Dmitrii Savchenko

TESTING MICROSERVICE APPLICATIONS

ACTA UNIVERSITATIS LAPPEENRANTAENSIS 868

Dmitrii Savchenko

TESTING MICROSERVICE APPLICATIONS

Thesis for the degree of Doctor of Science (Technology) to be presented with due permission for public examination and criticism in the Auditorium of the Student Union House at Lappeenranta-Lahti University of Technology LUT, Lappeenranta, Finland on the 18th of October, 2019, at noon.

The thesis was written under a joint doctorate agreement between Lappeenranta-Lahti University of Technology LUT, Finland and South Ural State University, Russia and jointly supervised by supervisors from both universities.

Acta Universitatis
Lappeenrantaensis 868

Supervisors Adjunct Professor Ossi Taipale
LUT School of Engineering Science
Lappeenranta-Lahti University of Technology LUT
Finland

Associate Professor Jussi Kasurinen
LUT School of Engineering Science
Lappeenranta-Lahti University of Technology LUT
Finland

Associate Professor Gleb Radchenko
School of Electrical Engineering and Computer Science
Department of System Programming
Federal State Autonomous Educational Institution of High Education
South Ural State University (National Research University)
Russian Federation

Reviewers Professor Timo Mantere
Dept. of Electrical Engineering and Automation
University of Vaasa
Finland

Professor Markku Tukiainen
School of Computing
University of Eastern Finland
Joensuu
Finland

Opponent Professor Ita Richardson
Lero - The Irish Software Research Centre
University of Limerick
Ireland

ISBN 978-952-335-414-2
ISBN 978-952-335-415-9 (PDF)
ISSN 1456-4491
ISSN-L 1456-4491

Lappeenranta-Lahti University of Technology LUT
LUT University Press 2019

Abstract

Dmitrii Savchenko
Testing microservice applications
Lappeenranta, 2019
59 p.
Acta Universitatis Lappeenrantaensis 868
Diss. Lappeenranta-Lahti University of Technology LUT
ISBN 978-952-335-414-2, ISBN 978-952-335-415-9 (PDF), ISSN-L 1456-4491, ISSN 1456-4491

Software maintenance costs are growing from year to year because of the growing software complexity. Currently, maintenance may take up to 92 percent of the whole project budget. To reduce the complexity of the developed software, engineers use different approaches. Microservice architecture offers a novel solution to the problem of distributed applications' complexity. The microservice architecture relies on the comprehensive infrastructure, which reduces the complexity of the application.

Several large companies have successfully adopted the microservice architecture, but not many studies have examined microservice applications testing and quality assurance. In addition, smaller companies are showing interest in the microservice architecture and trying to adopt it using different infrastructure solutions to reduce software development and maintenance costs or to integrate legacy software into newly developed software. Therefore, we explore the possible approaches to microservice testing, describe the microservice testing methodology, and use design science to implement the microservice testing service that adopts the described methodology.

This study provides an analysis of different software testing techniques and offers a methodology for microservice testing. In addition, an example implementation illustrates the described methodology.

Keywords: microservices, testing, cloud computing, infrastructure

Acknowledgements

I consider it a blessing to have had the opportunity to carry out this research work. It would not have been possible without the support of many wonderful people. I am not able to mention everyone here, but I acknowledge and deeply appreciate all your invaluable assistance and support.

I would like to thank my supervisors, Adjunct Professor Ossi Taipale, Associate Professor Jussi Kasurinen, and Associate Professor Gleb Radchenko, for their guidance, encouragement, and contribution throughout this research. Thank you for all your efforts and for providing a great research environment. I wish to thank the reviewers of this dissertation, Professor Timo Mantere and Professor Markku Tukiainen for your valuable comments and feedback that helped me to finalize the dissertation.

