and code quality.

For this paper's purposes, we aim to demonstrate a framework for SAR analysis, so we do not focus on a specific question and thus exclude phase 4.

2.2 Views

The core of successful SAR is the construction of effective views of the architecture of a system. Care must be taken to choose views that are relevant to the questions being asked about a system. Since we are focusing on creating a general framework for SAR, instead of asking a particular question, we chose a set of views that provide good coverage of the system's concerns. We took our set of views from a previous study on SAR methodology [18]. We will briefly outline the four of them below.

- *Domain View*: This view covers the domain concerns of an MSA application. It describes the entity objects of the system as well as the datasource connections of those objects.
- *Technology View*: This view focuses on the technology aspect of an application. It describes the technologies used for microservice implementation and operation.
- *Service View*: The view focuses on service operators. It describes the service models that specify microservices, interfaces, and endpoints.
- *Operation View*: This view focuses on the ops concern of a system. It describes service deployment and infrastructure, such as containerization, service discovery, and monitoring.

Each of these views can be understood as distinct concerns within the system but also as related to the others. For example, the domain view intersects with the service view since the service view can be thought of as the intersections of data between microservices and the technology view since the technology view describes how data is stored in the application.

Another key point about the construction of these views is that ultimately each view is an aggregation of a smaller view encompassing a disparate microservice. Each microservice has a bounded-context of its concerns, but these can be aggregated into a fully centralized perspective of the system's architecture.

2.3 Static-Code Analysis

Static-code analysis is what makes an internal inspection of an application possible. It's used throughout software development but is primarily used to detect bugs in a piece of software. There are two main processes for static code analysis - source code analysis and bytecode analysis. These are used for several processes, including *recognizing* components, classes, methods, fields or annotations, *tokenization*, and *parsing*, which produce graph representations of the code. These include Abstract Syntax Trees (AST), Control-Flow Graphs (CFG) [12, 20, 25], or Program Dependency Graphs (PDG) [21, 22].

Bytecode analysis [1] uses the compiled code of an application. This is useful in uncovering endpoints, components, authorization policy enforcements, classes, and methods. It can also be used to augment or build CFG, or AST [11, 10, 13]. However, the disadvantage is that not all languages have a bytecode.

To fill this gap, we can also turn to source code analysis [3]. It parses through the source code of the application, without having to compile it into an immediate representation. Many approaches exist to do this, however, most tools tokenize the code and construct trees, i.e., AST [21, 22], CFG [12, 20, 25], or PDG [6, 24].

Static-code analysis can also be extended beyond just the source-code and compiled artifacts to include other application artifacts. Static analysis gives the ability to access Docker images [16, 23] through the Kubernetes[1] platform. Since Kubernetes is typically the enterprise standard for service-meshes, it provides thorough coverage of the operation view for many enterprise applications.

Despite all of the usefulness of static-code analysis, limits exist with these representations in encapsulating enterprise systems' complexity. To mitigate the shortcomings of existing static code analysis techniques on enterprise systems, we must augment existing techniques with an understanding of enterprise standards [4, 14]. A more realistic representation of the enterprise application can be constructed with aid from either source code analysis or bytecode analysis. This primarily includes a tree representation and the detection of the system's endpoints, and the construction of a communication map. These augmented representations and metadata have been successful in other problem domains, including security, networking, and semantic clone detection.

[1] https://kubernetes.io/

3 Related Work

Most of the work done in SAR on microservices has focused on the methodology instead of the automation of the methodology. However, some works have partially automated a full SAR representation of a microservice-based system.

Ratemacher et al. [18], defined a methodology for the construction of all four of our targeted views. Their process involved the construction of a canonical representation of the data model. From there, they employed methods to fuse module views. Finally, it performed architecture analysis to answer questions about architecture implementations from the reconstructed architecture information. Unfortunately, they only provided a manual assessment of their benchmark application and included no attempt at automation. However, they highlighted the need for furthering the work in SAR to automate the process.

Alshuqayran et al. [2] conducted a manual reconstruction process that included modeling the application to derive the overall architecture while utilizing multiple module merge strategies such as data model integration and meta-model mapping rules. They do not, however, apply their strategies towards extraction or merging of domain concepts.

