Automatic performance monitoring and regression testing during the transition from monolith to microservices

Andrea Janes
Faculty of Computer Science
Free University of Bozen-Bolzano
Bolzano, Italy
ajanes@unibz.it

Barbara Russo
Faculty of Computer Science
Free University of Bozen-Bolzano
Bolzano, Italy
brusso@unibz.it

***Abstract*—The transition from monolith to microservices poses several challenges, like how to redistribute the features of system over different microservices. During the transition, developers may also redesign or rethink system services significantly, which can have a strong impact on various quality aspects of the resulting system. Thus, the new system may be more or less performing depending on the ability of the developers to design microservices and the capability of the microservice architecture to represent the system. Overall, a transition to microservices may or may not end up with the same or a better performing system. One way to control the migration to microservices is to continuously monitor a system by continuously collecting performance data and feeding the resulting data analysis back in the transition process. In DevOps, such continuous feedback can be exploited to re-tune the development and deployment of system's builds. In this paper, we present PPTAM+, a tool to continuously assess the degradation of a system during a transition to microservices. In an in-production system, the tool can continuously monitor each microservice and provide indications of lost performance and overall degradation. The system is designed to be integrated in a DevOps process. The tool automates the whole process from collecting data for building the reference operational profile to streamline performance data and automatically adapt and regress performance tests on each build based the analysis' feedback obtained from tests of the previous build.**

I. Introduction

The term "microservice architecture" describes an architectural style that structures applications as a set of independently deployable services [1]. Promoters of microservice architectures claim that such systems are easier to maintain and to scale.

While there is no precise definition of what a microservice architecture in detail implies [1], there are characteristics that are shared by many microservice architectures [1]. For instance, data is organized in a decentralized way (having each service manage its own data makes it independently deployable) and teams that build systems with microservices extensively use infrastructure automation techniques (like continuous integration or continuous delivery).

Developing a system using a microservice architecture is not straightforward: some aspects are new (e.g., one has to decide on a communication infrastructure) and other aspects that are valid when developing a monolith have to be reconsidered (e.g., how to keep communication between services minimal, as communication is costly and might impede scalability). Furthermore, a microservice system is rarely built from scratch and is often migrated from an existing monolith architecture. Thus, questions arise on how to organize such transition or how to split services of a monolith into a set of microservices.

The migration toward a microservice architecture requires the team to acquire new knowledge and to learn how to apply it [2], [3]. New software design patterns for microservice architectures also occur, like the Command Query Responsibility Segregation (CQRS) to organize data or the API Gateway pattern to organize how clients can access individual services (see e.g. [4]). In particular, performance can be an issue for such systems [3], [5]–[7]: "When decomposing systems into smaller microservices, we increase the number of calls that will be made across network boundaries. Where previously an operation might have involved one database call, it may now involve three or four calls across network boundaries to other services, with a matching number of database calls. All of this can decrease the speed at which our systems operate [5]."

In this paper, we propose an approach to observe and detect performance degradation during the transition towards a microservice application. In the context of testing, our approach aims at avoiding performance regression during the transition from an old, monolithic, to a new microservice architecture. In the context of Lean development, this approach can be seen as an implementation of Jidoka [8] (in English called Autonomation), which describes a mechanism, for which a machine is able to detect an anomaly by itself and then stop to allow workers to investigate the reasons of the anomaly and restart the production. We see the contribution of our tool in the same way: if the new architecture performs under a given threshold, it might be time to stop developing and to rethink the architecture or to rethink the used patterns to guarantee that the new system—while having all advantages of a microservice architecture—does not fall short in terms of performance.

