Extracting the Architecture of Microservices: An Approach for Explainability and Traceability

Pierre-Jean Quéval[1,2] and Uwe Zdun[1]

[1] University of Vienna, Faculty of Computer Science, Vienna, Austria
[2] University of Vienna, Doctoral School Computer Science, Vienna, Austria

Abstract. The polyglot nature of microservice architectures and the need for high reliability in security analyses pose unique challenges that existing approaches to automatic architecture recovery often fail to address. This article proposes an approach for extracting detailed architecture models from polyglot microservice source code focusing on explainability and traceability. The approach involves abstracting code navigation as a tree structure, using an exploratory algorithm to detect architectural aspects, and providing a set of generic detectors as input. The architecture models are automatically annotated with detailed information that makes them useful for architecture conformance checking and violation fixing. Our case studies of microservice software systems validate the usefulness of our approach, providing insights into its completeness, accuracy, and effectiveness for software architecture tasks.

Keywords: Architecture · Explainability · Microservices · Polyglot

1 Introduction

Understanding software architecture is essential for ensuring software systems' maintainability, scalability, and evolution. However, with the increasing complexity and diversity of modern software systems, extracting a comprehensive view of the architecture has become challenging. This is particularly true for polyglot microservice architectures [6, 5], which are becoming more prevalent in the industry. Existing approaches to automatic architecture recovery [3, 11] must address the unique challenges such architectures pose, leading to incomplete or inaccurate architecture models. In addition, existing approaches often need more explainability and traceability for extracting models from the source code.

To address these challenges, we present an approach for extracting detailed architecture models with security annotations from polyglot microservice source code, focusing on explainability and traceability, making them useful for tasks such as architecture conformance checking [9] and violation fixing [7] concerning microservice-specific patterns and best practices.

2 Related Works

Software architecture reconstruction is particularly challenging for decentralized and polyglot systems such as microservices [2]. Static analysis can be performed on a system before deployment, extracting information from existing artifacts [3, 2]. Such analyses can help to provide formal verification, generate test cases, support program or architecture comprehension (e.g., by generating UML models [10]), and maintain programs (e.g., by identifying code clones [12]). Rademacher et al. [11] propose a model for microservices that address their increased architectural complexity. Bushong et al. [1] present a method to analyze a microservice mesh and generate a communication diagram, context map, and microservice-specific limited contexts. Granchelli et al. [4] introduce MicroART, an architecture recovery approach for microservice-based systems. Ntentos et al. [8] extract an accurate architecture model abstraction approach for understanding component architecture models of highly polyglot systems. Like these studies, our study focuses on static analyses and polyglot systems, but in contrast, our approach aims to support traceability and explainability.

3 Our Approach

Our approach splits the process of extracting a software architecture model into three independent and decoupled steps. It allows us to work on the different concerns involved in each step separately and achieve better control and accuracy:

1. **Tree abstraction:** By abstracting ubiquitous structures, like folder hierarchies, lines in a text file, nested brackets inside a code file, or widespread file formats like XML or YAML, the detection logic can be expressed agnostically and applied to various languages and paradigms.
2. **Exploration:** The core of our approach is the exploration of the source code based on a minimal set of generic and configurable detectors representing the knowledge about the project in a concise and readable manner.
3. **Scan:** By decoupling the generation of a specific representation from the detection, the detection logic is focused on architectural and security features common to many projects rather than on the specific concerns of a single analysis.

3.1 Tree Structure Abstraction

In the tree structure abstraction approach, we use a TreeReader class to represent the tree's current position and navigate to other nodes in the tree. The TreeReader class has three public methods: MoveDown() moves the reader to the first child of the current node, ReadNext() reads the next sibling, and MoveUp() moves the reader to the parent node. The Value property holds the content of the current node as a string, which works well for handling string-based elements in source code.

