Performance Engineering for Microservices: Research Challenges and Directions

Robert Heinrich,[1] André van Hoorn,[2] Holger Knoche,[3] Fei Li,[4]
Lucy Ellen Lwakatare,[5] Claus Pahl,[6] Stefan Schulte,[7] Johannes Wettinger[2]

[1] Karlsruhe Institute of Technology, Germany
[2] University of Stuttgart, Germany
[3] Kiel University, Germany
[4] Siemens AG, Austria
[5] University of Oulu, Finland
[6] Free University of Bozen-Bolzano, Italy
[7] TU Wien, Austria

ABSTRACT

Microservices complement approaches like DevOps and continuous delivery in terms of software architecture. Along with this architectural style, several important deployment technologies, such as container-based virtualization and container orchestration solutions, have emerged. These technologies allow to efficiently exploit cloud platforms, providing a high degree of scalability, availability, and portability for microservices.

Despite the obvious importance of a sufficient level of performance, there is still a lack of performance engineering approaches explicitly taking into account the particularities of microservices. In this paper, we argue why new solutions to performance engineering for microservices are needed. Furthermore, we identify open issues and outline possible research directions with regard to performance-aware testing, monitoring, and modeling of microservices.

1. INTRODUCTION

Microservices [9] have had a huge impact on the software industry in recent years [1]. Scalability, flexibility, and portability are very often named as particular benefits of this architectural style. Furthermore, microservices are considered to be an enabler for emerging software development practices, specifically DevOps and continuous deployment (CD) [7], which aim for more frequent and rapid deployments from development into production. Despite the fact that performance is an inherent necessity to achieve scalability and elasticity, performance engineering for microservices has so far only achieved very little attention by both the microservices and the performance engineering research communities. An extensive body of approaches and best practices for performance engineering—involving measurement-based and model-based techniques—for traditional software engineering development environments and architectural styles is available. However, already their application in DevOps imposes both challenges and opportunities [3]. Therefore, we discuss and identify research challenges and opportunities for performance engineering for microservices in this paper.

To start with, we discuss in Section 2 specific characteristics of microservices and explain why it is not possible to simply transfer existing concepts from other fields to this area. Afterwards, we will discuss particular research challenges for performance engineering for microservices (Section 3) and conclude this paper.

ICPE '17 Companion, April 22-26, 2017, L'Aquila, Italy
© 2017 ACM. ISBN 978-1-4503-4899-7/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3053600.3053653

2. BACKGROUND

Microservices can be seen as an advancement of service-oriented architectures (SOAs). This is underlined by the fact that Netflix, a company relying heavily on microservices, used to refer to their architecture as *fine-grained SOA*. SOAs have gained extensive attention by the research community in the last decade. This includes performance engineering solutions, e.g., [2]. However, there are some important characteristics of microservices which make performance engineering for microservices more complex and/or do not allow the direct application of performance engineering approaches originally modeled for SOAs [11].

The fundamental difference between SOAs and microservices is that they pursue very different goals. The primary goal of SOAs is the integration of different software assets, possibly from different organizations, in order to orchestrate business processes. Thus, SOAs completely abstract from the internal structure of the underlying assets. It is irrelevant if services are provided by several well-structured applications or a huge, tangled legacy monolith. Microservices, on the other hand, aim at improving the development, delivery, and deployment of the software itself. Therefore, microservices are very much concerned with the internal structure of the software. Every service is required to be an autonomous, independently deployable unit of manageable size which interacts with other services only via technology-agnostic interfaces such as RESTful Web APIs. In particular, code sharing between services and shared databases is