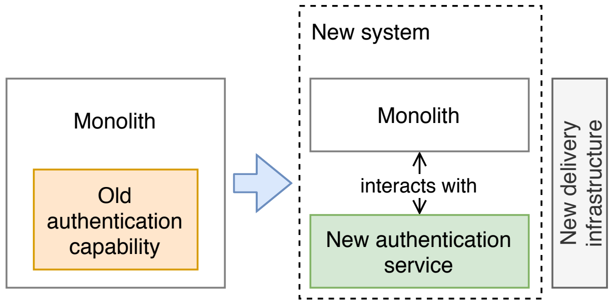

Monolith
Old authentication capability
New system
Monolith
interacts with
New authentication service
New delivery infrastructure

Fig. 1: Decoupling a capability from a monolith (based on [9])

The remaining of this paper is organized as follows. In Section II, challenges of a transition from monoliths to microservice architectures are presented; in Section III, our approach to continuously monitor system performance during a transition is described. In Section IV, our tool PPTAM+ is presented looking at its components. In Section V, we reflect on the challenges of the migration to microservices.

II. From monoliths to microservices

Changing architectural style is a costly endeavour: software development teams that are involved in such transitions not only need to perform the necessary modifications to the source code, they also face the risk to introduce new defects since existing, working source code has to be adapted or in part rewritten so that it fits into the new architecture. One strategy to reduce the risk is not to make the transition using a "big-bang approach" (i.e., designing a microservice architecture and rewriting/transfer the source code of the monolith into the new architecture) but to use an approach as described e.g., by [9] and depicted in Figure 1: we identify one capability in the monolith that we want to transform into a microservice, we decouple it from the monolith into an external service and maintain the old monolith with all its existing functionality. This approach also allows to setup and improve the new delivery infrastructure as more and more parts of the monolith become microservices, i.e., an infrastructure that helps to build, test, and deploy the new system as a collection of microservices. Typical technologies for such infrastructures are Gitlab[1], Jenkins[2], or Travis CI[3].

The solution we propose in this paper is part of the delivery infrastructure: its goal is to ensure that the performance of the new system (independently whether it is implemented using the "big-bang approach" or using the gradual approach described before) does not degrade. In the next section, we illustrate the architecture of the proposed approach.

III. Approach

The approach we propose is depicted in the concept map of Figure 2 and consists of the following steps:

1) We log all user interactions with the user interface of the monolith. The result of this step are the response times for different usage intensities and the number of accesses to each service. We log the current user, the UI elements s/he interacts with and the stack trace that this interactions produces.
2) We use an extended version of PPTAM [10], PPTAM+ (see Section IV) to test various configurations of the new system and collect the resulting response times.
3) We compare the response times of the original monolith and visualize the comparison on the visualization tool, with which the stakeholder can decide on how to proceed with development, accept or reject new microservices on the basis as their performance behavior.

Step 1 has the goal to create the baseline to compare the performance of the new architecture. Of course, one way to accomplish this would be to modify every call to some user interface component in the monolith so that it logs the interaction. This would be too costly, therefore practitioners developed various technologies to log application-wide user interface interactions without the need to modify its source code [11]:

1) If the monolith is written in Microsoft .NET[4], using the Microsoft Windows specific graphical subsystem for rendering user interfaces Windows Presentation Foundation (WPF), it is possible to define "behaviours", i.e., reusable components that can be attached to user interface elements of a given type. Such a behaviour can then log the user interactions with the UI such as a button click, a window that is opened or closed, etc.
2) Using Aspect Oriented Programming [12] it is possible to create so called "aspects" that can be inserted in the source code using predefined rules (this step is called "aspect weaving"). In this case we can define an aspects that logs a user interaction. Aspect Oriented Programming is available in many programming languages.
3) In web programming, it is possible to either write a JavaScript component on the client side or to log calls on the server side [13] to understand which pages and actions on pages are executed.
4) Customizing and integrating off-the-shelf Open Source solutions in the Application Performance Monitoring family as described in Section IV like openTracing. The customization of such tools might be done in combination of some of the solutions previously mentioned (e.g., Aspect Oriented Programming to identify the stack traces of interest).

One approach for traditional applications to monitor software usage without modifying the source code is to interact with the accessibility API included in every modern operating system, e.g., Windows, Mac OS, and Linux provide an accessibility API (e.g., [14], [15]), which allow a program to detect over which button the user positions the mouse cursor, which window is opened, which tab is selected, and so on.

