Model-driven Generation of Microservice Architectures for Benchmarking Performance and Resilience Engineering Approaches*

Thomas F. Düllmann and André van Hoorn
University of Stuttgart, Institute of Software Technology, Germany

ABSTRACT

Microservice architectures are steadily gaining adoption in industrial practice. At the same time, performance and resilience are important properties that need to be ensured. Even though approaches for performance and resilience have been developed (e.g., for anomaly detection and fault tolerance), there are no benchmarking environments for their evaluation under controlled conditions. In this paper, we propose a generative platform for benchmarking performance and resilience engineering approaches in microservice architectures, comprising an underlying metamodel, a generation platform, and supporting services for workload generation, problem injection, and monitoring.

1. INTRODUCTION

The new microservice architectural style [7] makes use of independent entities being loosely coupled to be more flexible in terms of maintenance and scalability. A guiding principle of microservice architectures is the assumption that misbehavior or outages may happen anytime ("design for failure"). This architectural style also makes it possible to adopt new software engineering paradigms like DevOps [1] which again makes heavy use of approaches like Continuous Deployment [4]. Many methodologies, techniques, and tools have been developed in the recent years to measure and improve the performance and resilience of software systems [8]. To evaluate these approaches in microservice environments, systems under test (SUTs) with representative characteristics (e.g., topology, size) are required. For approaches explicitly considering failures (e.g., detection, diagnosis, prevention, and tolerance), a way to inject them is needed [6].

In this paper, we propose a generative platform for benchmarking performance and resilience engineering approaches in microservice architectures. The approach comprises a metamodel (Section 2) defining the topology of the microservices, a generator (Section 3) for the deployable artifacts of the synthetic microservices, and supporting services (Section 4) for workload generation, problem injection, and monitoring. Figure 1 depicts the aspects that will be covered in this paper. Based on the metamodel (implemented with Eclipse Ecore [2]), a microservice environment can be specified by creating an instance of the said metamodel. After the generation process, the resulting artifacts can be built, deployed, and executed with the supporting services to generate workload, gain monitoring data, inject problems, etc.

Complementary to using real systems, this generative approach provides the possibility to generate microservice environments with specified properties, which can then be used for measurement-based evaluation of performance and resilience engineering approaches.

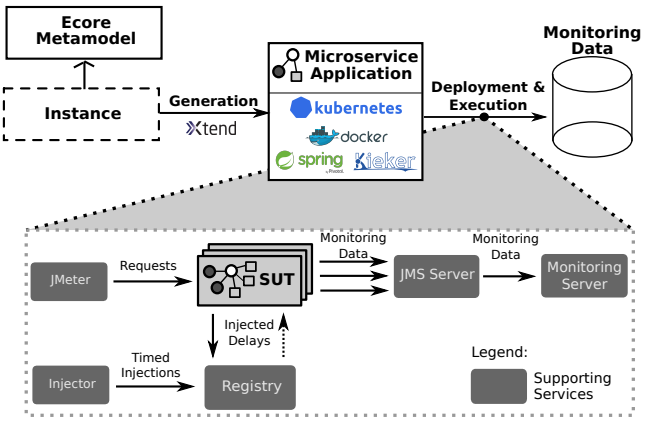

Figure 1: Overview of the generation steps

2. METAMODEL

In Figure 2 the part of the metamodel defining microservice types and microservice instances is depicted. The *MicroserviceRepository* holds all *MicroserviceTypes*, which have *RESTOperations* that define which URL path is mapped to which method of the microservice.

Microservice architectures are designed to be flexible and to be changed frequently. A *Configuration* represents a specific state of a microservice architecture in terms of instances of microservices. The *Configuration* object holds all *Microservices* which are deployed in an *ExecutionEnvironment*. An *ExecutionEnvironment* may be a physical host, a virtual machine or a container. A *Microservice* is an instance of

*This work is partially funded by the Baden-Württemberg Stiftung (ORCAS Project, Elite Programme for Postdocs).

ICPE '17 Companion, April 22-26, 2017, L'Aquila, Italy

DOI: http://dx.doi.org/10.1145/3053600.3053627

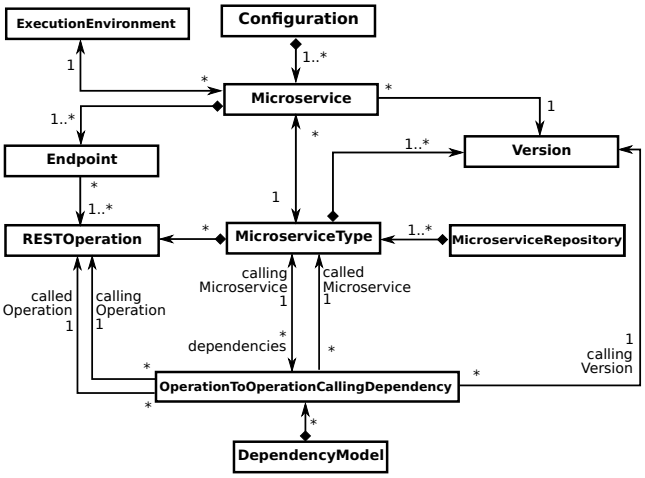

Figure 2: Metamodel excerpt for microservice types, instances, and their dependencies