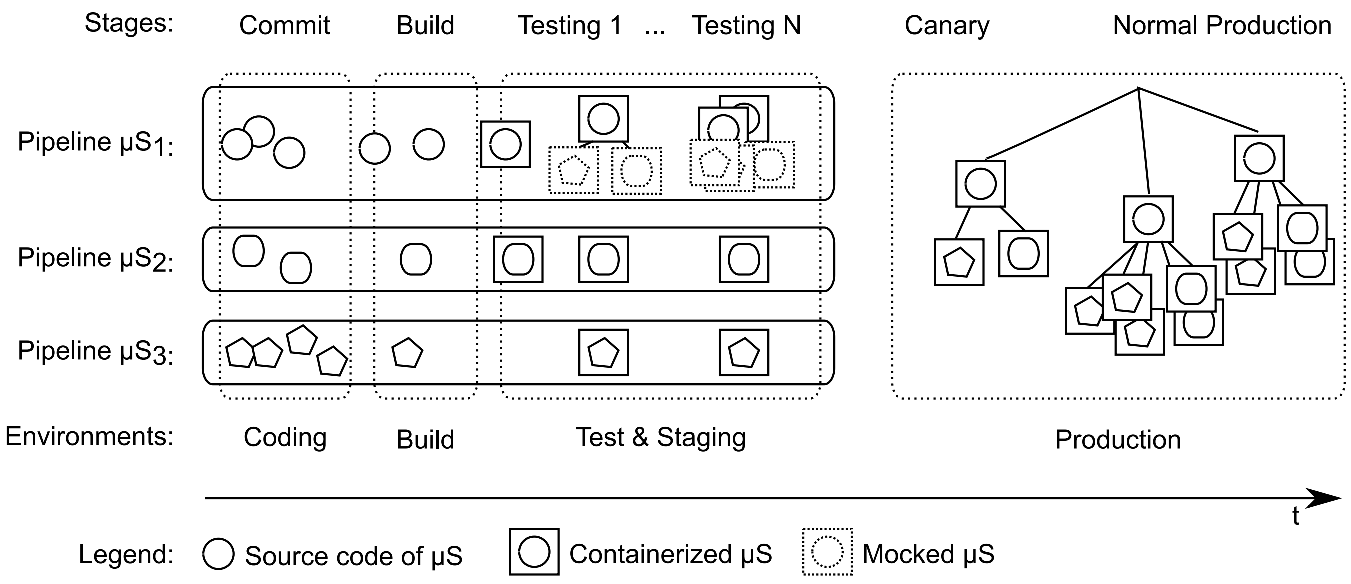

Figure 1: Three Microservices ($\mu S_1 \ldots \mu S_3$) in a CD setup

discouraged, as both may lead to tight coupling and a loss of the service's autonomy.

Microservices are often considered to be an enabler for CD and DevOps. The technical autonomy allows to create independent deployment pipelines for each service, as depicted in Figure 1. Thus, changes can be rapidly delivered to production, since only the affected service needs to be built and tested, and several pipelines can run in parallel. However, this rapid delivery poses several challenges for performance testing, which are discussed in Section 3.1.

Due to their highly distributed nature, microservices are more difficult to operate than monoliths. Therefore, sophisticated (container-based) virtualization and infrastructure technologies such as Docker[1] and Kubernetes[2] have emerged, which facilitate operations by providing functionality such as rolling updates, automated scaling, and rebalancing in case of node failure. As a consequence, microservice-based deployments are much more dynamic and volatile than traditional applications, creating challenges for both monitoring and performance modeling. These challenges are discussed in Section 3.2 and 3.3, respectively.

3. RESEARCH CHALLENGES

This section addresses research challenges for performance engineering for microservices, focusing on the aspects of testing, monitoring, and modeling.

3.1 Performance Testing

Compared to traditional software architectures, DevOps and microservices lead to a higher frequency and pace of releases. Microservices are often deployed several times a day (or even an hour), which influences the way performance testing can be conducted.

In general, early-stage testing for microservices [9] that happens before deployment (i.e., unit tests, etc.) follows common software engineering practices. However, when considering further test stages (i.e., integration and system tests that involve other microservices), there are microservice-specific challenges which need to be taken into account. For example, extensive system tests are not feasible due to the higher frequency of releases. Instead, service quality assurance is often compensated or even replaced by fine-grained monitoring techniques in production environments exposed to the users of the application. Contrary to conducting extensive tests before release, failures are monitored and quickly corrected by pushing new releases to production environments. Therefore, microservices are deployed before full-scale integration testing including all other relevant microservices in the application is performed. Consequently, potential problems and errors are identified in the production environment, e.g., by using canary deployment. This approach fits the idea of each microservice being an independently (re-)deployable unit [9], which is an essential technical foundation for microservice architectures to foster loose coupling between components. Thus, an initial challenge is to systematically investigate how to change and adapt established performance testing strategies, so that extensive integration and system tests are replaced and compensated by respective activities in the production environment.

By the application of (containerized) microservices, performance testing as well as performance regression testing get simplified in the first place, i.e., the performance of each microservice (technically often deployed as a single container) can be measured and monitored in isolation. However, since the associated performance tests can take several hours even for a single microservice, the feasibility of extensive performance testing is limited. Once a microservice is used in production, the monitored performance data (e.g., using established application performance monitoring approaches as discussed in Section 3.2) can in turn be utilized to devise and improve performance regression testing. This way, performance testing is made easier and faster by effectively taking feedback from operations. Another key challenge in this context is to better align performance testing and performance regression testing with CD practices,

[1]https://www.docker.com/
[2]https://kubernetes.io/

especially focusing on speeding up the involved test stages to make the pipeline as fast as possible while providing a sufficient level of confidence.