For financial support, I would like to acknowledge The Finnish Funding Agency for Technology and Innovation (TEKES) and the companies participating in Maintain research project. I appreciate the support and assistance of my colleagues at LUT School of Engineering Science. Special thanks to Tarja Nikkinen, Ilmari Laakkonen, and Petri Hautaniemi, for providing administrative and technical support.

My dear wife Anna, I am truly grateful for your love, patience, understanding, and support.

Dmitrii Savchenko
September, 2019
Lappeenranta, Finland

Contents

List of publications

Publication I

Savchenko D. and Radchenko G. (2014). Mjolnirr: private PaaS as distributed computing evolution. *37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 401-406.

Publication II

Savchenko D., Radchenko G., and Taipale O. (2015). Microservice validation: Mjolnirr platform case study. *38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 248-253.

Publication III

Savchenko D., Ashikhmin N., and Radchenko G. (2016). Testing-as-a-Service approach for cloud applications. *IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*, pp. 428-429.

Publication IV

Savchenko D., Radchenko G., Hynninen T., and Taipale O. (2018). Microservice test process: Design and Implementation. *International Journal on Information Technologies and Security.* ISSN 1313-8251, vol. 10, No 3, 2018, pp. 13-24.

Publication V

Savchenko D., Hynninen T., and Taipale O. (2018). Code quality measurement: case study. *41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 1455-1459.

Author's contribution in the listed publications

For *Publication I*, the author contributed the idea for the Mjolnirr platform, developed the platform, integrated it with the UNICORE grid system, and wrote the majority of the publication.

For *Publication II*, the author gathered information about microservice systems and different approaches to the test process, participated in the development of the microservice testing methodology, and wrote the majority of the publication.

For *Publication III*, the author implemented the prototype of the microservice testing service and wrote the majority of the publication.

For *Publication IV*, the author summarized the knowledge about the microservice testing service, evaluated it, and wrote the majority of the publication.

For *Publication V*, the author participated in the development of the architecture of the Maintain project and implemented the prototype of the Maintain system.

Symbols and abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
DevOps	Development and Operations
HAML	HTML Abstraction Markup Language
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IEC	International Electrotechnical Commision
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
IaaS	Infrastructure as a Service
JSON	JavaScript Object Notation
JS	JavaScript
PaaS	Platform as a Service
REST	Representational State Transfer
SDK	Software Development Kit
SOA	Service-Oriented Architecture
SUT	System Under Test
SaaS	Software as a Service
UNICORE	Uniform Interface to Computing Resources

Chapter I

Introduction

The budgets of the global IT sector are steadily growing from year to year, and that indicates an increase in the complexity of problems, solved by the software being developed (Stanley, 2017). Companies try to reduce development and maintenance costs, and therefore attempt to reduce software complexity. The complexity of software has two varieties: accidental complexity and essential complexity (Brooks, 1987). Accidental complexity is the set of problems that are provoked by engineering tools and can be fixed by software engineers, while essential complexity is caused by the subject area and cannot be ignored (Brooks, 1987). According to the definition, essential complexity is impossible to reduce, but accidental complexity can be shifted to some automated infrastructure. Companies and developers have attempted to follow this approach and have created different software development techniques as a result (Thönes, 2015).

Web service architecture was a response to the rising market demands and consumer dissatisfaction in the security and reliability of software on the market when the web service concept emerged (Erl, 2005). The main idea of web services was the provision of remote resources, which may belong to different owners (OASIS, 2006). Papazoglou and Georgakopoulos (2003) state that the services are open, self-determining software components that provide a transparent network addressing and supporting the fast building of distributed applications. However, over time, for example, a single database may grow too large to store and process with a single service. This fact has led to the separation and orchestration of more than one service (Thönes, 2015). Cloud computing, a "promising paradigm that could enable businesses to face market volatility in an agile and cost-efficient manner" (Hassan, 2011), then took form and was widely adopted by software engineers. The concept of cloud computing and a Platform-as-a-Service (PaaS) approach allowed developers to bypass the physical restrictions of hardware and use virtualized resources, for example, disk space or CPU time. Such opportunities led to the ability to implement a single application as a set of independent services, where each dedicated service has its own responsibility and can be managed automatically. This approach enables software developers to reduce development complexity by shifting accidental complexity to the infrastructure and focusing on the essential complexity of the

problem (Thönes, 2015). The described approach drove the emergence of a microservice architecture.

