EasyChair Preprint

№ 10188

Microservice Test Process: Design and Implementation

Dmitrii Savchenko, Gleb Radchenko, Timo Hynninen and Ossi Taipale

May 17, 2023

MICROSERVICE TEST PROCESS: DESIGN AND IMPLEMENTATION

Dmitrii Savchenko[1], Gleb Radchenko[2], Timo Hynninen[1], Ossi Taipale[1]

Lappeenranta University of Technology[1], South Ural State University[2]
dmitrii.savchenko@lut.fi, gleb.radchenko@susu.ru, timo.hynninen@xamk.fi, ossi.taipale@lut.fi
Finland, Russia

Abstract: The objective of this study is to develop a test process approach for microservices. Microservice architecture can reduce the development costs of web systems and shift the accidental complexity from the application to the infrastructure. In this paper, we provide an analysis and description of the microservice test process, as well as an example implementation of the software system, that supports the described process. An example of usage demonstrates the advantages of our solution in the real environment.

Key words: microservices, testing, test techniques, testing service.

1. INTRODUCTION

The increasing amount of automation in information technology (IT) can usually be traced back to rising maintenance costs of IT departments, data centers and server rooms [15, 28]. Microservices offer a novel architecture for development, testing, deployment, and maintenance in contrast to the traditional monolithic architectural solutions. In cloud applications, the monolithic architecture and microservice architecture are the two conceptual approaches to development, testing, deployment, and maintenance. Monolithic applications are derived from the client-server architecture in which the components that implement the business logic of the application are collected in the application server [22]. Microservice architecture implies that the server application is divided into microservices – isolated, autonomous components, each running on its own virtual machine [17]. Each microservice implements a separate role in the application's business process, retains its own state in a separate database and communicates with the other microservices using Hypertext Transfer Protocol (HTTP) and Representational State Transfer (REST) architectural styles [8]. Each microservice has its own responsibility, so the system can easily be scaled to handle different loads. Microservices can be created using different programming languages and developed by different teams because of the loose coupling of the modules [17].

Nowadays, microservices are adopted by numerous major organizations in the software market, such as Amazon, eBay, or Netflix, which have migrated their infrastructure to the microservice architecture [25]. The microservice approach shifts the accidental complexity (problems that are produced by a programming language and environment) from the application to the infrastructure [2]. Increased complexity and interconnectivity of IT systems and devices increase the probability and seriousness of defects [26]. With a defect having the potential to go viral in minutes, senior managers focus on protecting their

corporate image. They view this as the key objective of quality assurance (QA) and testing [26]. This means that business requires practical solutions to support microservice testing. The analysis of how to test microservices should become an essential part of service design.

In this paper, we develop an approach to microservice testing. First, we present the design of techniques for microservice testing based on a literature review. Second, we introduce a process of microservice testing using the proposed techniques and an example implementation of this process as a microservice testing service. Third, we illustrate the service usage in a real environment.

2. LITERATURE REVIEW

The microservice architecture is often compared with the monolithic architecture. A monolithic system as an application that has its components connected to one thread or process [24]. With an increasing load, the application should be reconfigured to carry the required load. The monolithic approach does not require special deployment tools because there is usually no need to organize communications between separated modules. But large service providers, such as Netflix, an Internet video subscription service, and several other companies decided to implement their infrastructures in the microservice style [30]. Shifting the accidental complexity from the application out into the infrastructure was the main driver for the migration. Some companies have implemented their products as big monoliths and have later been forced to split these monoliths into reusable modules [25], so they rebuilt their infrastructure in accordance with the microservice style. Amazon also started with a large database in a monolithic architecture and later revised the entire infrastructure as service-oriented architecture [30].