The Path property represents the path to the current node from the root. A Path itself consists of two attributes. The Name attribute serves as an identifying value for a specific type of navigation, e.g., a folder hierarchy. The Children attribute represents which child node was selected from the current node in the path. For example, a path $P = \{$ *"Directory" [2, 1, 2]* $\}$ means, from the root of a folder hierarchy, move to the second child node, then move to the first child node of the previous node, and finally move to the second child node of the previous node.

3.2 Generic Exploratory Algorithm

Overview Rather than directly generating a diagram, our generic exploratory algorithm aims to create a model of the source code that captures its architectural aspects. The model is represented as a tree structure, where each node in the tree corresponds to a localized part of the source code, known as an "Instance." An instance can be a function, a class, a file, or any specific part of the source code with a specific location. A branch in this tree is the Path from a parent Instance to its children. An Instance can have many Aspects associated with it. Aspects are semantic elements representing a particular characteristic or property of the Instance that are relevant to the analysis.

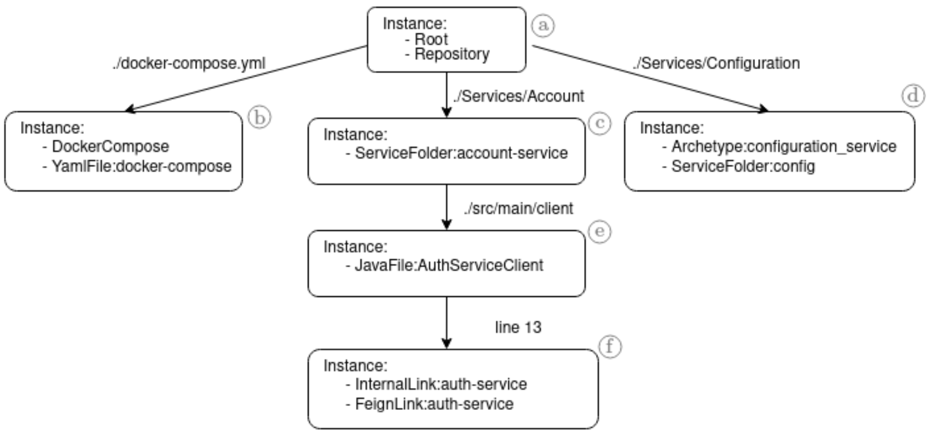

Fig. 1. Simplified model of a Repository

Figure 1 is an example of a simple model. It pinpoints six locations of interest in the source code:

- the root folder of the repository (a),
- a docker-compose file (b),
- two folders corresponding to a specific service. (c,d),
- a Java file (e), and
- a specific line in the Java file that declares a link (f).

An Aspect is a simple label that may contain a value. For instance, (e) has the aspect "JavaFile:AuthServiceClient," which can be understood as: "Here is an instance of a Java file named AuthServiceClient." An instance can exhibit more than one Aspect, e.g., as it is the case for the Instance (d), which is both the folder of a service called "config" and the root of a specific archetype called "config_service." An Instance is automatically created for the root repository with the Aspect "root." Still, apart from this, labels and values of Aspects have no further meaning for the algorithm. They only serve as inputs and outputs for the detectors.

If the location of an Instance is contained within the location of another Instance, it is considered its child, and the branch contains the Path from the parent Instance to the child. The Paths in Figure 1 are written in a readable format for better reading, such as the link "line 13" between (e) and (f), which would be a Path structure $P = \{$ *"Text file," [12]* $\}$ with zero-based indexing.

Detector-based algorithm The purpose of a detector is, from an Instance with a given Aspect, to detect another Instance with another Aspect. Using Figure 1, creating the Instance (f) from the Instance (e) could be done with a detector like "From a *JavaFile*, detect a *FeignLink* by using a *Text File* Tree Reader and detect a node whose value satisfies *a regex expression.*"

The algorithm in Figure 2 can thus extract a complete model, starting from a root instance. Note that the detectors are automatically ordered according to their dependencies. The exploratory algorithm also makes tracing how each aspect was detected straightforward. When an aspect is detected, our tool keeps a trace of the originating detector in the instance.