[1]https://about.gitlab.com
[2]https://jenkins.io
[3]https://travis-ci.org

[4]https://www.microsoft.com/net

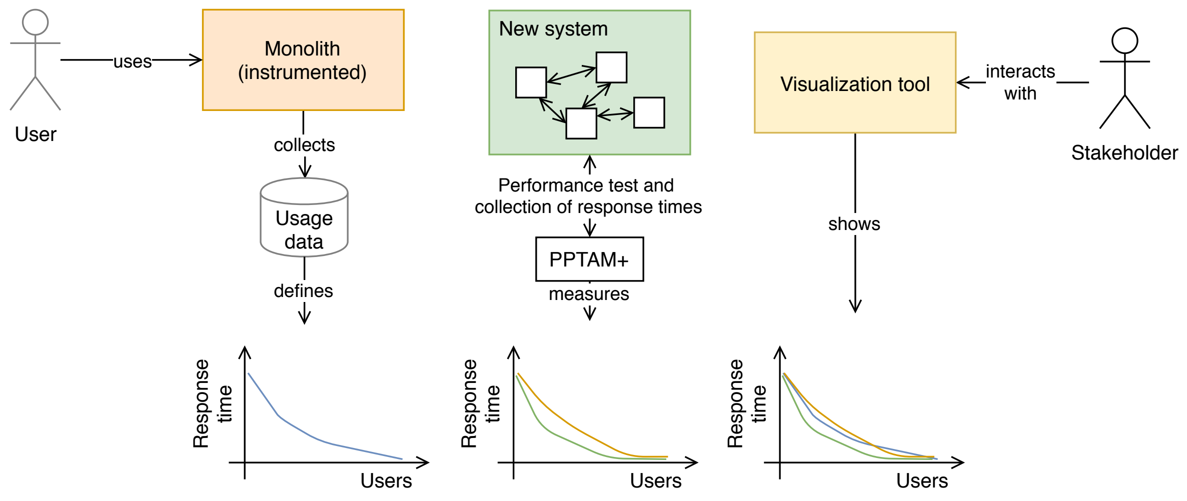


Fig. 2: Approach overview

The purpose of this API is to support the creation of software that helps blind users or users with bad eyesight to operate a computer. In such case, software that uses the accessibility API reads the currently focused element aloud using text-to-speech technologies. In our case, we use this API to infer—without any modification to the original source code—which element the user is currently using.

Table I illustrates the structure of the data collected by the usage data collector.

TABLE I: Output of usage data collection

ID	Timestamp	Log Entry	Response time
1	2019-05-01 09:00:00	(Stack trace of UI event)	00:00:30
2	2019-05-01 09:00:50	(Stack trace of UI event)	00:00:50
3	2019-05-01 09:01:10	(Stack trace of UI event)	00:00:20

The stack traces collected at this step are used to measure the usage intensity of a given component, set of components, or the entire application. They can then help to establish a performance benchmark for the application or sub-modules of it. Depending on the adopted strategy to split the monolith into microservices or to introduce microservices in the monolith, various usage data can be filtered out, based on the stack trace. For example, if Table I contains usage data over several months, which report the usage of control elements embedded in forms that are contained either in the package "it.companyName.productName.inventory" or in the package "it.companyName.productName.invoicing", and if later two microservices will be created that contain the functionality of inventory management and invoicing, it will be possible to use the usage data only for one specific component to test it.

IV. PPTAM+

In [10], we implemented a tool for Production and Performance Testing Based Application Monitoring (PPTAM). The tool monitors the frequencies of requests of concurrent users (workload) to micro-services systems to test system response under different workloads and configurations. Performance is then estimated with the frequency of microservices that do not fail under a given workload. To instrument PPTAM for a given in-production system, users and operations must be profiled.

In this paper, we propose an extension of such tool, namely **PPTAM+**, which integrates an Application Performance Monitoring (APM) tool that collects performance data and stack traces and provides to stakeholders interactive visualizations of the resulting performance analyses. In this work, the PPTAM tool is also enhanced to enable it detecting performance degradation under a transition from monoliths to microservice systems developed and deployed with DevOps.