Ibrahim et al. [9] considered container-based deployment configuration files to derive MSA module topology. In particular, they used Docker Compose to extract the topology of the module orchestration. In addition to topology, they generated "attack graphs", that depict actions which attackers may use to reach their malicious goal. Attack graphs help developers identify attack paths that comprise exploitable vulnerabilities in deployed services. They underlined that testers commonly construct such graphs manually, but container configuration files provide a well-structured input for automation. Their open-source tool is based on Clair [19], a vulnerability scanner for Docker containers and images. It generates image vulnerabilities linked to CVE [17] as well as connections to an attack vector for each vulnerability. An attack vector describes the conditions and effects that are connected to vulnerability.

Mayer et al. [15] propose an automatic method for extracting the domain, service, and operation information of an application. It utilizes a combination of static analysis and runtime analysis to construct a language-agnostic representation of each service and its interaction with other services. The downside to this approach is that to fully construct the representations, and not just a domain model, the application needs to be deployed. The proposed upside to this is that we no longer need a litany of parsers for the different language implementations of a heterogeneous MSA application; however since it still relies on parsing for the domain view, the parsers are needed regardless, and the dynamic approach provides extra overhead. It also creates the possibility of an incomplete view since communication paths that are not traversed during the extraction phase are absent from the final view.

MicroART [7] tool automates SAR. It extracts information about the service concern of a module (service names, ports, etc.) from source code repositories. It also performs log analysis during runtime to discover containers, network interfaces, and service interaction. The user must provide the running container's

location since MicroART does not extract that information automatically. It then uses that information to generate views for the service and operations concerns. It does not consider the domain or technology views.

Zdun et al. [26] propose a methodology for the service view; however, it approaches the problem differently from the previously mentioned tools. It proposes a method for measuring MSA conformance. Because of this approach, it uses a formal model when conducting SAR. Conformance is then assessed via metrics and constraints defined by the relationships between these types. Though it considers the service view of an application, domain, technology, and operation views are not considered for its conformance measure.

We are aware of no tool at present that is capable of fully automating all four of our chosen architecture views on a microservice-based system. A breakdown of the capabilities of the tools mentioned previously is available in Tab.1.

Table 1. Comparison of Modern SAR Tools

	Domain	Technology	Service	Operation	*Automated*
Alshuqayran et al		X	X	X	No
Ibrahim et al				X	Yes
Mayer et al	X		X	X	Yes
MicroART			X	X	Yes
Rademacher et al	X	X	X	X	No
Zdun et al			X		No
Our approach	X	X	X	X	Yes

4 Proposed Method

In this section, we introduce our proposed method for automatically conducting SAR. In particular, we use static-code analysis, both source-code and bytecode, as well as other dependency and application analysis to construct bounded-contexts of the different views for each microservice. Then we aggregate them into a full-scope centralized perspective for each view, which consists of all the microservices aggregated into a mesh.

We start with the *extraction* phase of SAR. In particular, we extract Control-Flow Graphs (CFG) and Program Dependency Graphs (PDG). This allows us to construct a representation of the method calls and internal flow of data in a microservice. We also detect endpoints and their metadata, including security constraints and policies along with parameters and internal method branches, conditions, and loops. Lastly, we detect all entities within a specific module. All of this metadata is important to be able to construct a full centralized perspective of a view and offer a complete SAR overview.

The next phase in the SAR process is the *construction* phase. Once we extract the module-specific information, we represent it in a graph format. We further link the additional extracted metadata to its corresponding module. By aggregating all of the individual metadata artifacts in a cohesive and consistent way, we can easily move into the next phase.

To begin the *manipulation* phase, we use existing strategies of module fusion. Based on DDD [18], each module considers a bounded context [5], which includes a limited perspective of the data model, and often partially overlaps through certain data entities with other modules. This overlap is a key strategy in this phase of SAR. We begin with entity matching by looking for entities from distinct modules with a subset match of properties, data types, and possibly names. For this matching, we also considered natural language processing strategies (Wu-Palmer algorithm [8]). We derive the canonical data model and, through the matched entities, promote data and control dependencies.

