Reconstructing the Holistic Architecture of Microservice Systems using Static Analysis

Vincent Bushong[a], Dipta Das[b] and Tomas Cerny[c]
Computer Science, Baylor University, One Bear Pl, Waco, TX, U.S.A.

Keywords: Software Architecture Reconstruction, Reverse Engineering, Static Analysis, Microservices, Cloud-native.

Abstract: Cloud-native systems, fueled by microservice architecture, provide several benefits to enterprises. These benefits include scalability, short deployment cycles, and flexibility for evolution. Most benefits come from microservices' independence and decentralization. However, the pay-off comes as a lack of a centralized view of the overall system's architecture. The system's data model is separated among and partially replicated between each microservice, requiring extra effort to create a single view on the context map. Additionally, while a microservice's API and its interaction can be statically documented (i.e., communication diagram), system evolution makes it difficult to maintain. As a result, modifications to the system can decay from the original intended design, and the changes will be obscured by the lack of an up-to-date centralized view. To address this, we propose a method of software architecture reconstruction based on static code analysis of the microservice mesh, generating a communication diagram, context map, and microservice-specific bounded contexts. This gives developers and Development and Operations engineers (DevOps) a centralized view of how the overall program works, useful for furthering system comprehension and observation.

1 INTRODUCTION

Cloud-native systems build on the Microservice Architecture (MSA). The use of this architecture is more or less standard for enterprise software development. Microservices break the application functionality into many small, independently developed, and deployed services that communicate with each other. The integration and business logic are implemented within each microservice, and thus no centralized model is agreed upon. Microservices are aware of the producer interfaces, which introduces loose coupling. As a result, microservices can be developed, deployed, and scaled independently, leading due to more rapid development and update cycles and improved performance of the entire system.

Due to the broad decentralization, any overall system documentation can quickly become out of date. A developer new to the system will likely need to parse through documentation and code from different projects, possibly using different technologies, coding styles, and conventions. Such complex readability leads to increased costs: time that could have been spent on development has to be spent on learning how a system works. It would be difficult to identify specific system concerns on the overall system since each concern is implemented individually among the separate microservices.

One specific feature of the microservice approach is that there is no global data model as that would be a form of coupling, which microservices aim to avoid. This makes individual service development faster and the whole system flexible and loosely coupled, but it can lead to inconsistencies in how each microservice views and uses its data. Entities are used in certain microservices and not others, and even when they share entities, microservices do not always need to use all of their properties. Therefore, the extent of the data stored about any single entity and where and how it is used in the system is hidden among the individual services. Creating and updating domain models and other documentation artifacts help to alleviate this, but in practice, the documentation does not always remain current, especially in the decentralized environment common for microservices.

Another hidden aspect is inter-service communication. Without an integration layer, microservice evolution is not affected by other microservices, but there is no central location for the business logic of the

[a] https://orcid.org/0000-0003-0475-4232
[b] https://orcid.org/0000-0001-8366-2453
[c] https://orcid.org/0000-0002-5882-5502

broader functionality. For example, to know which services call each other to determine services affected by an API update, a developer would have to check each individual service for calls to the others.

Practitioners have well noticed the challenges with microservices. In particular, a large survey on microservice industry practice (Bogner et al., 2021) revealed a missing system-centric view as one of the dominant challenges. This challenge is followed by others, such as outdated or possibly missing documentation and inter-service dependencies, causing a ripple effect. Given that the systems evolve, there is a clear gap in instruments that the industry practitioners need from what they currently have.

To address issues of hidden information and challenges or microservice practitioners, we propose a method of Software Architecture Reconstruction (SAR) for microservices. We have introduced this method in the Late-Breaking Results (Bushong et al., 2021). This paper details our method, which uses static code analysis to automatically generate context maps and inter-service communication diagrams for microservice projects. Using this method, we recover the data model and communication logic of microservice systems. We demonstrate our method's effectiveness with a case study on a benchmark system.

In the remainder of the paper, we discuss background and previous work in sections two. We describe the our method, its rationale and outline our approach and prototype Prophet, in section three. We demonstrate our approach's viability with a case study in section four. Finally, we discuss the impact and conclusions in section five.

2 RELATED WORK

As the mainstream approach to enterprise application development (NGINX, Inc., 2015), MSA offers numerous benefits owing much to the independence of the individual services. Independent service scalability improves performance; independent service deployability reduces the complexity of deployment processes and reduces the time required to push updates. Independent service development allows each piece of functionality to be developed using the most appropriate tools (Cerny et al., 2018).