A. *Application Performance Monitoring*

An Application Performance Monitoring (APM) tool aims at monitoring the performance of an in-production system (e.g., a monolith). An APM tool is typically built by integrating few components each performing a major activity of performance monitoring. A typical output of such tools are log accesses per service, stack traces and usage time. To get such data, various tools needs to be integrated or configured if not developed from scratch. Understanding dependencies among such components in the rich realm of software choices can be an hard task. The openAPM[5] initiative is an online tools' configurator that helps select open source applications to build an APM tool. In our tool, we want to collect logs of usage data and stack traces.

Figure 3 illustrates a default configuration proposed by openAPM for the APM analytic activity implemented with the Elastic[6] and Kibana[7] stack (ELK). Through the configuration, an Elastic APM agent is installed on the application to monitor and collect performance data, which is then sent to the Elastic

[5]https://openapm.io
[6]https://www.elastic.co
[7]https://www.elastic.co/products/kibana

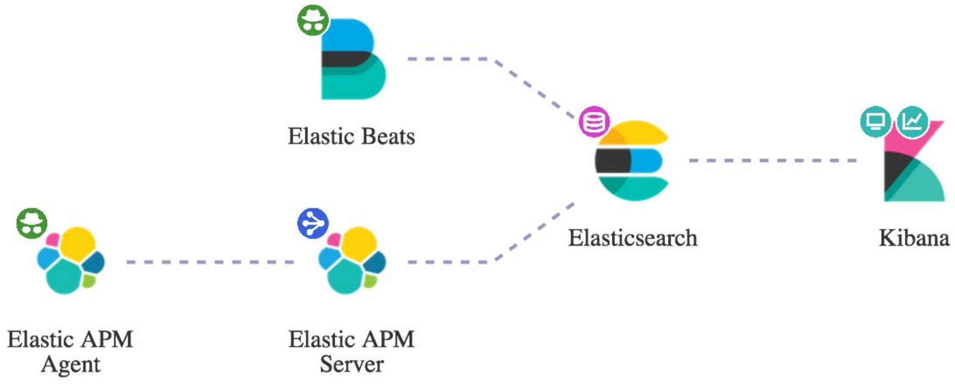


Fig. 3: APM solution with Elasticsearch and Kibana stack (ELK)

APM server. In a distributed environment one can use the agent Elastic Beats that sends performance data from applications running in different machines. Via the APM server or directly, the two agents send performance data to Elastisearch. Elasticsearch is a search engine based on Apache Lucene[8] that can search and aggregate performance data (text and numbers). Data are finally presented with Kibana, a web user interface that visualizes complex aggregations, graphs, and charts (e.g., time series). For analysis of time series data (e.g., response time over nightly builds of a System Under Test (SUT)), data can be stored in InfluxDB[9]. At the same time, Kibana can be substituted with Grafana[10] that provides more support for time series analysis. Alternatively, once performance data is collected in InfluxDB, scripts in R[11] (or Python[12]) can be executed online with Jupiter[13] notebooks to analyze and present the results of the analysis. Such solution may be better implemented for demonstrative use.

The new x-pack[14] of ELK further enhances the above APM tool with machine learning capabilities. Thus, depending on the SUT and the data of interest that can be collected from it, different solutions can be integrated to build the most appropriate APM tool for analytics. To collect stack traces, openTracing[15] can be used. As stack traces have time stamps, traces can be stored in InfluxDB together with other performance data collected with ELK stack. Traces can used to reconstruct the path that transactions take through a microservices architecture. As we mentioned in Section III, the use of some of these tools might also require the instrumentation of the System Under Test (e.g., using Aspect Oriented Programming) or exploiting the API of the underlying operating system.

