Qualitative Evaluation of Dependency Graph Representativeness

Tuomas Nurmela[2], Petteri Nevavuori[1], and Imran Rahman[1]

[1] Tampere University, Korkeakoulunkatu 1, Tampere, Finland
[2] Aalto University, Otakaari 24, Espoo, Finland

Abstract. Background: Enterprise application and open source software (OSS) platform and infrastructure projects are often today agile time-boxed projects. To enable project scaling, microservices software architecture (MSA) is considered to enable autonomous cross-functional teams. MSA results to loosely coupled services which communicate via well-designed APIs. Previous research on automated extraction of Microservice Dependency Graphs (MDGs) could provide means of reducing this documentation effort.
Aims: The aim of the study was to look at the MDG representativeness of a Spinnaker OSS project micro-services-based software architecture and MDG, providing assessment of possibilities in using MDGs for documenting microservices-based software architectures.
Method: The study uses a qualitative approach to evaluate the MDG representativeness of software architecture description. Evaluation is done through assessment of limitations, issues and future development possibilities.
Results: MDG of Spinnaker OSS is extracted with an automation tool and contrasted to the software architecture as described on OSS project documentation. Compile-time MDG description and runtime focused documented software architecture lead to limitations in MDG rpresentativiness.
Conclusions: Focusing on a particular OSS microservices project, the MDG extraction through static code analysis limits to compile-time information. Limitations in capturing inter-service communication at runtime to describe key architectural views of software architecture lead to a need to look for complementing approaches.

1 Introduction

Agile and devops projects are typical in enterprise applications and development of open source software (OSS) infrastructure and platform software. These projects are time-boxed, focused on working code over documentation [6]. These projects have started to use microservices architectural style [8], to better support e.g. concurrent development and evolutionary architecture. These attributes (time-boxed projects, focus on working code, evolution support with faster cycle time) put pressure on documenting and communicating through architecture

descriptions. Therefore, the capability to automatically extract meaningful documentation that can be used towards multiple stakeholders to cover their concerns is central. Architectural documentation is considered to be documentation that takes into account different stakeholders and their concerns.

The focus of our research was to evaluate the implementation and the outputs of the *Microservice Dependency Graph* (MDG) [3][10] by scrutinizing on a distinct microservice architecture. Effectively we perform verification and validation by comparing the MDG outputs to a documented microservice architecture. The evaluation architecture was drawn from *Microservice Dependency Graph Dataset* (MDGDS) [4], a dataset of open-source software projects employing microservice architecture hosted in Github. From MDGDS we chose *Spinnaker* [5] as the evaluation architecture.

Microservices are independenently deployable and maintanable small services often utilized via explicitly defined application programming interfaces (APIs). They have well defined and constrained bounding contexts that enable reuse and development without having to change the achitecture of the systems utilizing them. The development of the microservice architecture paradigm has aided in de-coupling the service-oriented tangled software architectures towards smaller autonomous units. However, microservices in themselves are by no means a siler bullet solution: the approach is fairly new with still evolving set of design patterns [13]. At the same time multiple anti-patterns and bad-smells, whether idenfied as design or architectural smells telling of recurring and possibly deeper problems, manifest themselves in the projects [12].

The outline of the paper is as follows. Section 2 covers the target architecture. In Section 3 we then describe the implementation of the MDG. The similarities and discrepansies between the MDG outputs and the target architecture are discussed in Section 4 and future development routes are for MDG are then proposed in Section 5. Lastly, the findings are concluded in Section 6.

2 Evaluation Architecture

Software architecture is commonly considered to describe the system in terms of its components, their basic operational principles and their interconnections [11]. In empiric studies, notions on software architecture vary in time orientation, formality, detail, purpose of architect activity, objective of work and focus on business and technology issues [11]. One way to represent software architecture is through architecture descriptions. These describe a system, taking into account stakeholder concerns framed from different viewpoints to address a particular concerns [1]. However, with cloud native projects, the time-boxed agile and devops approaches can be expected to reduce time for documentation of to satisfy the different contexts and concerns of stakeholders.