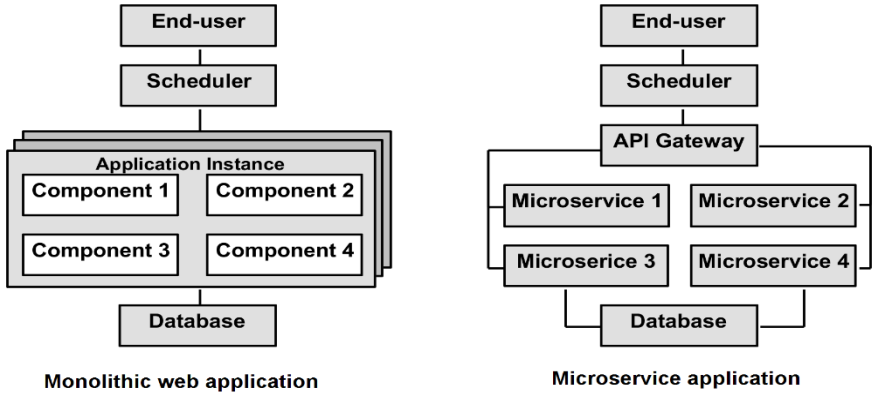

Fig 1. Differences between monolithic and microservice architectural styles

Fig 1 illustrates a comparison of the microservice style with the monolithic style. In the monolithic style, the application is built as one unit (Fig. 1, left). The server side in a monolithic application handles HTTP requests, executes domain-specific tasks, retrieves and updates data in the database, and selects and populates Hypertext Markup Language (HTML) views. This design approach usually leads to a situation where a change made to a small part of the application requires the entire monolith to be rebuilt, tested and deployed, and over time, it is difficult to maintain a good modular structure where changes affect only one

module within the system [17]. In the microservice style (Fig. 1, right) the system is divided into several independent microservices. The application can be scaled by applying the duplication ability of microservices and scheduling the load between different servers. The system is most useful in high availability systems with many reusable modules [14].

2.1. Microservice test design techniques

The microservice architectural style was introduced by Fowler and Lewis in 2014, so there is lack of microservice testing approaches [17]. Ford [9] describes the basic ideas of microservice systems development, monitoring, and testing, but a technical solution for microservice testing is missing. Netflix describes the microservices-based Platform-as-a-Service (PaaS) system and mechanisms, which are used to integrate legacy software into the platform [3]. Clemson [5] analyses the microservice testing strategies for the whole system and proposes the implementation of microservice testing using mocks (simulated objects that mimic the behavior of real objects in controlled ways). Ashikhmin et al. [1] describe a mock service generator based on RAML (RESTful API Modeling Language). Mock services are implemented as Docker containers [20] and they can be directly deployed in the microservice environment. Mock approach facilitates microservice development and testing and does not require much effort for maintenance.

Another popular approach to microservice testing is the fault injection [19]. It allows finding and eliminating possible unpredictable faults, such as database overloads. Meinke and Nycander offer a learning-based approach to evaluate the functional correctness of distributed systems and their robustness against injected faults [19]. This system simulates the communication faults between microservices and data sources (for example databases) and then checks that the system provides the expected output. The initial testing scenario evolves on every iteration and does not require any further manipulation [10]. Another approach to the microservice fault injection testing is described by Heorhiadi et al. [23]. The authors describe the Gremlin, a framework for systematically testing the failure-handling capabilities of microservices. This framework simulates the failures in communication between different microservices and other network components.

Described test design techniques do not fully cover microservice testing. Some of them consider microservice systems only as a set of service-oriented entities and concentrate, for example, on the integration level. However, they can be used in the design of the novel approaches.

2.2. Service-oriented architecture

The microservice architecture is closely related to the service-oriented architecture (SOA). SOA is defined by the Organization for the Advancement of Structured Information Standards (OASIS) as the paradigm for organizing and utilizing distributed capabilities under the control of different ownership domains [22]. This study uses a practical definition of SOA provided by Thomas: "SOA is a form of technology architecture that adheres to the principles of service-orientation. When realized through the Web services technology platform, SOA establishes the potential to support and promote these principles throughout the business process and automation domains of an enterprise" [7]. SOA imposes the following requirements when microservices are considered as a subset of SOA. They are reusable, loosely coupled, composable, autonomous, stateless, and share formal contracts and an abstract underlying logic [7]. The microservices return the same results for different requests with the same parameters and are discoverable because they operate in an environment that

decides which microservice receives the message. By microservices, we mean processes as a set of fine-grained services instead of building a monolithic service to cater to all requirements with one solution. Microservices might be considered as a subset of SOA and, therefore, they meet the same requirements on microservice systems.