The second perspective of static analysis we considered to merge modules is interaction. We identify all endpoints, parameter types, and metadata, and then the remote procedure calls within the methods. Next, we aim to match them and generate a complete service view of the entire application. This allows us to augment the result involving the canonical model.

We do not aggregate the technology view as each microservice is distinct in its choice of implementation; this is a hallmark of the microservice architecture.

The output of our application is a set of module metadata and the connections between them. Each individual microservice contains its bounded context domain model, while the centralized perspective contains the fused domain model. Similarly, each microservice contains its own service registry information, and the centralized perspective contains a graph that shows the connection edges between the disparate services. Each microservice contains its technology information, and an aggregate list of the technologies, broken up by layer, is in the centralized perspective. Lastly, each microservice owns its own deployment and operation information, and the centralized perspective contains a graph of connected deployments.

5 Case Study: Train Ticket

To demonstrate our framework's effectiveness, we tested it on an existing microservice benchmark, TrainTicket[2]. We chose this benchmark since it was specifically designed to emulate a real-world microservice application, consisting of 41 microservices and over 60,000 lines of code. It is written in Spring Boot and uses MongoDB for the datasource. It uses either Docker[3] or Kubernetes for deployment and either NGINX[4] or Ingress[5] for routing.

5.1 Domain View

The domain view is constructed through the analysis of the entities within each microservice. Our testbed broke from the convention with entity objects. Typically, entities can be discovered through the use of Enterprise standard annotation such as @Document for MongoDB objects or @Entity for MySQL objects.

[2] https://github.com/FudanSELab/train-ticket
[3] https://www.docker.com/
[4] https://www.nginx.com/
[5] https://kubernetes.io/docs/concepts/services-networking/ingress/

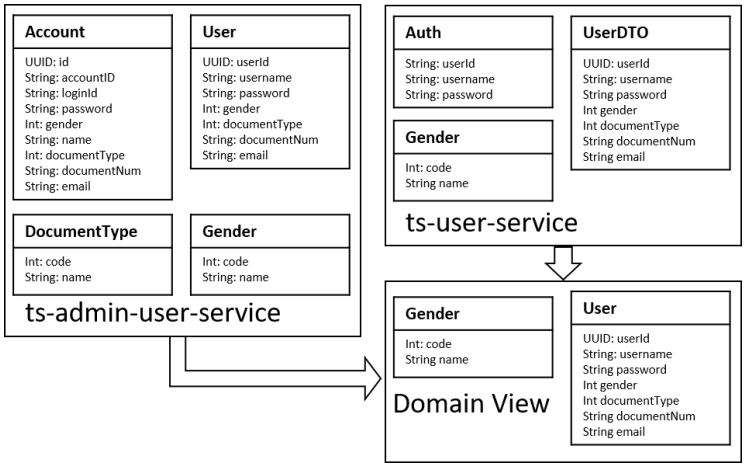

Fig. 1. Merged Domain View from TrainTicket

However, not all entities were annotated within the application. Because of this, we extended our definition of a system entity to include an annotated object, any object that is a field to such an object, or a POJO that matches a known entity in the system. Based on this, we could determine that object was an entity in its microservice, even if it was explicitly marked. An example merging is in Fig. 1.

5.2 Technology View

The technology view was broken into three sections, each representing one application layer (presentation, business, and data). For each layer, the system's underlying technologies were found by analyzing the source code files and the dependency management files (e.g., pom.xml for Maven), and system configuration files. We verified our findings against the existing TrainTicket documentation.

5.3 Service View

The service view is constructed using the generated internal communication diagram and the metadata of each module. In Fig. 2, we show a small selection of the canonical model for the service view of TrainTicket. Even a small selection