Containerization is often the underlying deployment mechanism, usually by means of a cloud-based container orchestration framework such as Kubernetes. To more easily handle management and monitoring concerns such as service discovery, scaling, and load balancing, another tool such as OpenShift is used above the cloud-based container framework.

The domain models used in MSA reflect the services' independence. The most common MSA approach is to take a single problem domain and separate it into bounded contexts. The idea of a bounded context originates within Domain-Driven Design (DDD) (Evans, 2014). Instead of creating a single, unifying model for an entire large system, the model is split into multiple bounded contexts. In each bounded context, entities are defined in the way most useful for that context; this way, anyone working with an entity in one context does not need to worry about the details of that entity's usage in other contexts since they can use their own definition (Evans, 2014).

This separation is well-suited to the MSA mindset of independent services. Each microservice represents a different bounded context of the overall system, created and maintained by a separate team for that service's particular needs (Cerny et al., 2018). Entities in a microservice's bounded context can be created with only those properties needed within that particular service, even if it uses other properties in the wider system. This keeps the services decoupled. However, the lack of a centralized model may require restating information across the services. It becomes difficult to maintain a single view of how the entire system represents its data and the business rules and policies applicable to that data (Cerny et al., 2018).

When designing bounded contexts in DDD, it is useful to create a context map, a diagram that shows the bounded contexts and their entities' relationships across the entire system (Evans, 2014). We aim to extract context maps and individual bounded contexts from existing systems, yielding a view of how the system and individual microservices view the entities.

The goal of obtaining a centralized view of how a system works is known as commonly referred to as Software Architecture Reconstruction (SAR) (Rademacher et al., 2020a). SAR derives a representation of software architecture from artifacts like documentation or source code. SAR is a key step in architecture verification, conformance checking, and trade-off analysis (Rademacher et al., 2020a). Besides these, SAR itself is relevant in combating software architecture degradation, the process wherein changes to a software system shift the architecture away from what was originally intended. In the rest of this section, we will examine existing approaches to SAR and its applications to architecture degradation.

There are three broad categorizations for SAR methods: dynamic or runtime analysis, where a tool constructs the view at runtime; static analysis, where the view is constructed from artifacts available before deployment; and manual analysis, where a human examines the system and manually constructs a representation of it. Manual analysis, while clearly not an approach to creating an automated tool, is an important step in proving a proposed method or validating the results; we performed a manual analysis in our case study to compare our results. Ratemacher et al. (Rademacher et al., 2020a) manually collected architecture-related artifacts, constructed a canonical representation of the data model, and based on that fused module views. They then performed architecture analysis on the results to answer hypotheses about architecture implementations from the reconstructed architecture information. We see this as a promising direction to automate the SAR process.

2.1 Dynamic SAR

Dynamic analysis can operate on several different runtime data sources. It has been used for a myriad of end goals, ranging from analyzing runtime traces extracted from logs in order to find timing errors (Cinque et al., 2019) to extracting service dependency graphs by extracting remote procedure calls from network logs in a microservice mesh (Esparrachiari et al., 2018) or by uncovering dependencies between monitored metrics for components of a distributed system (Thalheim et al., 2017).

Runtime analysis-based SAR has taken many forms. One technique is to use a language's instrumentation features to report data based on a framework or custom-made annotations in a program, which can then be analyzed at runtime. This data can be used to model some concerns of a microservice system by detecting where microservices call each other. This approach has been used to model microservice dependencies and find incomplete test coverage of calls across an entire microservice mesh (Ma et al., 2018). This approach can also be used to detect discrepancies between required and provided service versions, as well as generate performance metrics from service error data (Ma et al., 2019).

Interceptors can be used to a similar end; (Mayer and Weinreich, 2018) use the Spring framework's interceptors to monitor runtime calls between services to generate an architectural view of a microservice system. Similarly, calls could be intercepted and rerouted through a security gateway (Torkura et al., 2017), but this brought great performance overhead, and violation of the distribution with potential bottlenecks, and any approach that depends on code instrumentation, as in (Mayer and Weinreich, 2018), brings additional development difficulty and overhead.

Another runtime approach is to utilize the underlying containerization engine. Since microservices are often deployed using containers, this can be a valuable source of information about the application's architecture. (Granchelli et al., 2017) query the containerization framework to retrieve calls between microservices at runtime. The extracted calls are used along with deployment metadata collected from service descriptors to create an architectural model for a microservice system. This approach is limited in extracting further system concerns because not all information is available through the containerization engine, especially information relating to how the application represents and operates on data.