Kalamegam and Godandapani [13] describe testing challenges from the viewpoint of different stakeholders and different test techniques that could be applied to SOA testing. The most effective way to test service-oriented software is to use a continuous end-to-end testing process that should validate requirements when it is developed, run and maintained. Jehan, Pill, and Wotawa [12] describe a Business Process Execution Language (BPEL), a constraint-based approach to SOA test case generation. The approach uses the BPEL definition for services to generate different test cases to produce the expected output [18]. However, this solution considers services as a black box and does not cover the integration of different services. Li et al. [18] describe another approach based on BPEL, and it takes into account both service interfaces and communications between different services. Such an approach implements gray-box testing of service-oriented software. Canfora and Di Penta [4] describe the challenges faced by different stakeholders involved in testing. They propose an approach to divide the testing process for service-oriented software into different levels and perform testing of different levels separately. BPEL and test levels division could be also used in microservice testing in addition to microservice test design techniques of different test levels.

2.3. Actors and agent-oriented systems

Microservices, multi-agent systems, and actors share a common architectural principle. They have a set of logically independent entities, which work together to provide a response to an external request [16]. However, unlike microservices, a multi-agent system is an approach to problem solving. The agents are not only programmable entities but they can be humans or any other external entities that provide information. The actor model differs from microservices in many aspects, but the main difference is that one actor can create another actor, but microservices cannot. An analysis of the existing test techniques for different models provides a set of approaches that can be used for microservice systems. For example, it is possible to apply the existing approach based on a detailed description of the input and output to test microservice communication [21].

Rahman and Gao [23] describe an approach to microservice testing based on behavior-driven development (BDD). BDD is a software development methodology, in which an application is specified and designed by describing how its behavior should appear to an outside observer [27]. In practice, the BDD testing of microservices could be implemented by using a BDD support library, such as Cucumber [6]. However, some approaches cannot be easily implemented for a microservice model. For example, Tasharofi et al. [29] present a testing framework that operates in accordance with the actor model. Such an analysis can be carried out only for actor systems because, unlike the actor model, a microservice cannot create a new instance of itself or any other microservice. Therefore, the actor system is a problematic approach to modeling or testing microservices since one of its prominent features violates the microservice architecture.

The analysis of the existing test techniques of microservice systems shows detached examples of microservice testing, but a comprehensive model of microservice testing is missing. We observed different test design techniques that could be applied to microservice testing, but according to our understanding, they cover the microservice testing process only partially. Our analysis showed, that testing of a single microservice is similar to any other

SOA entity testing, but there is higher complexity in the microservice infrastructure and deployment process.

3. TESTING SERVICE DESIGN AND IMPLEMENTATION

As said above, the biggest complexity of microservice testing is concentrated in infrastructure, and we decided to split the process of microservice testing by three different levels, according to existing software testing practices – component testing, integration testing, and system testing. Figure 2 describes the proposed test process for microservice testing.