3.3 Scanner

The next step is to scan the model into a specific format. The Component and Connector (C&C) view is generated from the model, a high-level abstraction of the system's components and their relationships. The C&C view clearly and concisely represents the system's architecture, allowing for more straightforward analysis and evaluation. The scanner for the C&C also receives parameters in input, such as which aspects to include or what constitutes a component. This two-step process maintains a decoupling between the detection of architectural aspects and the specific view of the architecture.

4 Case study

We based our study on our prior work [13], which studies case studies of 10 microservices from Github repositories to automatically assess their conformance to a set of Architectural Design Decisions. The Component & Connector views were manually modeled based on line-by-line inspection of their source, and industrial experts confirmed the assessment scheme conformed to the most widely used security tactics for microservices today. Using the Component & Connector views of the study [3], we ensure that we have a ground truth of models that

[3] https://zenodo.org/record/6424722

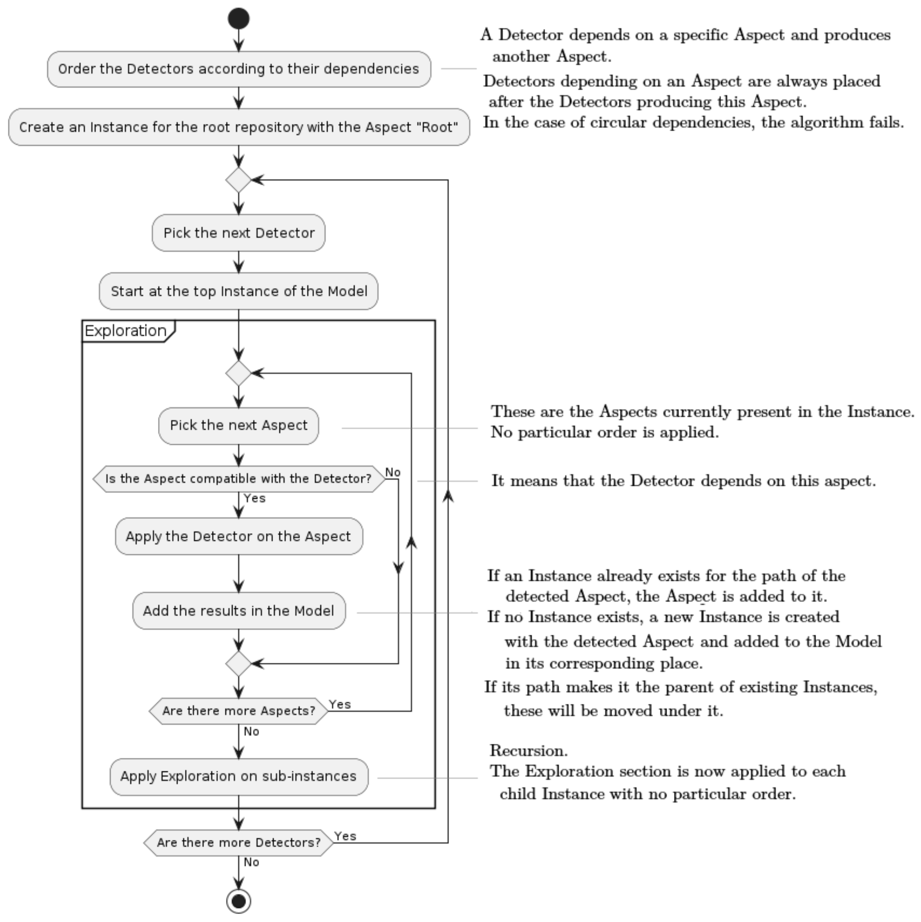
Order the Detectors according to their dependencies
A Detector depends on a specific Aspect and produces another Aspect.
Detectors depending on an Aspect are always placed after the Detectors producing this Aspect.
In the case of circular dependencies, the algorithm fails.
Create an Instance for the root repository with the Aspect "Root"
Pick the next Detector
Start at the top Instance of the Model
Exploration
Pick the next Aspect
These are the Aspects currently present in the Instance.
No particular order is applied.
Is the Aspect compatible with the Detector?
No
Yes
It means that the Detector depends on this aspect.
Apply the Detector on the Aspect
Add the results in the Model
If an Instance already exists for the path of the detected Aspect, the Aspect is added to it.
If no Instance exists, a new Instance is created with the detected Aspect and added to the Model in its corresponding place.
If its path makes it the parent of existing Instances, these will be moved under it.
Are there more Aspects?
Yes
No
Apply Exploration on sub-instances
Recursion.
The Exploration section is now applied to each child Instance with no particular order.
Are there more Detectors?
Yes
No