[8]https://lucene.apache.org
[9]https://www.influxdata.com
[10]https://grafana.com
[11]https://www.r-project.org
[12]https://www.python.org
[13]https://jupyter.org
[14]https://www.elastic.co/what-is/open-x-pack
[15]https://opentracing.io

B. Test automation

PPTAM uses the open-source BenchFlow Declarative domain Specific Language (DSL) and framework [16] to automate the deployment of experiments on performance tests for a given workload and system configurations. Benchflow provides DSL templates for automating performance tests. The DSL is used to specify the intent of performance tests and control their entire end-to-end execution process in a declarative manner, and it is particularly tailored to container-packaged microservice systems. The DSL provides the specification of load functions, workloads, simulated users and profiles, test data, test-bed management and analysis of performance data. Benchflow also adds a goal-oriented and declarative specification of performance intents. The automation test is coded along three phases: exploration, execution and analysis. The exploration phase defines way a performance test is executed in order to reach its performance goal. After the test definition has been statically verified for correctness, based on the goal, one or more experiment definitions are generated together with the corresponding SUT deployment descriptor, and a given number of trials are scheduled for execution. In this phase, also the termination criteria and quality gates are evaluated to decide how to continue the exploration. Deploying and running a performance test is done in the execution phase using containers (e.g., using Docker[16]. A basic statistical analysis of the distribution of the performance measure (e.g., response time) is defined in the analysis phase. Finally, the Faban[17] load driver framework is used to run the tests and collect the performance measures.

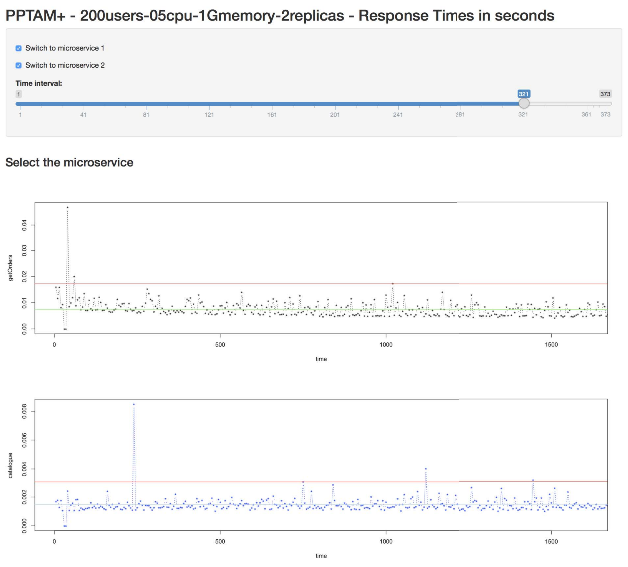
C. Visualization of performance data

To exchange data between the components of the PPTAM+ tool, several scripts in R have been used. To have a sense of their execution, the scripts have been implemented in Jupiter notebooks. Although notebooks are very useful for demonstration purposes, they are not intended to be used for user's visualization. On the other hand, existing visualization tools

[16]https://www.docker.com
[17]http://faban.org

like Kibana and Grafana that can be integrated in the APM solution for analytics are not readily extensible and tailor-made to follow evolution of the analysis during a migration or the specific requests of the stakeholders.

Thus, PPTAM has been extended by exploiting the package RShiny of RStudio[18] that provides a web framework for building web applications using R. Given the wide-choice of R packages, PPTAM+ can be equipped with very advanced features of performance analysis and graphical visualizations as well be easily tailored and adapted with the migration process. For example, a simple version of the GUI built with RShiny can already show interactive graphs that sample the performance of different versions of the systems (Figure 4). Stakeholders can select the period of observation of the system or the type of microservice to monitor from the web page and observe the performance degradation of the percentage of services that do not achieve the reference benchmark.


PPTAM+ - 200users-05cpu-1Gmemory-2replicas - Response Times in seconds

[✓] Switch to microservice 1

[✓] Switch to microservice 2

Time interval: 1 – 321 (of 373)

Select the microservice

Fig. 4: Simple visualization in web browser of microservices' performance with RShiny. Red lines are benchmark thresholds.

D. Integration with DevOps technologies

To integrate our approach in DevOps, we apply the load testing process proposed by Schulz *et al.* [17]. The process starts with monitoring data of an in-production SUT consisting of request logs, traces, and response times of the service interfaces. As described in Section III this step can be performed with an Open Source APM solution. In this step, data are further enriched by contextual information (e.g., SUT availability). A second step is workload analysis, during which the request logs are transformed into workload models and stored in a workload repository along with the contextual information. As we mentioned this can be InfluxDB as by continuously streaming performance data and workload analysis we can build up time series of performance over workloads. A context analysis also serves to build the goals and testing specifications needed to configure Benchflow templates whereas the analysis of stack traces is used to identify the microservices to be tested or deployed after testing.