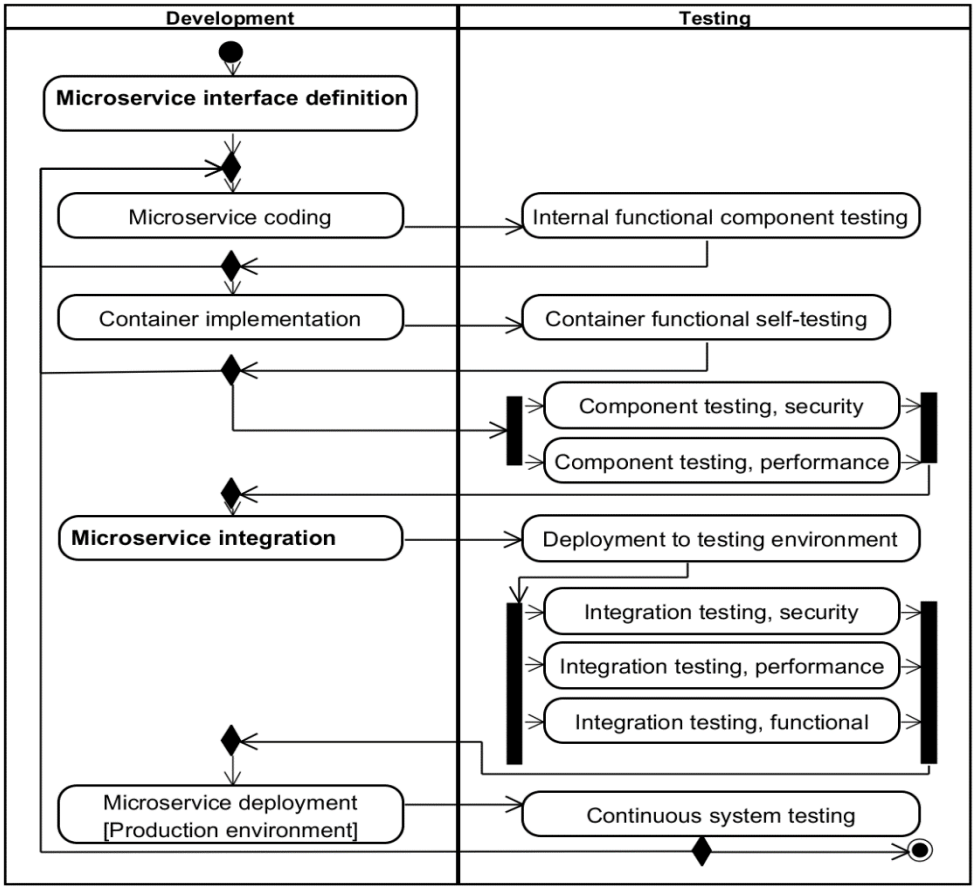

Fig. 2. Microservice test process

The following steps should be made when testing the microservice system in practice:

1. The automated testing of a microservice's interface and communication process requires the definition of the interface of every microservice as a test basis (it means, that each microservice has a JSON file that describes inputs, outputs, and their syntax or possible values).

2. According to the features of the programming language and the chosen implementation framework, the developer provides appropriate automated component test cases of the microservice's source code to ensure its compliance with the requirements (for example, JUnit for Java, Karma for JavaScript, RSpec for Ruby).
3. If the source code passes static tests and component tests, the microservice will be packed into a Docker container, and container self-tests (external interface testing, made by service itself) will be carried out to ensure that all of the components of the container are functioning correctly and the interface of the container corresponds to the interface definition provided during step 1.
4. If the self-tests are successful, security and performance tests will be carried out for the microservice. Steps 1-4 are run locally on the developer's machine.
5. If all of the tests are passed, the microservice test environment will be set up and integration tests for security, performance efficiency and functional suitability will be performed. This allows detecting issues in the microservice's orchestration, including load balancing, lifecycle management, and communication issues.
6. If the microservice passes all of the tests in the test environment, it can be deployed in the production environment. A continuous stability test is performed in the production environment according to the methods of deliberate provocation of random failures of the system components.

This approach allows microservice component, system, and integration testing. Also, the described process implies three different environments for microservices: local developer PC, test environment, and production environment.

The microservice testing service was created based on the described testing algorithm. The service facilitates the automatic testing of microservices during different phases of the microservice's life cycle from the component testing of the code to the stability testing of the system under test (SUT). In this service, the end-users can test the SUT and extend the testing service using their own test cases and designs. In building the artifact, the microservice testing service activities were selected from test level and test type examples. The microservice testing service example includes the following activities:

1. Component testing of the microservice source code.
2. Microservice self-testing: the microservice tests its own external interface.
3. Component testing of security.
4. Integration testing of security to determine if is it possible to intercept and/or change the contents of the messages between individual microservices. Security and isolation testing of the SUT.
5. Integration testing of performance efficiency to test the interaction's behavior under the load.
6. Integration testing of functional suitability to test the microservice's interactions.

Each test design and implementation is an external service because the microservice testing service uses a plug-in structure to enable end-users and designers to add more test designs and cases to the system.