Microservice architecture implies that the application is implemented as a set of isolated, independent and autonomous components, working in their own virtual machines (Merkel, 2014). A single microservice provides transparent access to its functions and implements dedicated business capability. Microservices usually store their state in an isolated database and communicate with other services using some non-proprietary protocol; microservice interfaces are usually implemented in accordance with the REST API style (Fielding, 2000). Compared to a more traditional client-server architecture built around a single database, microservice architecture has several advantages: as long as each microservice has its own responsibility and is logically isolated, the workload can be scaled according to a required external load. In addition, microservices may be developed and maintained by different teams. The distributed nature of microservices also enables developers to use different programming languages, frameworks or operating systems within a single microservice application to reduce development and maintenance complexity. Responsibility isolation also leads to the reusability of existing microservices because of weak coupling between microservices. The weak coupling also enables developers to modify some parts of the system without changing or redeploying the whole system if a microservice contract is not changed. This makes systems development and maintenance less costly and more efficient, as has been illustrated by Amazon, Netflix, and eBay (Runeson, 2006). These companies implemented their infrastructure according to the microservice style because they faced challenges in their software in scaling and adapting to the high load.

Growing software complexity and coupling leads to a higher probability and seriousness of faults (Sogeti, 2016). Sogeti's report states that chief executive officers and IT managers should concentrate on the quality assurance and testing of their products to prevent possible faults and, therefore, to prevent major financial losses for the company (Sogeti, 2016). Those goals may be reached only through comprehensive software testing. This study describes a microservice testing service that gives companies special testing and deployment instruments. Those instruments enable development teams to test the microservice application as a whole, as well as dedicated microservices. This type of testing usually imposes additional requirements on the developed software and makes the development process somewhat predefined. Such an approach may increase development costs.

This study focuses on microservice testing methodology development, testing service development, and evaluation.

Chapter II

Microservice testing

Microservice architecture is a relatively new development approach and mostly adopted by large companies. Microservice development is not as formalized as other development approaches, so each implementation may have different features, and the only similarity is the infrastructure. According to our knowledge, there are no well-known or widely adopted general microservice application testing approaches. The implementation of the microservice testing service requires microservice architecture analysis to highlight the similarities between microservice architecture and other distributed computing architectures and approaches and to describe the general approach to microservice application testing. As the starting point, we used the ISO/IEC 29119 software testing standard and the ISO/IEC 25000 software quality standard series (ISO 25010, 2011; ISO 29119, 2013).

Clemson introduces possible microservice testing strategies (Clemson, 2014). This study describes microservice testing at the component, integration, and system levels, as well as mock testing. Mock testing is aimed at creating entities that are simplified representations of real-world objects. Such an approach is useful in microservice system testing but is not enough for comprehensive microservice application testing.

Other distributed computing techniques appear to be very similar, and they are applied to different levels of business logic to implement the system. For example, the actor programming model operates actors: programmatic entities that can make local decisions, send and receive messages from other actors, and create new actors. This model is used within a single computational node or local network to imitate a large number of entities communicating with each other (Tasharofi et al., 2012). On the other hand, service-oriented architectures are aimed at the wider service provisioning – for example, the Internet.