For the purpose of this paper, we adopt the notions of 4+1 architecture framework from Kruchthen [9]. Furthermore, given the current nascent arena of CI/CD tools, we focus also on the support of extendibility or adaptability of system. We extract the use case (Kruchten "+1") for Spinnaker based on

available information, describing this in the next subsection. The architecture figure and available description is then contrasted to the 4 architecture views (physical, logical, development and process).

Spinnaker is a Netflix-initiated continuous delivery (CD) tool for modern applications, with support for multiple cloud native infrastructures. The developers note seven key concerns with cloud native infrastructures, in particular 1) credentials management, 2) regional isolation, 3) autoscaling, 4) immutable infrastructure, and 5) service discovery, 6) multi-cloud and 7) abstraction of cloud operations from users [7]. Spinnaker's design indicates a focus on CD taking these concerns into account. This separates it from many of the other popular CD tools (e.g. Jenkins, Gitlab), in which CD is extension of an existing continuous integration tool. The Spinnaker runtime architecture elements are depicted in Figure 1. This architecture mainly depicts high level elements which in themselves do not provide as much information as e.g. UML diagrams, yet provide support for the process view (which covers runtime communication) and logical view (high level functionality).

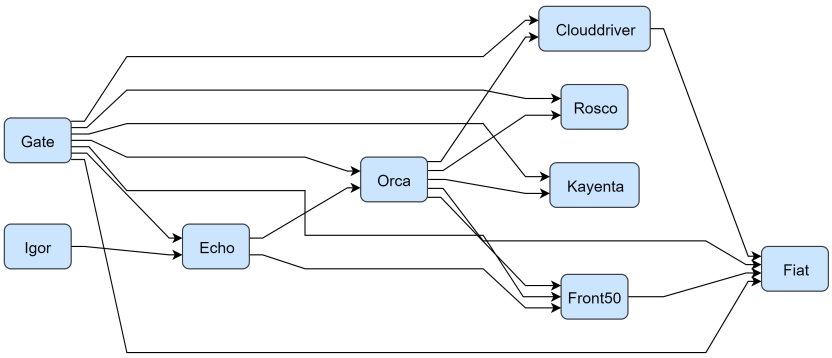

Fig. 1. The microservice dependencies as given by the Spinnaker documentation [2, 5]. Third party and configuration time services are omitted for clarity.

On the other hand, Spinnaker community has been discussing extending the project to cover CI-aspects, as developers have been perceived to convert back to familiar CI-tools which have CD features. Taking into account the context of Spinnaker, its goals, use cases of distributions and central concerns of architecture descriptions for OSS products for which future proofing is important in quickly changing environments, we can describe the Spinnaker architecture elements as follows. The descriptions for the corresponding microservices are provided in Table 1.

It should be noted not all of the developer key concerns noted previously are encapsulated in projects but are more typically cross-cutting concerns. Also, the above excludes a non-key elements of the Spinnaker, namely for runtime Kork (for adapting NetflixOSS to Spring and Spring Boot) and Swabbie (clean up ser-

Element	Description and architectural implication
Gate	API Gateway used for basic Authentication (with Fiat) and routing into microservices. Gate covers the common API gateway pattern [9] of the microservice system.
Igor	Integration layer for CI. Architecturally allows extending product to different CI systems.
CloudDriver	Infrastructure abstraction layer. Architecturally allows extending product to different cloud platforms.
Orca	Orchestrator engine. Key element to cover CD pipelines, stages and tasks.
Echo	Event router. Key element to allow event-based microservices.
Front50	Metadata abstraction layer. Architecturally allows substitution in metadata datastore, reducing dependence to default subsystem (Cassandra)
Rosco	Bakery for images. Implemented on per cloud infrastructure basis. Relies on Packer. Key element to support immutable infrastructure.
Kayenta	Automated canary service. Key functional feature of CD pipelines to support seamless releases to end users.
Deck	UI for Spinnaker. Key element to provide abstraction of cloud operations from user.
Fiat	Authorization service. Key element to support security.