3.1. Microservice testing service architecture

Since microservices can be implemented using different programming languages, frameworks or even operating systems, it is difficult to build one monolithic test software to handle different microservices and microservice systems. To solve this problem, the microservice testing service uses external containers that host different test designs and their

implementations. Microservice testing service provides HTTP API to enable easy integration of third-party test designs. The testing service is implemented in accordance with the microservice architecture to support and allow extensions. Fig. 33 depicts the proposed architecture.

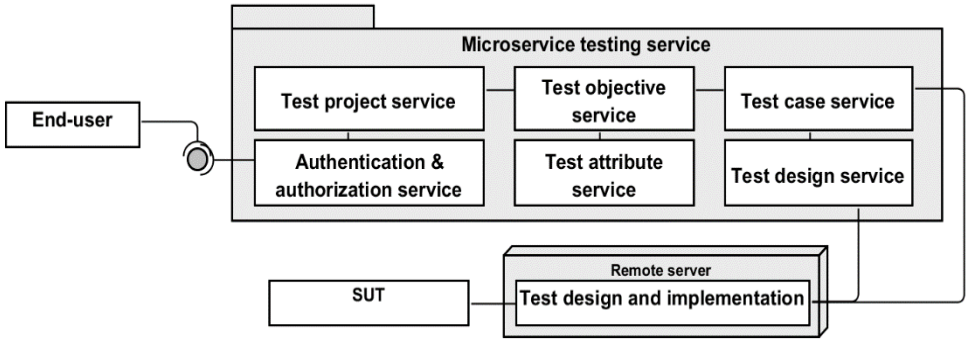

Fig. 3. Microservice testing service architecture

The authentication and authorization service identifies users, their roles, and permissions using their credentials when the end-user enters the testing service. The test project service and test objective service both provide create, read, update and delete actions for the test project entity and objective entity. Each test project and objective can have parameters that will be passed to underlying test objectives and cases to simplify the test configuration.

The test attribute service provides a list of associated test designs and cases according to application attributes. An application attribute is a characteristic that describes the features of the SUT, such as the implementation language, operating system or database type. In addition, this service can accept new links between application attributes and test designs and cases.

The test case service provides create, read, update and delete actions for test designs and cases. Each change of any test design or the case will trigger the test design service that registers all added or modified test designs and cases and parses their interfaces. Test cases are executed during test runs.

Fig. 44 describes the test project creation and run process.

1. The end-user enters his or her credentials in the authentication and obtains rights in the authorization.
2. The end-user creates a new test project with a name.
3. For the created project, the end-user creates a test objective with a name and attaches the desired test levels and types as application attributes.
4. According to the supplied application attributes, the service offers a set of test cases if available.
5. The end-user adds, creates or modifies the test cases, sets the order of execution and provides the required parameters.
6. To initiate the testing process, the end-user provides the final testing parameters, such as the microservice system location or programming language. The service executes the test cases.

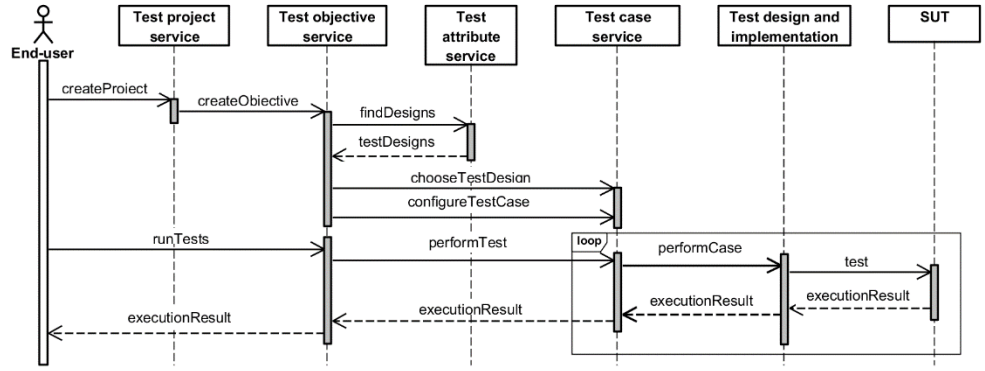

Fig. 4. Microservice testing service sequence diagram

3.3. The microservice testing service interface

The microservice testing service prototype has an HTTP API interface for external communication and a visual interface. According to the scope of the SUT, the visual interface is planned for the configuration of test projects, objectives, test designs, test cases, names, parameters, the test execution order, etc. Using the HTTP API interface, the end-user can run tests to reach test objectives. HTTP API can also return the test execution status, error log and other test-related information.