As stated, microservice architectures are typically based on CD practices [4, 7]. The pace of these practices, however, would be hard to achieve when simply following extensive integration and performance testing approaches. Thus, the scope of tests must be carefully chosen: Which tests are run for each commit and which tests are run for a consolidated set of commits? Sophisticated test selection and classification criteria are required for this decision. These must be combined with dynamic mechanisms to decide which tests to run in which situations. The selection criteria cannot be static, but must be dynamically and (semi-)automatically adapted to continuously guarantee a reasonable testing strategy. Input from monitoring data, operational failures, and the actual application status can be used to decide which tests should be performed. Consequently, a further research challenge in this context is the dynamic and (semi-)automated selection and separation of tests, in order to decide which tests to run for which commit.

To sum up, three key challenges appear in terms of testing applications that are following a microservice architecture:

- Replacing and compensating extensive integration and system testing by fine-grained monitoring of production environments
- Aligning performance testing and performance regression testing with CD practices, i.e., speeding up the corresponding test stages
- Dynamic and (semi-)automated performance test selection

3.2 Monitoring

Monitoring in microservice architectures can be done using similar techniques as in the state-of-the-art [6]. In traditional architectures, application performance management (APM) tools that support the collection of various measures, are used to collect performance-relevant runtime data of the entire system stack starting from low-level measures such as CPU and memory usage, via measures on the operating system, virtualization layer, middleware, and application (especially detailed execution traces). Agents that collect data from heterogeneous systems implemented with a variety of technologies (Java, .NET, JMS, etc.) are provided by the APM tools. Based on the history of data, APM tools compute a model of the normal behavior (e.g., baselines) that are then used to detect anomalies, such as exceptionally high response times or resource usage. Similarly, performance-relevant data in microservice architectures can be collected from the microservice inside a container, from the container, and from interrelated microservices. However, this poses several challenges.

First, since microservices are commonly used to decompose a single application, end-to-end traces can be collected for individual client requests, as opposed to SOAs where monitoring usually stops at calls to other services because they are owned by other parties. A technical instrumentation challenge is imposed by the microservice characteristic of polyglot technology stacks, particularly involving the use of emerging programming paradigms and languages (e.g., Scala). We expect the APM tool vendors to provide suitable solutions in the near future; leading APM tools such as Instana[3] particularly focus on monitoring microservice architectures. In addition to the technical instrumentation challenge, these tools focus on dedicated interactive graphical views displaying the large-scale topologies of these systems.

Second, additional measures that are important for the ability to monitor specific architectural patterns at runtime in microservice architectures are needed. Examples of these include the state of resilience mechanisms [10], such as circuit breakers (open, half-closed, etc.). Usually, these patterns can be used by including third-party libraries such as the Netflix OSS stack and respective monitoring interfaces are already included. Another reason for additional metrics is the use of auto-scaling. For instance, data on the level of containers (e.g., CPU or memory utilization, and startup times of Docker containers) is needed to decide on scaling up or down, as used by Kubernetes, for instance. Moreover, online testing techniques such as A/B testing require dedicated monitoring.

Third, in microservice architectures, it becomes difficult to determine a normal behavior. This is due to the frequent changes (updates of microservices, scaling actions, virtualization) where no "steady-state" exists. Existing techniques for performance anomaly detection may therefore raise many false alarms. A promising solution is to explicitly incorporate logs of change events into the decision whether a deviation in runtime behavior is classified as an anomaly or not [5]. We see the main field for researchers in the topic of developing accurate and precise anomaly detection techniques—and even addressing the consequent next step, namely the diagnosis and resolution of the anomalies.

In summary, three key areas with respect to monitoring in microservice architectures have been identified:

- Instrumentation for distributed monitoring in microservices that are characterized by a polyglot technology stack
- Additional measures to monitor microservices
- Precise anomaly detection techniques in microservice architectures

3.3 Performance Modeling

Performance modeling has gained considerable attraction in the software performance engineering community in the past two decades, most prominently for component-based software systems [8]. Initially, performance modeling and prediction had been proposed for early design-time performance evaluation. In the last years, the focus of modeling moved to runtime aspects—e.g., extracting models from operational data and using them for online performance evaluation and resource management. However, so far no approaches for performance modeling of microservices exist. In the remainder of this section, we present a number of observations to argue why existing approaches cannot simply be applied to microservices.