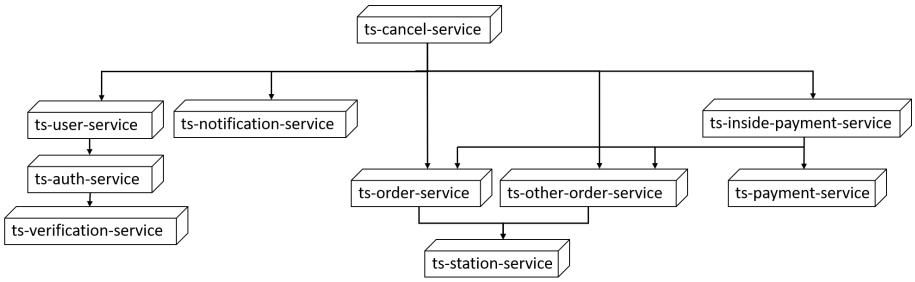

Fig. 2. Service View from TrainTicket

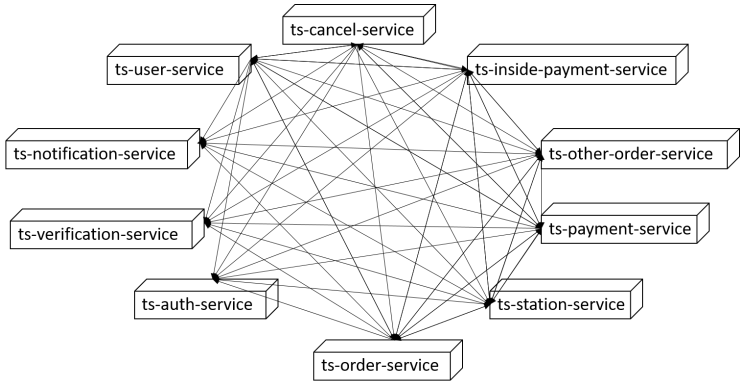

Fig. 3. Operation View from TrainTicket

from the overall service view can show some relevant observations. For example, from the documentation[6] of the TrainTicket system, it appears that the cancel service only has a dependency on the order service however through the lens of the service view, we can see that the cancel service is far more coupled to the other service. It relies on four other services, in addition to the order service.

5.4 Operation View

The operation view of the SAR process defines a topology of the containerization of the application. Fig. 3 shows a small section of the TrainTicket topology. The TrainTicket containerization defines a singular network to connect each of the containers, which creates a graph where each node is connected to every other. This example shows our method benefit for full automation of the four application views. If only the operation view was available, it would create an incorrect/incomplete view of the application. By combining the operation view with the service view, the edges can be mapped to create the topology shown in Fig. 2.

5.5 Threats to Validity

Internal Validity: To verify our approach, we utilized the methodology we automated to extract the SAR views manually for our benchmark system. We had multiple people extract the information and construct their views.
External Validity: The effectiveness of SAR, regardless of if manual or automated, comes down to the availability of artifacts to analyze. For our application, we lack the ability to analyze artifacts such as images, diagrams, or textual descriptions of an application. We do not believe this impacts our application's overall effectiveness; however, since the concerns we automated, domain modeling, services, and containerization can be extracted through enterprise standards that do not vary from application to application. This means that our framework is capable of analyzing any application which uses the enterprise standards.

[6] https://github.com/FudanSELab/train-ticket/blob/master/image/2.png

6 Conclusion

MSA is the mainstream direction for modern software development, and while extensive work has been done to define SAR methodologies on distributed systems, there is a lack of work on automation. Without automation, developers rely on expending a large amount of effort to generate the SAR artifacts manually. This paper demonstrated a framework for automatic SAR across four important views of the application, domain, technology, service, and operations. With our framework, developers can focus on answering questions about their application, phase 4 of SAR, instead of focusing all their time in phases 1-3.