These runtime analysis approaches benefit from being able to access runtime data (like performance metrics and real-time service calls), but they require the system to be deployed and running and cannot operate on the source code alone. For these reasons, we turn to static analysis.

2.2 Static SAR

Static analysis can be performed on a system before it is deployed, extracting information from existing artifacts that would otherwise have to be manually analyzed. In particular, analyzing a program's source code has played a part in formal verification of a system's correctness (Chlipala, 2013; Albert et al., 2007)

It has also been applied in the realm of microservices. It has been used to identify calls between microservices to generate security policy automatically (Li et al., 2019). Also, it has been used to analyze monolithic applications to recommend splits for converting to microservices (Eski and Buzluca, 2018). In generating a service dependency graph, (Esparrachiari et al., 2018) posit that source code analysis is not sufficient since the deployment environment may impact the actual dependencies a given deployed module has. Furthermore, (Soldani et al., 2021) analyzes docker configurations to understand the microservice deployment topology graph and then uses this information for dynamic analysis to monitor and determine interactions among components. However, our goal is different from the previous two works; we do not necessarily target every possible call in a system for dependency detection; rather, we find the calls that are part of the application's business logic, and for this purpose, the source code contains sufficient information.

Source code is not the only artifact available for static analysis. Ibrahim et al. use a project's Dockerfiles to search for known security vulnerabilities of the container images being used, which they overlay on the system topology extracted from Docker Compose files to generate an attack graph showing how a security breach could be propagated through a microservice mesh (Ibrahim et al., 2019). This allows the creation of a centralized security concern for the system, but since it does not extend to source code, it cannot include security flaws in the programs deployed in the containers, only flaws with the images themselves. Another static source of information is in the API definitions. Mayer and Weinreich use API definitions generated by Swagger as an input to their architecture generation system, but their system is also dependent on runtime data extracted from calls between services (Mayer and Weinreich, 2018).

Another approach to pre-runtime SAR is to embed a source of information into microservices in their development. For example, (Salvadori et al., 2016) propose creating semantic microservices that expose information about their resources, allowing them to be automatically composed. In this way, a centralized view of microservice communication is always available. However, this approach depends on using a fundamentally different approach to development, and it cannot be used to analyze existing codebases.

2.3 Software Architecture Degradation

Software architecture degradation is a phenomenon where code changes cause the implemented architecture of a system to differ from its planned architecture (Baabad et al., 2020). This process goes by many names, including architecture erosion (Perry and Wolf, 1992), software degeneration (Hochstein and Lindvall, 2005), etc., but the principle is the same: the original intent for the architecture is not kept.

Software architecture degradation has many causes. Baabad et al., in a survey of open-source software systems, identified the most common of these causes. Degradation tends to happen when the rapid system changes cause small issues to go unnoticed when the system grows in complexity through new features or unforeseen design changes. These changes occur when developers have a limited understanding of the overall architecture or are not dedicated long-term to the project. It also happens when pressures of time and deadlines cause suboptimal and ostensibly temporary solutions to become a permanent part of the system (Baabad et al., 2020).

While the issues that tend to cause software architecture degradation overlap with more general issues of code practice, Hochstein and Lindvall note that the actual code that causes the offending changes to the architecture may be functionally correct in and of itself (Hochstein and Lindvall, 2005). Therefore, traditional debuggers and code analysis tools may not be sufficient for combating this issue since the problems only become apparent at higher levels of abstraction than the tools operate at.

Because traditional code analysis tools are insufficient for countering software architecture degradation, specific techniques have been developed. In their study, Baabad et al. sorted these techniques into several broad categories. The four most common categories which contained a majority of the approaches were tools that detected degradation by various metrics (e.g., measuring stability or cohesion/coupling measures), approaches based on defining and enforcing architectural rules, direct methods of detecting problems through heuristics and known code or architecture smells, and methods based on architecture recovery/reconstruction (Baabad et al., 2020). Of the four categories, our approach is most relevant to the last category since it is a SAR-based tool; none of the existing tools in this category are designed to recover the architecture of a microservice-based system.