For example, if a back-end service is to be tested, one can compute the workload on the back-end service resulting from the selected workload models and run the load directly to the back-end without deploying any front-end services. Benchflow testing templates may need to manually changed with the evolution of the transition to microservices. This changes can be retrieved from a control versioning system or through changes of the API specifications of the SUT.

The selection of the microservices to test or deploy after test can also be driven by computing the performance behavior of the individual services. To this aim, we use the approach we proposed in [10], [18] applied to the transition to microservices: the frequency of microservices that perform better than a given threshold determine the performance behaviour of the SUT in the various stages of the transition. Figure 5 illustrates the comparison of different SUT behaviour against a reference performance behaviour (outer polygon), one of the curves in the Figure represents the monolith behaviour and the others represent versions of microservice SUT developed during the transition. The goal of the transition is to approach the ideal behaviour the most (polygon over green area).

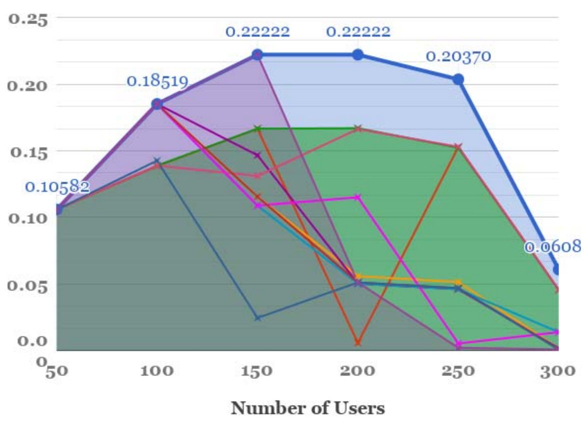


Fig. 5: Comparing microservices' performance

The microservice SUT that best performs is deployed together with the tested services using tool coming, e.g., from the Jenkins[19] family. The knowledge about the services deployment comes from a composition model, e.g., a docker-compose file.

In this phase, also the degree of automation can be defined: in a completely automated DevOps approach, it is necessary to define thresholds after which, PPTAM+ would warn the team that the performance of a given microservice is degrading too much. In a setting, where the collected information is used in

[18]https://www.rstudio.com

[19]https://jenkins.io

weekly or daily meetings, the output of the visualization tools can be used by the team to decide how to proceed.

V. Conclusions and future work

In DevOps, continuously monitoring the performance of a system can help understanding whether a system migrating to microservices is experiencing any degradation. The feedback provided by the monitoring can be used to re-design or efficiently allocating system's features over microservices. In this work, we propose a tool to continuously streaming down performance data, analyzing them and feeding back to the migration process the results of the analysis. The analysis provide actionable metrics that indicate performance degradation against a reference benchmark. The tool extends a previous one, built to evaluate the behaviour of a system running with microservices. The proposed extension includes an APM component for analytics and visualization components built to illustrate to the stakeholders of a DevOps process the performance degradation of the system. To feedback the information back to the migration process, the tool is integrated in the load testing process proposed by Schulz *et al.* [17] for which the templates for continuous performance testing are automatically updated with the feedback of the monitoring (e.g., selection of the microservice to test).

A. Remaining Challenges of a transition to a microservice architecture

An interesting overview of the lesson learned during a migration to microservices is Netflix [19]. Reading the Netfix's experience. we need to acknowledge that there are various aspects that even a tool as PPTAM+ cannot solve. Some of them are connect to developers' capabilities and knowledge of the SUT (e.g., "Keep Code at a Similar Level of Maturity" between the monolith and the microservices system).