Table 1. Microservice descriptions for the Spinnaker project [2, 5].

vice) as well as for configuration/deployment time element, Halyard (Spinnaker installation, configuration and lifecycle manager tool).

3 Microservice Dependency Graph

Listing 3.1. A single service in Spinnaker's Docker compose file.

```
igor:
  container_name: igor
  env_file: ./compose.env
  environment:
    - SERVICES_ECHO_HOST=echo
    - SERVICES_CLOUDDRIVER_HOST=clouddriver
  image: quay.io/spinnaker/igor:master
  links:
    - redis
    - clouddriver
    - echo
  ports:
    - "8088:8088"
```

The MDG project has been developed to be used for extracting intraproject microservice dependency relations with static code analysis. The implementation, with which the MDGDS has been produced as well, employs a single modes for extracting dependencies as directed graphs using Docker compose files for building the dependecy graph. The MDG implementation attempts using the Docker compose file first as the mode of building the directed dependecy graph. Given that microservice project contains a YAML-file called *docker-compose.yml/.yaml* with either suffix, the file is located and read to a

corresponding class structure. The Docker compose file typically contains information about services, container runtime configuration, linked services and networking configurations. An example of a Docker compose file contents are given in Listing 3.1.

The graph is then formed using the root-level services as nodes and the information about linked services as edges to other nodes. A general depiction of linking the root nodes (*services*) via edges (*links*) is given in Fig. 2. Using the example contents of Listing 3.1, the node would be `igor` and its corresponding edges `redis`, `clouddriver` and `echo`. The dependecy graph for Spinnaker is shown in Fig. 3.

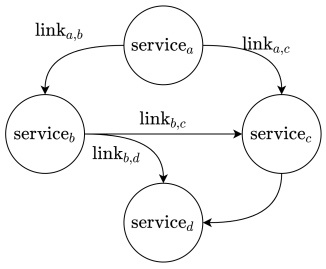

Fig. 2. The directed graph of general linked services is built using Docker compose file and parameters.

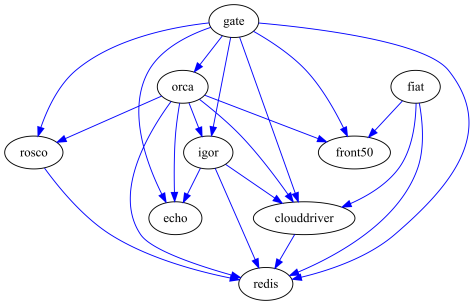

Fig. 3. The directed graph of microservice dependecies for Spinnaker as generated by the MDG.

4 Discussion

At the time of writing the implementation of the MDG considers only the contents of a possibly available Docker compose file for extracting microservice dependecies. As discussed in Sec. 2, however , there are multiple ways to view the architecture. Comparing the the output of the MDG and the architecture of the Spinnaker immediately shows that the two are not in mutual agreement. This doesn't need imply a contradiction or faulty implementation of the MDG. The differing views can be consolidated, if both are considered as distinct vantage points to the project's architecture and dependencies. Considering the fact that MDG used composition-time files for dependency inference, the MDG outputs effectively a deployment-time microservice dependency graph.

5 Future Developments

In microservices architecture the inter-service communication is a key factor. There are quite many ways to implement this communication but it should be

kept in mind that the endpoints are smart and pipes are dumb as Martin Fowler has described in his article [8]. There are mainly two ways to communicate between microservices: 1) synchronous and 2) asynchronous. To determine the architectural pattern it is essential to know what type of communication is done in the given microservice architecture. Albeit currently operating with Docker compose files, the MDG analysis too could be developed further analyzing the REST or synchronous calls between services in a microservice architecture.

From the vantage point of the architecture itself, relying only on REST or synchronous calls, however, is not a good practice and has negative consequence for future development of the architecture itself. For example, only depending on the inner RESTful service calls introduces tight coupling between services. Blocking is another issue to consider when only using REST calls. When invoking a REST service, the service is blocked waiting for a response. This reduces application performance because the thread might be processing other requests.