3 OUR SAR METHOD

The need for our approach is twofold. First, it fills the niche of a SAR tool entirely dependent on static analysis, rather than dynamic analysis, that is specifically suited for microservices. Particularly, it identifies HTTP calls between individual microservices without the need for runtime analysis. These identified calls allow us to extract a view of the system architecture before it is deployed, whereas other methods require a deployed system (Mayer and Weinreich, 2018; Granchelli et al., 2017; Esparrachiari et al., 2018; Torkura et al., 2017). Using this method, developers can get an updated view of the system's service APIs and service interactions as the code changes, rather than waiting for deployments. This API/interaction view was identified by (Mayer and Weinreich, 2017) as one of the most important aspects of a microservice monitoring tool. This view also corresponds to the service viewpoint identified by (Rademacher et al., 2020b) as one of the fundamental viewpoints to be obtained from the SAR process on microservice systems. Furthermore, we include a feature to extract the bounded context for each individual microservice, then combine them into a single context map for the entire system. It will combine entities that are shared across multiple microservices into a single combined entity, preserving all properties and relationships the entity is part of across the entire system. This view corresponds to the domain viewpoint also identified by (Rademacher et al., 2020b), and to our knowledge, ours is the only microservice-oriented SAR tool to include this viewpoint.

The second role is more specific to detecting and fixing software architecture degradation. Microservices, in particular, are vulnerable to architecture degradation; their development is characterized by rapid evolution by developers who specialize in a subset of the microservices of the whole system, perhaps lacking knowledge of the overall view. As discussed previously, these are major risk factors for a system to experience architecture erosion (Baabad et al., 2020). Our method is meant to be a complement to other methods to combat software architecture degradation. As a visual tool, it is useful to get a view of the system architecture as-is, which can be used to compare against the planned system architecture. Developers can use this as a first warning to detect if the architecture has drifted from the original plan.

Our approach takes place in two broad phases. A *context map* is built from the individual bounded contexts extracted from each microservice in the first phase. In the second phase, calls between the services are found to construct a *communication diagram*.

3.1 Context Map

Creating a *context map* consists of two parts. First, each microservice project's bounded context has to be created, and second, those bounded contexts are combined into a context map for the entire system.

Extracting a bounded context from each microservice requires each service to be analyzed with static analysis. We perform this primarily with source code analysis, although bytecode analysis is also possible for languages with intermediate bytecode representations; we perform bytecode analysis for Java projects, for example. The goal of this first analysis is to extract a list of all local classes used in the project.

Once the classes have been identified, the next goal is to determine which of them are serving as data entities and which are not; for example, classes acting as REST controllers or internal services need to be filtered out. This is where we use enterprise standards to our advantage; development frameworks use standard components and constructs involving annotation descriptors that indicate a class's semantic purpose. For instance, there are multiple standards for persistence, input validation, transaction boundaries, synchronization, layering, and security. We use these descriptors to identify which classes are acting as entities. Even though we reference mostly Java, similar standards are adopted across platforms.

After the entities are identified, the bounded context of the microservice can be built. This is done by identifying the relationships the entities have with each other. These relationships have three different components, which we extract using static analysis: the types involved in the relationship (i.e., the entities that are on either side of the relationship), the multiplicity of the relationship, and the directionality of the relationship. Identifying the types is done on the basis of the type names of the entities' fields, the multiplicity can be determined by whether or not the field is a collection, and its directionality can be determined by whether or not there is a corresponding field in both of the entities involved or in only one entity.

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

3.2 Communication Diagram

After the context map is created, the next step is to create the communication diagram. The code is analyzed again using static analysis to find HTTP calls among the microservices. This consists of two phases: identifying each service's API endpoints and finding where these endpoints are called from other services. With these two pieces of information, we can create a graph showing the paths of communication between services. Regardless of the method of extracting this information, certain metadata must be collected regarding the endpoints and calls; this includes the path, HTTP method, parameters, and return type (or the expected return type for a call).

Our method depends on using the standardized formats that enterprise standards use to encode this information. In enterprise applications, exposing endpoints is most commonly done in code using functions or annotations specific to a framework or library; this means the definitions will appear consistent each time they appear in code, so code analysis can be used to identify the metadata about defined endpoints. Likewise, in enterprise applications, HTTP requests are made from an HTTP client. The client may be part of the same framework used to define the endpoints, but developers will likely use the same client for all calls in the system, even if it is not. Therefore, code analysis again can identify the metadata about every request in the system by finding the function call formats appropriate to the known HTTP library. Once the requests have been identified, they can be matched with the catalog of known service endpoints collected earlier; a match means there is a communication path between the two services.