Acknowledgement

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

References

1. 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 Heidelberg, Berlin, Heidelberg (2007)
2. Alshuqayran, N., Ali, N., Evans, R.: Towards micro service architecture recovery: An empirical study. In: 2018 IEEE International Conference on Software Architecture (ICSA). pp. 47–4709 (2018)
3. Chatley, G., Kaur, S., Sohal, B.: Software clone detection: A review. International Journal of Control Theory and Applications **9**, 555–563 (01 2016)
4. DeMichiel, L., Shannon, W.: JSR 366: Java Platform, Enterprise Edition 8 Spec (2016), `https://jcp.org/en/jsr/detail?id=342`
5. Finnigan, K.: Enterprise Java Microservices. Manning Publications (2018), `https://books.google.com/books?id=KaSNswEACAAJ`
6. Gabel, M., Jiang, L., Su, Z.: Scalable detection of semantic clones. In: Proceedings of the 30th International Conference on Software Engineering. pp. 321–330. ICSE '08, ACM, New York, NY, USA (2008). https://doi.org/10.1145/1368088.1368132, `http://doi.acm.org/10.1145/1368088.1368132`
7. Granchelli, G., Cardarelli, M., Di Francesco, P., Malavolta, I., Iovino, L., Di Salle, A.: Towards recovering the software architecture of microservice-based systems. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW). pp. 46–53 (2017)
8. Han, L., L. Kashyap, 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 (Jun 2013)
9. Ibrahim, A., Bozhinoski, S., Pretschner, A.: Attack graph generation for microservice architecture. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. p. 1235–1242. SAC '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3297280.3297401

10. 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. pp. 36–42. IWSC '12, IEEE Press, Piscataway, NJ, USA (2012), http://dl.acm.org/citation.cfm?id=2664398.2664404
11. Keivanloo, I., Roy, C.K., Rilling, J.: Sebyte: Scalable clone and similarity search for bytecode. Science of Computer Programming **95**, 426 – 444 (2014). https://doi.org/https://doi.org/10.1016/j.scico.2013.10.006, http://www.sciencedirect.com/science/article/pii/S0167642313002773
12. Kumar, K.S., Malathi, D.: A novel method to find time complexity of an algorithm by using control flow graph. In: 2017 International Conference on Technical Advancements in Computers and Communications (ICTACC). pp. 66–68 (April 2017). https://doi.org/10.1109/ICTACC.2017.26
13. Lau, D.: An abstract syntax tree generator from java bytecode (2018), https://github.com/davidlau325/BytecodeASTGenerator
14. Makai, M.: Object-relational mappers (orms) (2019), https://www.fullstackpython.com/object-relational-mappers-orms.html
15. 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)
16. Merkel, D.: Docker: Lightweight linux containers for consistent development and deployment. Linux J. **2014**(239) (Mar 2014), http://dl.acm.org/citation.cfm?id=2600239.2600241
17. MITRE: Common Vulnerabilities and Exposure (2020), http://cve.mitre.org
18. Rademacher, F., Sachweh, S., Zündorf, A.: A modeling method for systematic architecture reconstruction of microservice-based software systems. In: Enterprise, Business-Process and Information Systems Modeling. pp. 311–326. Springer International Publishing, Cham (2020)
19. Red Hat, I.: Clair : a vulnerability scanner for Docker containers and images (2020), https://coreos.com/clair/docs/latest/
20. Ribeiro, J.C.B., de Vega, F.F., Zenha-Rela, M.: Using dynamic analysis of java bytecode for evolutionary object-oriented unit testing. In: 25th Brazilian Symposium on Computer Networks and Distributed Systems (SBRC) (2007)
21. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Sci. Comput. Program. **74**(7), 470–495 (May 2009). https://doi.org/10.1016/j.scico.2009.02.007
22. Selim, G.M.K., Foo, K.C., Zou, Y.: Enhancing source-based clone detection using intermediate representation. In: 2010 17th Working Conference on Reverse Engineering. pp. 227–236 (Oct 2010). https://doi.org/10.1109/WCRE.2010.33
23. Soppelsa, F., Kaewkasi, C.: Native Docker Clustering with Swarm. Packt Publishing (2017)
24. Su, F.H., Bell, J., Harvey, K., Sethumadhavan, S., Kaiser, G., Jebara, T.: Code relatives: Detecting similarly behaving software. In: ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 702–714. FSE 2016, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2950290.2950321
25. Syaikhuddin, M.M., Anam, C., Rinaldi, A.R., Conoras, M.E.B.: Conventional software testing using white box method. Kinetik: Game Technology, Information System, Computer Network, Computing, Electronics, and Control **3**(1), 65–72 (2018). https://doi.org/10.22219/kinetik.v3i1.231
26. Zdun, U., Navarro, E., Leymann, F.: Ensuring and assessing architecture conformance to microservice decomposition patterns. In: Service-Oriented Computing. pp. 411–429. Springer International Publishing, Cham (2017)