To create the test objective, the end-user specifies the name of the objective, describes it and chooses one available application attribute or more. The testing service uses this information to offer a list of applicable test designs and test cases. After creating the objective, the end-user can set 'static' parameters. 'Static' parameters are not changed from run to run, for example, programming language or database endpoint. Moreover, the end-user can set the order of the test case execution and fault logic; for example, whether the test process should stop if one test case fails. The objective or project can be run using the HTTP request (Table 1). The HTTP request provides the necessary information for the test execution linked with the objectives except for 'static' parameters. Table 1 provides the source code repository location, programming language, and location of the SUT.

Table 1. Example of a request to run tests with available test methods

```
POST /api/testing {
 test_set_type: "project",
 test_set_id: 1,
 test_params: {
  source:            "git@github.com:skayred/netty-upload-
example.git",
  methods: "all",
  language: "java",
  endpoint: "https://netty-upload.herokuapp.com"
 }
}
```

4. MICROSERVICE TESTING PROCESS EVALUATION

To estimate improvements in the real environment, we implemented a Docker container, which contains the microservice testing service implementation (source code available at https://bitbucket.org/skayred/cloudrift2). Typical example for the evaluation was selected because the number of different test scenarios is unlimited. The evaluation was conducted on an open source microservice example (https://github.com/kbastani/spring-cloud-microservice-example). To demonstrate the typical workflow used in Continuous Integration with real-world microservices, we have implemented the scenario shown in Figure 5.

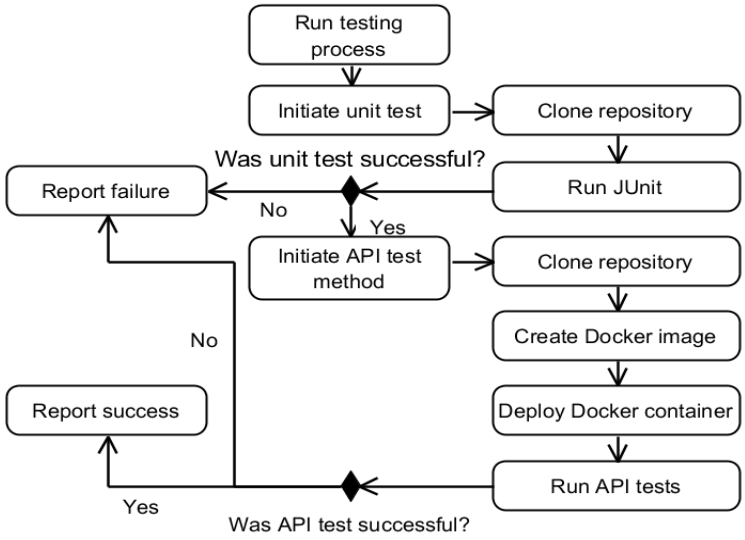

Fig. 5. Workflow in the evaluation example

This scenario follows the process, described in section 3, and enables microservice component, system, and integration testing, concurrently following the guidelines of ISO/IEC 29119 standards [11] and implementing the collected observations. The scenario was implemented using two simple Ruby scripts, that were calling command line interface and executed described commands. The success of the operation was determined using exit codes: 0 for a successful test and any other for a failed test. This workflow illustrates how the microservice testing service can be integrated into the Continuous Integration infrastructure. The pluggable structure allows third-party developers to add new test designs and cases and use them for testing. The HTTP API of the testing service helps end-users to integrate continuous integration or continuous delivery systems. Described testing service supports creating, reusing and executing test cases according to the end-user request, but it does not yet support run-time monitoring.

5. CONCLUSION

Microservices offer an architecture with fine-grained, isolated components. In this paper, we offer an approach to microservice testing. Microservices are tested using test techniques modified from existing test techniques and automated as a service. We derived

the test design and implementation process for microservices, selected example test levels and test types, derived corresponding examples of test techniques, and generalized the results as a generic testing service for microservices.