By extracting the communication diagram and aggregated context diagram, we now have reconstructed the system's architecture in how the services communicate among themselves and how the system treats its data. Next, we will describe how we implemented our method.

3.3 Prophet Prototype

To demonstrate the proposed approach, we implemented Prophet, an implementation of our approach suited for Java projects using the Spring framework for creating microservices. Prophet operates in three phases: source selection, context map generation, and communication diagram generation.

Prophet takes as input a GitHub repository containing the microservices. It downloads the repository and generates a list of the directories of each individual microservice project.

To create the bounded context of an individual microservice, Prophet gets all local classes in the project using a source code analyzer; this results in the analysis context. Next, we filter this list down to classes serving as data entities, which we call the system context. For this demonstration, the filtering is done using persistence annotations, including JPA standard entity annotations, and annotations from Lombok, a tool for automatically creating entity objects. The relationships and their multiplicity and directionality are determined as described above. The entities with their fields and relationships are form the bounded context.

For the context map of the whole system, the entities of the different microservices and their fields are merged based on name similarity, as described above. To detect name similarity, we use the WS4J project, which uses the WordNet project (Christiane and Brown, 2005). This creates a single context map for the whole system.

For the communication diagram, we use the API endpoints and their calls. Prophet scans for JAX-RS annotations that define endpoints, combining class-level and method-level annotations to create a definition for each endpoint that includes its path, method, parameters, and return type. We then scan each microservice for the use of the Spring Boot REST client (or alternative) to find HTTP calls between the services. If the call matches the path, parameters, method, and return type, it is added to the diagram as an edge between the two microservice nodes.

Once the bounded contexts, context map, and communication diagram are completed, they are displayed to the user as graphs. The bounded contexts are displayed as class diagrams showing each class's fields and the relationships between them, and the communication diagram is shown as a graph with the microservices as vertices and the calls between them as edges, labeled with the metadata about the call, including the HTTP method, parameters, and return type. These diagrams are displayed in a dashboard that could be expanded in the future to include views of other concerns, as well as runtime analytics.

Limitations. There are currently several limitations to our implementation to address. One limitation is that it only searches for a single format of defined HTTP endpoints and requests, in this case, those defined by the functions included with the Spring framework. Other formats can be substituted for other frameworks with minimal refactoring, and a slightly more extensive set of changes could define a different target framework for each microservice while still retaining the same overall method. Another limitation is that we currently only target systems using the Java language. Again, other targets could be addressed as long as an appropriate parser exists for the language; however, unlike targeting a different framework, different languages would require non-trivial changes to the underlying logic as the current approach depends on Java's annotations to identify entities and the desired language may not share that feature.

4 CASE STUDY

To demonstrate the effectiveness of Prophet as a tool for SAR, we ran it on the Train Ticket benchmark microservice system introduced in (Zhou et al., 2018). The system consists of 41 microservices; of these, 36 microservices are Java-based, and they contain 27,259 lines of Java code, counted by the CLOC (Count Lines of Code) tool. The benchmark was analyzed using a MacBook Pro with a 2.9 GHz Quad-Core Intel Core i7 processor and 16 GB of RAM. Prophet took 2 minutes and 37 seconds on this device, to clone, analyze, and generate the graphs for the repository. To manually analyze the project and enumerate the entities and inter-service calls, it took approximately 1.5 hours.

4.1 Results

For class entities, manual analysis of the source code revealed 64 distinct entities in the project, and the