Fig. 2. Exploration

are accurate and relevant to the security analysis of the given repositories. The study's full results cannot be directly presented within the scope of this short paper but a replication package is provided online[4].

Our case study focuses on analyzing the architecture of the Piggymetrics application, a widely known microservices-based system. Piggymetrics is a financial management platform demonstrating the complexities and challenges associated with polyglot microservice architectures. We aim to answer the following questions:

- **RQ1** *Can the approach extract an accurate Component & Connector View?* This would mean extracting the same components and connectors as the manual view.

[4] https://zenodo.org/record/8100928

– **RQ2** *Can the approach extract accurate security annotations?* This would mean extracting the same annotations as the manual view.
– **RQ3** *Can the approach explain its result?* This would mean providing each annotation a link to a location in the source code.

4.1 Comparison of manually derived and automatically extracted views

The automatic extraction detected the same components as the manual one, but the names are less informative. For instance, a component referred to as "Oauth2 Server" in the manual model becomes "auth-service" in the automatic one. This is an interesting finding since formalizing the naming conventions was not considered when starting this study, but it would be beneficial in improving the usefulness of the generated views. This work would primarily affect the scanner transforming the model into a Component & Connectors view. The extracted annotations for the components are the same, which is not highly significant since we used the manual view as a reference for the desired features. However, it shows that these features could be translated neatly into our detectors.

Our automatic tool detected an additional link among the connectors, from "auth-service" to "config", compared to the manual one. Since it was adequately justified in the traces, we consider this a correct result by our automatic extractor. The most noteworthy difference lies in the security annotations on the connectors. The manual view presents not only the intrinsic attributes of the connector, e.g., that a given connector is a database connection and uses plain text credentials, but also contextual information, like "authentication scope / all request" that was often missing in the automatic view.

The automatic extraction of features that can be traced back to a location in the source code is more straightforward than identifying features deduced from multiple sources of information in the repository. While some of these contextual features are also automatically extracted, e.g., to identify internal links between services or external configuration files, these rely on multiple detectors and require more fine-tuning.

4.2 Detectors

The automatic generation of a model for Piggymetrics requires 65 atomic detectors, not counting the one required to transform the model into a Component and Connector View. Most are very generic, for instance, identifying an XML file or an ArtefactId inside a POM file. While not each of these detectors will be relevant to every single source code, they are, as a whole, describing patterns widely prevalent among microservices projects.

20 detectors were more specific, encoding information related to the Spring framework used in Piggymetrics. For instance: "*@EnableDiscoveryClient* in a Java file denotes a registry link." These specific detectors were of two types: (1) recognizing one or more specific values in a single node; (2) recognizing a specific path for a node. That detecting specific features is so straightforward is promising since it opens the possibility of generating these detectors automatically.

4.3 Traces

The purpose of the trace is to allow a human to quickly verify a specific element of the view. Each annotation, whether from a component or a connector, is listed with the locations in the source code explaining it. Traces are exported as a Json file, which list the code location for each element's annotation (component or link.) For instance: "*element:* gateway { *annotation:* csrf_scope_all_requests { *location:* /piggymetrics/auth-service/[...]/auth/config/WebSecurityConfig.java, line 28"

These work well for most annotations but would become cumbersome for those requiring multiple detectors, as the number of concerned locations can quickly become combinatorially too important; therefore, these annotations are currently not traced.