Our analysis showed, that the testing of a single microservice is similar to any other SOA entity testing, but there is complexity in the microservice infrastructure and deployment process. The described microservice testing process could be implemented using different Continuous Integration system (for example, TeamCity and Jenkins), but microservices could be implemented using different environments, that makes the testing infrastructure more complex. The described process was illustrated using the open source example, based also on the idea of microservices.

This paper presents an example solution and its evaluation. The microservice test techniques and test automation, together with the benefits of the microservice architecture, reduce testing, deployment and maintenance costs. This study helps both researchers and practitioners to develop microservice test techniques with test cases and to offer them as an automated service. The described service architecture and implementation allow the plug-in of new test techniques and cases, making the service extensible and capable of covering the most common scenarios in microservice testing and applicable to the testing of traditional software.

According to the usage example, the proposed system facilitates basic microservice testing. However, further study is required to assess its actual performance and effectiveness in increasing the number of available test designs and cases. In future work, we will expand the described testing service to support more test techniques, test cases, and run-time monitoring.

6. ACKNOWLEDGEMENTS

This study is partially supported by the Maintain project (http://www2.it.lut.fi/projects /maintain/) funded by the Finnish Funding Agency for Innovation (TEKES) 1204/31/2016, the EU Erasmus Global Mobility programme, act 211 Government of the Russian Federation, contract No. 02.A03.21.0011, and the Russian Foundation for Basic Research (RFBR) 18-07-01224-a.

REFERENCES

[1] Ashikhmin, N., Radchenko, G., Tchernykh, A. RAML-based mock service generator for microservice applications testing. *Communications in Computer and Information Science*. 2017. vol. 793. pp. 456–467.

[2] Brooks, F. No Silver Bullet: Essence and Accident of Software Engineering, *IEEE Software*, vol. 20, p. 12, 1987.

[3] Bryant, D. *Prana: A Sidecar Application for NetflixOSS-based Services*. [Online]. Available: https://www.infoq.com/news/2014/12/netflix-prana. [Accessed: 01-Aug-2017].

[4] Canfora, G., Di Penta, M. Service-oriented architectures testing: A survey. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2009. vol. 5413. pp. 78–105.

[5] Clemson, T. *Testing Strategies in a Microservice Architecture.* [Online]. Available: https://martinfowler.com/articles/microservice-testing/. [Accessed: 31-Jul-2017].

[6] Dees, I., Wynne, M., Hellesoy, A. Cucumber Recipes Automate Anything with BDD Tools and Techniques, *The Pragmatic Bookshelf*, p. 266, 2013.

[7] Erl, T. SOA: Principles of Service Design, *2003 Symposium on Applications and the Internet Workshops, 2003. Proceedings.*, 2005.

[8] Fielding, R. Representational state transfer, *Architectural Styles and the Design of Netowork-based Software Architecture*, 2000.

[9] Ford, N. *Building Microservice Architectures*, 2015. [Online]. Available: http://nealford.com/downloads/Building_Microservice_ Architectures(Neal_Ford).pdf. [Accessed: 15-Oct-2015].

[10] Heorhiadi, V., Rajagopalan, S., Jamjoom, H., Reiter, M. K., Sekar, V. Gremlin: Systematic Resilience Testing of Microservices. *Proceedings - International Conference on Distributed Computing Systems*. 2016. vol. 2016–Augus. pp. 57–66.

[11] ISO/IEC/IEEE. 29119-1:2013 - ISO/IEC/IEEE International Standard for Software and systems engineering — Software testing — Part 1: Concepts and definitions, *ISO/IEC/IEEE 29119-1:2013(E)*, vol. 2013, pp. 1–64, 2013.

[12] Jehan, S., Pill, I., Wotawa, F. Functional SOA testing based on constraints. *2013 8th International Workshop on Automation of Software Test, AST 2013 - Proceedings*. 2013. pp. 33–39.

[13] Kalamegam, P., Godandapani, Z. A survey on testing SOA built using web services, *International Journal of Software Engineering and its Applications*, vol. 6, no. 4, pp. 91–104, 2012.