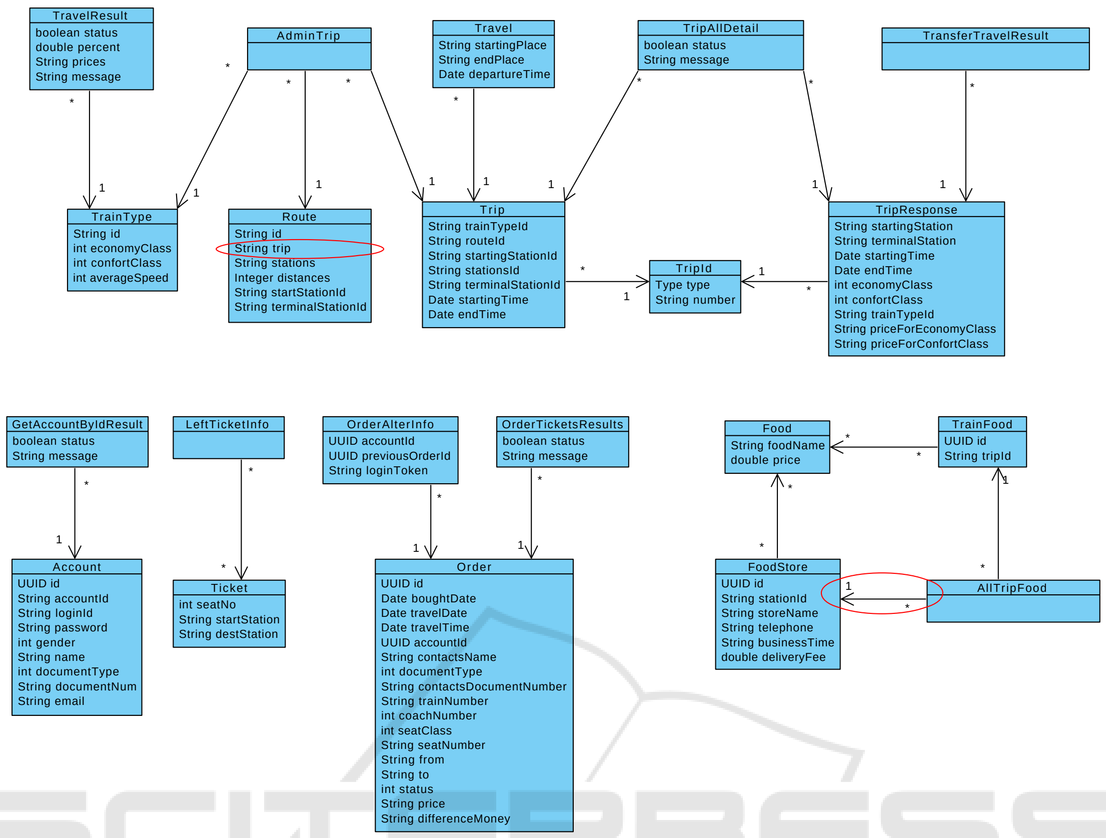

Figure 1: The class diagram from manual analysis of the benchmark. Elements not found by Prophet circled in red.

combined relationships between those entities are shown in Figure 1 (we included in the diagrams only those entities that were involved in relationships due to space constraints). This includes relationships that are spread across several services but exclude relationships not syntactically defined by the code, e.g., a string that is the ID of another entity without any further connection is not included. Prophet was able to recover all of the entities, their properties, and their relationships, with the exception of one property and one relationship (highlighted with red circles in Figure 1). Prophet also reported five additional classes that were not considered entities during the manual inspection because they were being used as data transfer objects (DTO) instead of entities.

For interservice communication, the manual analysis showed 153 connections between the services and their endpoints; separate calls. A subset of these connections are recreated in Figure 2 (due to space constraints, the entire communication graph is not included). Prophet successfully identified 135 of these, missing 18 calls. The missing calls were all due to those calls choosing from multiple potential URLs, an ambiguity our tool was not designed to detect. All other calls always used a single URL. The missing edges in the Prophet result are highlighted with red circles in the manual result (Figure 2).

4.2 Threats to Validity

To discuss the threats to the validity of our study, we will mention the internal and external threats and the measures taken to mitigate them.

Internal threats to validity. The primary internal threat to our study's validity is the possibility that the manually reconstructed architecture is incomplete and thus provides a faulty reference for our tool's reconstruction. To combat this, the manual reconstruction was performed before the rest of the study was conducted, and two developers separately performed a manual analysis to ensure the results were complete.

External threats to validity. One external concern is that the benchmark we tested on is not broadly applicable. To mitigate this, the microservice benchmark we chose was intended to be used as a basis for microservice research and so was designed using common microservice designs, ensuring that methods applied to it are generalizable to other systems. Another threat is that the method itself will not generalize. Prophet is currently limited to discovering