However, some benefits of using our approach and tools during the transition towards a microservice architecture remain and support the Netflix's experience. For example, with PPTAM+ we can identify that a performance degradation has happened for individual microservices ("Do a Separate Build for Each Microservice"), but also find possible causes and remedies like identifying microservice architecture patterns or antipatterns as possible causes of a performance degradation. With our approach, we can also evaluate the coupling between microservices and monitor it over the transition. By integrating advanced statistical tools in the R scripts of PPTAM+ like genetic algorithms, we can also evaluate architectural choices by exploiting existing literature on software coupling [20].

References

[1] J. Lewis and M. Fowler, "Microservices: a definition of this new architectural term," March 2014, https://martinfowler.com/articles/microservices.html, accessed on Aug 2, 2019.

[2] M. Kalske, N. Mäkitalo, and T. Mikkonen, "Challenges when moving from monolith to microservice architecture," in *Current Trends in Web Engineering*, I. Garrigós and M. Wimmer, Eds. Cham: Springer International Publishing, 2018, pp. 32–47.

[3] C. Kerstiens, "Give Me Back My Monolith," March 2019, http://www.craigkerstiens.com/2019/03/13/give-me-back-my-monolith/, accessed on Aug 2, 2019.

[4] V. F. Pacheco, *Microservice Patterns and Best Practices: Explore Patterns Like CQRS and Event Sourcing to Create Scalable, Maintainable, and Testable Microservices*. Packt Publishing, 2018.

[5] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, February 2015.

[6] V. Alagarasan, "Seven Microservices Anti-patterns," August 2015, https://www.infoq.com/articles/seven-uservices-antipatterns/, accessed on Aug 2, 2019.

[7] M. Richards, *Microservices AntiPatterns and Pitfalls*. O'Reilly Media, July 2016.

[8] T. Ōno, *Toyota Production System: Beyond Large-Scale Production*. Productivity Press, 1988.

[9] Z. Dehghani, "How to break a Monolith into Microservices," April 2018, https://martinfowler.com/articles/break-monolith-into-microservices.html, accessed on Aug 2, 2019.

[10] A. Avritzer, D. Menasché, V. Rufino, B. Russo, A. Janes, V. Ferme, A. van Hoorn, and H. Schulz, "Pptam: Production and performance testing based application monitoring," in *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19. New York, NY, USA: ACM, 2019, pp. 39–40. [Online]. Available: http://doi.acm.org/10.1145/3302541.3311961

[11] A. Janes, "Non-distracting, continuous collection of software development process data," in *Synergies Between Knowledge Engineering and Software Engineering*, 2018.

[12] G. Kiczales, "Aspect-oriented programming," *ACM Comput. Surv.*, vol. 28, no. 4es, Dec. 1996.

[13] A. Croll and S. Power, *Complete Web Monitoring: Watching your visitors, performance, communities, and competitors*. O'Reilly Media, 2009.

[14] Microsoft, ".NET Framework Development Guide, UI Automation Overview," 2016, https://msdn.microsoft.com/en-us/library/ms747327(v=vs.110).aspx, accessed on Aug 2, 2019.

[15] Apple, "Accessibility Programming Guide for OS X," 2016, https://developer.apple.com/library/mac/documentation/Accessibility/Conceptual/AccessibilityMacOSX/, accessed on Aug 2, 2019.

[16] V. Ferme and C. Pautasso, "A declarative approach for performance tests execution in continuous software development environments," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. ACM, 2018, pp. 261–272.

[17] H. Schulz, T. Angerstein, and A. van Hoorn, "Towards automating representative load testing in continuous software engineering," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18, 2018, pp. 123–126.

[18] A. Avritzer, V. Ferme, A. Janes, B. Russo, H. Schulz, and A. van Hoorn, "A quantitative approach for the assessment of microservice architecture deployment alternatives by automated performance testing," in *Proceedings of the 12th European Conference on Software Architecture (ECSA)*, 2018, pp. 159–174.

[19] T. Mauro, "Adopting Microservices at Netflix: Lessons for Architectural Design," https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/, accessed on Aug 2, 2019.

[20] I. Candela, G. Bavota, B. Russo, and R. Oliveto, "Using cohesion and coupling for software remodularization: Is it enough?" *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 3, pp. 24:1–24:28, 2016.