a *MicroserviceType* with a specific *Version*. Depending on the deployment, the endpoint is deducted in terms of IP address or hostname, and port. The dependencies between microservices are modeled as follows. The caller is defined by the microservice type, its version, and the calling REST operation while the callee is defined by the microservice type and the called REST operation.

3. GENERATION

Based on an instance of the metamodel, the code artifacts for each microservice are generated using the template features of Xtend[1]. The resulting artifacts are:

- Java source code using Spring Boot[2] annotations for creating a web service environment and including instrumentation for the Kieker monitoring framework [9]
- An Apache Maven[3] *pom.xml* file for building the synthetic microservice application.
- A *Dockerfile* that allows the creation of a Docker[4] container for the microservice application.
- YAML files required for deploying the microservice container on a Kubernetes[5] cluster.

After the generation process has finished, the Maven *pom.xml* can be used to pull in all required dependencies and compile an executable Java JAR file. Once the executable is present, the Docker image for the microservice can be created and it can be deployed to a Kubernetes cluster.

4. SUPPORTING SERVICES

The approach comprises supporting services as depicted in Figure 1. User requests are generated by a microservice that runs Apache JMeter.[6] Every generated microservice is set up to send its monitoring data to a microservice running a JMS server which collects the monitoring data of all microservice instances in the SUT. The *Monitoring Server* collects the monitoring data (including data about distributed execution traces) received from the JMS server. By default, each microservice of the SUT is configured to request information about problems (e.g., delays or other performance anti-patterns [5]) to be injected for every REST operation from the *Registry* service. Based on the microservice type, its version, and its unique ID, the *Registry* microservice returns the corresponding problem. These problems get controlled by a component named *Injector*. It uses the application programming interface (API) of the *Registry* microservice to change the behavior by modifying the problems that are returned to the SUT microservice instances.

[1]https://www.eclipse.org/xtend/
[2]http://projects.spring.io/spring-boot/
[3]https://maven.apache.org/
[4]https://www.docker.com/
[5]https://kubernetes.io/
[6]https://jmeter.apache.org/

5. CONCLUSION

With the proposed model-based generation tool set for performance and resilience benchmarking it is possible to use instances of a metamodel to define the microservice architecture that will be generated. The generated synthetic microservices can then be monitored while problems get injected into the microservice environment. The proposed setup was developed and used for evaluation purposes in our work on anomaly detection in microservice architectures [3]. Even though it is not a full-fledged tool set yet, it can provide building blocks for further extension for different aspects in terms of performance and resilience benchmarking. In the future, the range of possible injections could be extended by antipatterns as proposed in [5] and resource demand injections. Furthermore, support for more monitoring tools, automatic extraction of model instances from monitoring data, and generation of user request scripts could be extended.

The presented generative platform is publicly available at https://github.com/orcas-elite/arch-gen.

6. REFERENCES

[1] L. J. Bass, I. M. Weber, and L. Zhu. *DevOps - A Software Architect's Perspective*. SEI series in software engineering. Addison-Wesley, 2015.

[2] M. Brambilla, J. Cabot, and M. Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.

[3] T. F. Düllmann. Performance anomaly detection in microservice architectures under continuous change. Master's thesis, University of Stuttgart, 2017.

[4] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010.

[5] P. Keck, A. van Hoorn, D. Okanovic, T. Pitakrat, and T. F. Düllmann. Antipattern-based problem injection for assessing performance and reliability evaluation techniques. In *Proc. ISSRE Workshops 2016*, pages 64–70, 2016.

[6] R. Natella, D. Cotroneo, and H. S. Madeira. Assessing dependability with software fault injection: A survey. *ACM Comput. Surv.*, 48(3):44:1–44:55, Feb. 2016.

[7] S. Newman. *Building Microservices*. O'Reilly Media, Inc., 2015.

[8] M. Nygard. *Release it!: design and deploy production-ready software*. Pragmatic Bookshelf, 2007.

[9] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: a framework for application performance monitoring and dynamic software analysis. In *Proc. ICPE'12*, pages 247–248, 2012.