Our first observation is that there has been a shift in use cases for performance modeling, especially design-time modeling. Being a so-called cloud-native architecture, microservices almost perfectly exploit the elasticity provided by cloud platforms and container-based virtualization. Thus,

[3]https://www.instana.com/

for instance, a traditional use case for design-time performance modeling, namely capacity planning, has become much less important in such settings. Instead of detailed upfront planning, a thorough cost controlling is employed. On the other hand, we see applications for design-time performance modeling emerge in new areas, such as reliability and resilience engineering and the design of runtime adaptation strategies.

These new applications require some fundamental changes to the models themselves, which is our second observation. In particular, new abstractions are needed to adequately capture the recent advances in deployment technology. Cluster management tools like Kubernetes automatically manage the deployment of services on a pool of machines, and continuously adapt the deployment structure at runtime. Amazon Lambda[4] even goes one step further and completely hides machines from the developer. In such settings, traditional performance models based on the notion of distinct (virtual) machines are inadequate. New modeling strategies must be found to adequately represent such structures and the behavior of the automated infrastructure components. Another challenge is the size of the models, as large microservice installations may consist of tens of thousands of service instances. We expect that it will be necessary to reduce the level of detail to be able to cope with such models in a timely fashion.

Our third observation is that the creation of such models becomes more difficult. Unlike traditional models, it is not sufficient to discover the relevant structural entities, such as clusters or services, and extract static parameters from runtime data. The behavior of the dynamic components, in particular the aforementioned management tools, must also be learned. We assume that techniques from machine learning can be helpful in this task. Due to the highly dynamic nature of those deployments, it is also necessary to keep the models up-to-date automatically.

Thus, four key challenges emerge for performance modeling for microservices:

- Adopting performance modeling to shifted use cases
- Finding appropriate modeling abstractions
- Automated extraction of performance models
- Learning of infrastructure behavior and integration into performance models

4. CONCLUSIONS

Microservices are an emerging architectural style enabling the efficient use of cloud technologies and DevOps practices. So far, performance engineering for microservices has not gained attraction in the relevant communities. In this paper, we argued that the existing performance engineering techniques—focusing on testing, monitoring, and modeling—cannot simply be re-used. Particular challenges include *i.)* strategies for efficient performance regression testing, *ii.)* performance monitoring under continuous software change, *iii.)* appropriate performance modeling concepts for shifted use cases.

For our community, these challenges impose a number of interesting research questions and directions — involving both testing, monitoring, and modeling in isolation along the pipeline but especially settings in which they complement each other. Examples include:

- The use of performance models to prioritize tests cases, including the decision whether and when (pipeline vs. production) tests are executed
- The use of monitoring data to create and refine performance tests, including the creation of representative usage profiles and performance-aware service mockups
- The use of tests and resulting monitoring data to automatically create performance models, including the combination of data from different environments
- The use of models to guide the diagnosis of performance problems
- Learning and assessing deployment strategies based on monitoring data

[4]https://aws.amazon.com/lambda/

5. ACKNOWLEDGMENTS

This paper is a result of the GI-Dagstuhl Seminar 16394 "Software Performance Engineering in the DevOps World" (http://www.dagstuhl.de/16394).

6. REFERENCES

[1] L. Bass, I. Weber, and L. Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley, 2015.

[2] P. C. Brebner. Performance modeling for service oriented architectures. In *Comp. of the 30th Int. Conf. on Software Engineering (ICSE '08)*, 2008.

[3] A. Brunnert et al. Performance-oriented DevOps: A Research Agenda. Technical Report SPEC-RG-2015-01, SPEC Research Group — DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC), August 2015.

[4] L. Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2), 2015.

[5] M. Farshchi, J. Schneider, I. Weber, and J. C. Grundy. Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis. In *Proc. 26th IEEE Int. Symp. on Software Reliability Engineering (ISSRE 2015)*, pages 24–34, 2015.

[6] C. Heger, A. van Hoorn, D. Okanović, and M. Mann. Application performance management: State of the art and challenges for the future. In *Proc. 8th ACM/SPEC Int. Conf. on Performance Engineering (ICPE '17)*. ACM, 2017.

[7] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.

[8] H. Koziolek. Performance evaluation of component-based software systems: A survey. *Elsevier Performance Evaluation*, 67(8):634–658, 2010.

[9] S. Newman. *Building Microservices*. O'Reilly, 2015.

[10] M. Nygard. *Release It!: Design and Deploy Production-Ready Software (Pragmatic Programmers)*. Pragmatic Bookshelf, 2007.

[11] M. Richards. *Microservices vs. Service-Oriented Architecture*. O'Reilly Report, 2016.