5 Discussion

As of **RQ1**, the case study demonstrated the ability of our approach to extract a component and connector view from the source code. Our approach provided the needed detectors to produce an accurate component and connector view of the system's architecture. The only shortcoming here is the absence of a naming system that makes the names less informative than in a manual view.

Considering **RQ2** and **RQ3**, while we achieve promising results, they are less satisfying. Some security features can only be detected by analyzing multiple locations in the source code. It makes the definition of detectors and the tracing less straightforward.

Future research could explore ways to address the limitations of our approach and further improve its accuracy and scalability. One promising direction is to rely on a common core of generic detectors and automatically produce the specifics with an analyzer and a higher-level description of the expectations. Such an improvement could reduce the dependence on manually crafted detectors and increase the system's coverage under study.

6 Conclusion

This paper presented an approach for extracting a component and connector view from source code. We evaluated our approach using case studies in microservices and demonstrated its effectiveness in extracting a clear and understandable representation of the system's architecture. Our approach has limitations, particularly its dependence on the quality of the detectors provided as input. Future research could address these limitations by automatically generating the detectors from a higher-level expectation description and may make our approach a helpful tool for software architects and developers.

7 Acknowledgements

Our work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 952647 (AssureMOSS project). This work was supported by: FWF (Austrian Science Fund) project API-ACE: I 4268; FWF (Austrian Science Fund) project IAC^2: I 4731-N.

References

1. Bushong, V., Das, D., Al Maruf, A., Cerny, T.: Using static analysis to address microservice architecture reconstruction. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE
2. Cerny, T., Abdelfattah, A.S., Bushong, V., Al Maruf, A., Taibi, D.: Microservice architecture reconstruction and visualization techniques: A review. In: 2022 IEEE International Conference on Service-Oriented System Engineering (SOSE). IEEE
3. Ducasse, S., Pollet, D.: Software architecture reconstruction: A process-oriented taxonomy. IEEE Transactions on Software Engineering
4. 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). IEEE
5. Hasselbring, W., Steinacker, G.: Microservice architectures for scalability, agility and reliability in e-commerce. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW). IEEE
6. Newman, S.: Building microservices. O'Reilly Media, Inc.
7. Ntentos, E., Zdun, U., Plakidas, K., Geiger, S.: Semi-automatic feedback for improving architecture conformance to microservice patterns and practices. In: 2021 IEEE 18th International Conference on Software Architecture (ICSA). IEEE
8. Ntentos, E., Zdun, U., Plakidas, K., Genfer, P., Geiger, S., Meixner, S., Hasselbring, W.: Detector-based component model abstraction for microservice-based systems. Computing
9. Ntentos, E., Zdun, U., Plakidas, K., Meixner, S., Geiger, S.: Assessing architecture conformance to coupling-related patterns and practices in microservices. In: Software Architecture: 14th European Conference, ECSA 2020, L'Aquila, Italy, September 14–18, 2020, Proceedings 14. Springer
10. Papotti, P.E., do Prado, A.F., de Souza, W.L.: Reducing time and effort in legacy systems reengineering to mdd using metaprogramming. In: Proceedings of the 2012 ACM Research in Applied Computation Symposium
11. 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: 21st International Conference, BPMDS 2020, 25th International Conference, EMMSAD 2020, Held at CAiSE 2020, Grenoble, France, June 8–9, 2020, Proceedings 21. Springer
12. Rattan, D., Bhatia, R., Singh, M.: Software clone detection: A systematic review. Information and Software Technology **55**(7)
13. Zdun, U., Queval, P.J., Simhandl, G., Scandariato, R., Chakravarty, S., Jelic, M., Jovanovic, A.: Microservice security metrics for secure communication, identity management, and observability. ACM Trans. Softw. Eng. Methodol.