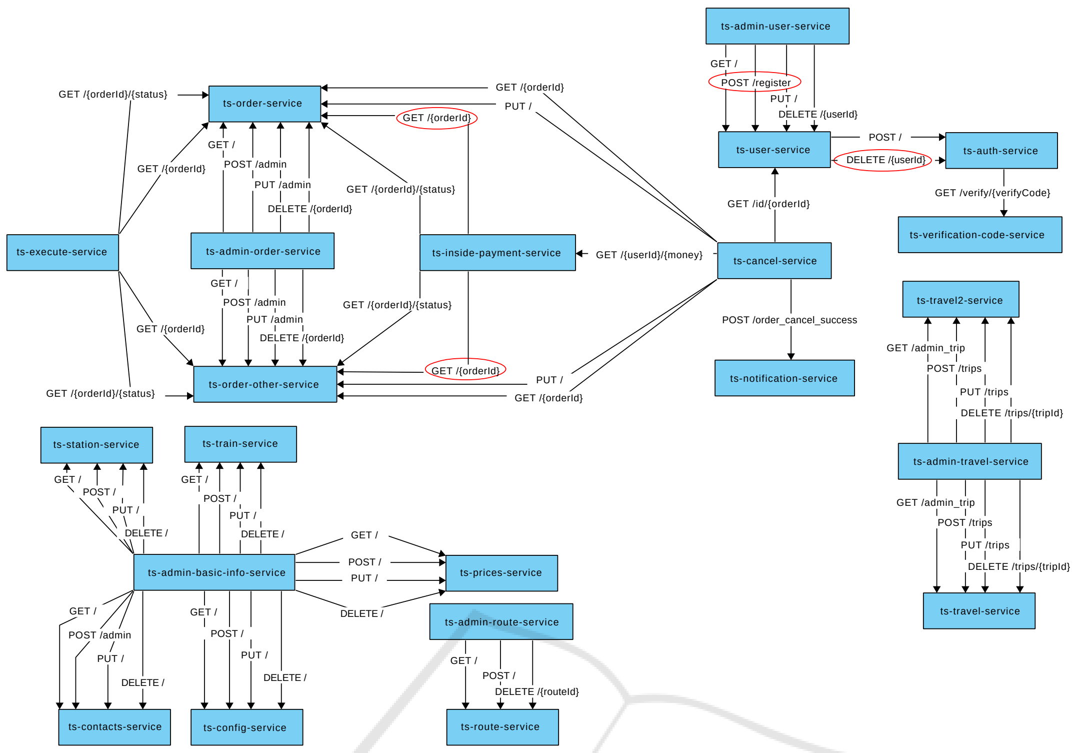

Figure 2: The communication diagram from manual analysis of the benchmark. Elements not found by Prophet circled in red.

entities defined using JPA or Lombok annotations, as described above, and it only finds communication between services that use Spring Boot's REST client. However, this is only a limitation in implementation and not the underlying method. Different implementations can target different frameworks and HTTP clients simply by adjusting what pattern of code is identified. This method is broadly applicable to any microservice-based system that uses similarly-named entities across its microservices, facilitates direct communication between those services.

5 CONCLUSIONS

We have presented a method to analyze microservice systems to generate their up-to-date data models and communication diagrams. This method provides several contributions. The analysis is fast and re-run upon project change. It effectively serves as a centralized source of documentation of decentralized cloud-native systems. This makes it easier for development teams to maintain reliable, up-to-date documentation of large, microservice projects even as it evolves. As demonstrated in the case study, our approach can accurately extract this information with relatively few errors. Finally, this approach can gain a quick overview of the structure of a project for developers who are unfamiliar with it.

Future work on the Prophet tool includes research on language agnosticism to enable other language integration. Prophet presents a basis for automated reasoning. Other system aspects, i.e., security, could be targeted. We will also research novel visualization methods since two-dimensional space becomes insufficient for microservices.

The code for the Prophet can be found at GitHub: https://github.com/cloudhubs/prophet-utils. Besides, we host a supportive web with real-time analysis at https://cloudhubs.ecs.baylor.edu/prophet/

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1854049 and a grant from Red Hat Research.

REFERENCES

Albert, E., Gómez-Zamalloa, M., Hubert, L., and Puebla, G. (2007). Verification of java bytecode using analysis and transformation of logic programs. In Hanus, M., editor, *Practical Aspects of Declarative Languages*, pages 124–139, Berlin, Heidelberg. Springer.

Baabad, A., Zulzalil, H. B., Hassan, S., and Baharom, S. B. (2020). Software architecture degradation in open source software: A systematic literature review. *IEEE Access*, 8:173681–173709.

Bogner, J., Fritzsch, J., Wagner, S., and Zimmermann, A. (2021). Industry practices and challenges for the evolvability assurance of microservices. *Empirical Software Engineering*, 26(5):104.

Bushong, V., Das, D., Al Maruf, A., and Cerny, T. (2021). Using static analysis to address microservice architecture reconstruction. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1199–1201.