[14] Kant, N; Tonse, T. Karyon: The nucleus of a Composable Web Service, *Karyon: The nucleus of a Composable Web Service*, 2013. [Online]. Available: http://techblog.netflix.com/2013/03/karyon-nucleus-of-composable-web-service.html. [Accessed: 08-Oct-2015].

[15] Khosla, V. Keynote speech, *Structure Conference, San Francisco*, 2016.

[16] Krivic, P., Skocir, P., Kusek, M., Jezic, G. Microservices as agents in IoT systems. *Smart Innovation, Systems and Technologies*. 2018. vol. 74. pp. 22–31.

[17] Lewis, J., Fowler, M. Microservices, *http://martinfowler.com*, 2014. [Online]. Available: http://martinfowler.com/articles/microservices.html.

[18] Li, Z. J., Tan, H. F., Liu, H. H., Zhu, J., Mitsumori, N. M. Business-process-driven gray-box SOA testing, *IBM Systems Journal*, vol. 47, no. 3, pp. 457–472, 2008.

[19] Meinke, K., Nycander, P. Learning-based testing of distributed microservice architectures: Correctness and fault injection. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2015. vol. 9509. pp. 3–10.

[20] Merkel, D. Docker: lightweight Linux containers for consistent development and deployment, *Linux Journal*, vol. 2014, no. 239. p. 2, 2014.

[21] Nguyen, C. D., Perini, A., Tonella, P. Ontology-based Test Generation for Multiagent Systems, *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3*, pp. 1315–1320, 2008.

[22] OASIS. Reference Model for Service Oriented Architecture 1.0. OASIS Standard, *Public Review Draft 2*, no. October, pp. 1–31, 2006.

[23] Rahman, M., Gao, J. A reusable automated acceptance testing architecture for microservices in behavior-driven development. *Proceedings - 9th IEEE International Symposium on Service-Oriented System Engineering, IEEE SOSE 2015*. 2015. vol. 30. pp. 321–325.

[24] Richardson, C. *Monolithic Architecture pattern*. [Online]. Available: http://microservices.io/patterns/monolithic.html. [Accessed: 16-Nov-2017].

[25] Runeson, P. A survey of unit testing practices, *IEEE Software*, vol. 23, no. 4, pp. 22–29, 2006.

[26] Sogeti. WORLD QUALITY REPORT 2015-16, 2016.

[27] Solís, C., Wang, X. A study of the characteristics of behaviour driven development. *Proceedings - 37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011*. 2011. pp. 383–387.

[28] Stanley, M., *Morgan Stanley. Software Sector Is Set for Strong Growth in 2017* | Morgan Stanley. [Online]. Available: https://www.morganstanley.com/ideas/software-sector-growth. [Accessed: 31-Jul-2017].

[29] Tasharofi, S., Karmani, R. K., Lauterburg, S., Legay, A., Marinov, D., Agha, G. *TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs*, Springer, Berlin, Heidelberg, 2012, pp. 219–234.

[30] Thönes, J. Microservices, *IEEE Software*, vol. 32, no. 1. pp. 113–116, 2015.

Information about the authors:

Dmitrii Savchenko – Researcher and doctoral student at the Lappeenranta University of Technology. Main research topics include service-oriented architecture, software development practices, and microservices. Acquired the degree of Master of Science in Database Engineering from South Ural State University in 2014.

Gleb Radchenko – Director of School of Electrical Engineering and Computer Science of South Ural State University. Main research topics include high-performance systems, cloud computing, distributed computing and scientific workflow. Acquired the degree of Ph.D. from Moscow State University in 2010.

Ossi Taipale – Doctor of Science with the Lappeenranta University of Technology, representative of Finland in the ISO/IEC WG26 that develops the software testing standards. Main research topic if software testing.

Mr. Timo Hynninen – Researcher and doctoral student at the Lappeenranta University of Technology. Main research topics include areas of software quality, software measurement, and long-distance developer platforms. Acquired the degree of Master of Science in Computer Science from the Lappeenranta University of Technology in 2016.

Manuscript received on 31 May 2018