On the other hand, in asynchronous communication, the clients does not need to wait for a response in a certain time. The asynchronous communication between microservices is done with the use of a lightweight and dumb message broker. The message broker is a centralized component with high availability and does not have any business logic. Some of the major components in asynchronous communication are the message event producer and the event consumer.

In the context of MDG and analyzing dependencies in microservices architecture it would be important addition to check the asyncchronous calls so that any microservices architecture could be analyzed using this tool. To achieve this, there are some features that can be implemented. First the tool should be able to analyze which are the event producers in the microservices. Then it can map the services according to the producers. After that it can look for the event listeners for a specific producer forming a link-like a dependency. As there are several technologies are used to implement the async communication such as RabbitMQ or Kafka it would be a challenge for future work to analyze both implementations in a project. But it can be solved if there is a general pattern matched in both technologies. As the MDG is only analyzing the code statically, a preferable addition would be the ability to determine the dependencies during runtime. This way both static and dynamic analysis results could be combined for increased accuracy in dependency graph of microservice architecture pattern.

6 Conclusions

In this paper we performed evaluation of an open-source microservice dependency extractor tool called *Microservice Dependency Graph* (MDG). The evaluation was performed by selecting a single microservice project from the *Microservice Dependency Graph Dataset* (MDGDS), namely the Spinnaker. We analyzed and described the extraction algorithm of the MDG and the documented microservice architecture of the Spinnaker project. Our focus was on comparing the output of the MDG tool, the microservice dependecy graph, for Spinnaker to its documented architecture. We found out that the inferred de-

pendencies were not in unanimous agreement with the documented architecture due to the MDG extracting the deployment depencies and documentation stating the logical or runtime dependencies. Being a static code analysis tool, the MGD produces effectively a view about microservice dependencies, albeit a view limited to the scope of deployment. To have MGD produce a dependency graph providing insights into runtime microservice dependencies the tool should be developed further to include analysis of inner workings of a project, i.e. intraproject API-calls between microservices and message bus related event dispatchers and listeners.

References

1. Systems and software engineering - architecture description. ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000) pp. 1 –46 (1 2011). https://doi.org/10.1109/IEEESTD.2011.6129467
2. Architecture documentation (2019), https://www.spinnaker.io/reference/architecture/
3. Microservice dependency graph (2019), https://github.com/clowee/MicroDepGraph
4. Microservice dependency graph dataset (2019), https://github.com/clowee/MicroserviceDataset
5. Spinnaker (2019), https://github.com/spinnaker/spinnaker
6. Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M.: Manifesto for Agile Software Development (2001), http://agilemanifesto.org/
7. Burns, E., Feldman, A., Fletcher, R., Lin, T., Reynolds, J., Sanden, C., Wander, L., Zienert, R., Farnham, B., Tokyo, S., Boston, B., Sebastopol, F., Beijing, T.: Continuous Delivery with Spinnaker Fast, Safe, Repeatable Multi-Cloud Deployments. O'Reilly (2018)
8. Fowler, M., Lewis, J.: Microservices (2014), https://martinfowler.com/articles/microservices.html
9. Kruchten, P.: The 4+1 view model of architecture. IEEE Software **12**, 45–50 (11 1995). https://doi.org/10.1109/52.469759
10. Rahman, M.I., Panichella, S., Taibi, D.: A curated dataset of microservices-based systems. In: Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution. (2019)
11. Smolander, K., Rossi, M., Purao, S.: Software architectures: Blueprint, literature, language or decision? EJIS **17**(6), 575–588 (2008). https://doi.org/10.1057/ejis.2008.48
12. Taibi, D., Lenarduzzi, V.: On the Definition of Microservice Bad Smells. IEEE Software **35**(3), 56–62 (2018). https://doi.org/10.1109/MS.2018.2141031
13. Taibi, D., Lenarduzzi, V., Pahl, C.: Architectural patterns for microservices: A systematic mapping study. CLOSER 2018 - Proceedings of the 8th International Conference on Cloud Computing and Services Science **2018-January**(Closer 2018), 221–232 (2018). https://doi.org/10.5220/0006798302210232