Cerny, T., Donahoo, M. J., and Trnka, M. (2018). Contextual understanding of microservice architecture: Current and future directions. *SIGAPP Appl. Comput. Rev.*, 17(4):29–45.

Chlipala, A. (2013). The bedrock structured programming system: Combining generative metaprogramming and hoare logic in an extensible program verifier. *SIGPLAN Not.*, 48(9):391–402.

Christiane, F. and Brown, K. (2005). Wordnet and wordnets. In *Encyclopedia of Language and Linguistics*, pages 665–670. Oxford: Elsevier.

Cinque, M., Cotroneo, D., Corte, R. D., and Pecchia, A. (2019). A framework for on-line timing error detection in software systems. *Future Generation Computer Systems*, 90:521 – 538.

Eski, S. and Buzluca, F. (2018). An automatic extraction approach: Transition to microservices architecture from monolithic application. In *Proceedings of the 19th International Conference on Agile Software Development: Companion*, XP '18, New York, NY, USA. Association for Computing Machinery.

Esparrachiari, S., Reilly, T., and Rentz, A. (2018). Tracking and controlling microservice dependencies. *Queue*, 16(4):10:44–10:65.

Evans, E. (2014). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley.

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

Hochstein, L. and Lindvall, M. (2005). Combating architectural degeneration: A survey. *Inf. Softw. Technol.*, 47(10):643–656.

Ibrahim, A., Bozhinoski, S., and Pretschner, A. (2019). Attack graph generation for microservice architecture. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, pages 1235–1242, New York, USA. Association for Computing Machinery.

Li, X., Chen, Y., and Lin, Z. (2019). Towards automated inter-service authorization for microservice applications. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, SIGCOMM Posters and Demos '19, pages 3–5, New York, NY, USA. Association for Computing Machinery.

Ma, S., Fan, C., Chuang, Y., Lee, W., Lee, S., and Hsueh, N. (2018). Using service dependency graph to analyze and test microservices. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 02, pages 81–86.

Ma, S., Liu, I., Chen, C., Lin, J., and Hsueh, N. (2019). Version-based microservice analysis, monitoring, and visualization. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 165–172.

Mayer, B. and Weinreich, R. (2017). A dashboard for microservice monitoring and management. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 66–69.

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

NGINX, Inc. (2015). The Future of Application Development and Delivery Is Now Containers and Microservices Are Hitting the Mainstream.

Perry, D. E. and Wolf, A. L. (1992). Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52.

Rademacher, F., Sachweh, S., and Zündorf, A. (2020a). A modeling method for systematic architecture reconstruction of microservice-based software systems. In Nurcan, S., Reinhartz-Berger, I., Soffer, P., and Zdravkovic, J., editors, *Enterprise, Business-Process and Information Systems Modeling*, pages 311–326, Cham. Springer International Publishing.

Rademacher, F., Sachweh, S., and Zündorf, A. (2020b). A modeling method for systematic architecture reconstruction of microservice-based software systems. In Nurcan, S., Reinhartz-Berger, I., Soffer, P., and Zdravkovic, J., editors, *Enterprise, Business-Process and Information Systems Modeling*, pages 311–326, Cham. Springer International Publishing.

Salvadori, I., Huf, A., Mello, R. d. S., and Siqueira, F. (2016). Publishing linked data through semantic microservices composition. In *Proceedings of the 18th International Conference on Information Integration and Web-Based Applications and Services*, iiWAS '16, pages 443–452, New York, NY, USA. Association for Computing Machinery.

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

Thalheim, J., Rodrigues, A., Akkus, I. E., Bhatotia, P., Chen, R., Viswanath, B., Jiao, L., and Fetzer, C. (2017). Sieve: Actionable insights from monitored metrics in distributed systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, pages 14–27, New York, NY, USA. ACM.

Torkura, K. A., Sukmana, M. I., and Meinel, C. (2017). Integrating continuous security assessments in microservices and cloud native applications. In *Proceedings of The10th International Conference on Utility and Cloud Computing*, UCC '17, pages 171–180, New York, USA. Association for Computing Machinery.

Zhou, X., Peng, X., Xie, T., Sun, J., Xu, C., Ji, C., and Zhao, W. (2018). Benchmarking microservice systems for software engineering research. In Chaudron, M., Crnkovic, I., Chechik, M., and Harman, M., editors, *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 323–324. ACM.