2.1 Microservice architecture premises

Microservice architecture is a currently popular approach to building big systems (Thönes, 2015). The microservice approach is mostly used in connection with high load web services. At first glance, the microservice architecture looks very similar to the existing

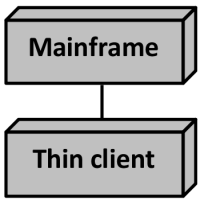

Figure 2.1: Mainframe client-server architecture

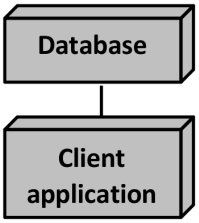

Figure 2.2: Two-tier architecture

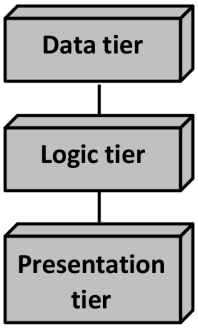

Figure 2.3: Three-tier architecture

software development approaches, for example, service-oriented architecture, actor programming, or agent-oriented programming, but it cannot be described using existing software development approaches. Therefore, we describe the premises that have led to microservice architecture derivation.

In the 70s, mainframe computers used to be expensive, and their computational resources were shared between several users. To address this issue, access to the mainframes was implemented through remote terminals (King, 1983). This approach preceded the client-server architecture because it allowed access to a remote resource through the network using a lightweight terminal (Figure 2.1). Client-server architecture also implies that one or more clients together with a server are parts of a single system to provide remote resource usage (Sinha, 1992). Client-server architecture also concentrates business logic in one place. Business logic contains the rules that determine how data is processed within a domain (Wang and Wang, 2006).

The client-server architecture later evolved into a two-tier architecture. A two-tier architecture implies that the software architecture is separated into two tiers: the client tier and the server tier (Gallaugher and Ramanathan, 1996). This architecture is usually implemented as a set of clients working with a single remote database. Business logic is implemented on the client tier, while the server tier is only responsible for data storage (Figure 2.2). This approach has several drawbacks that limit its applicability: a large number of connected clients can produce high load on a server, and this connection is usually not secured. Consequently, a two-tier architecture is usually implemented within the local network of one organization: the local network may be physically separated from the global network, while the number of clients is conditionally constant. In addition, any business logic change leads to a need to manually update all clients, which may be difficult within a global network (Gallaugher and Ramanathan, 1996). These drawbacks may be solved using security policies within the enterprise's local network.

Global network development has led to the evolution of distributed systems and remote resource delivery rules, and the two-tier architecture has been replaced by a three-tier one. A three-tier architecture implies that the architecture is logically divided into three levels:

the presentation tier, the logic tier, and the data tier (Helal et al., 2001) (Figure 2.3):

1. The *presentation tier* handles the user interface and communication. This tier is usually the only tier available to end-users, and end users cannot directly communicate with the data tier. In addition, the presentation tier usually has a minimal amount of business logic (Helal et al., 2001).
2. The *logic tier* is responsible for the business and domain logic of the application. This tier transforms data from the data tier to the domain logic objects for the presentation tier and handles requests that the end-user submits using the presentation tier. The software implementation of the logic tier is called the *application server* (Helal et al., 2001).
3. The *data tier* provides persistent data storage. This tier is usually implemented as a database server and used as a storage. Database management systems offer different functions to implement parts of the business logic in the database, such as stored procedures, views, and triggers, but in the three-tier architecture, those functions are often considered as bad practices because business logic should be implemented only on the logic tier (Helal et al., 2001).

Scalability is the capability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged to accommodate that growth (Bondi, 2000). A three-tier architecture provides higher scalability capabilities than a two-tier architecture because a three-tier architecture relies on thin clients and implements business logic in a single place. The three-tier architecture is usually scaled with new instances of the application server. Such an approach to scaling can be taken if the system is implemented as a black box that accepts a specific set of requirements as an input, processes those parameters, and discards the session (Bennett et al., 2000). This approach increases the load on the data tier and makes it a bottleneck (Fox and Patterson, 2012), requiring many resources for scaling.

A three-tier architecture solves the essential problems of a two-tier architecture, but a rising number of clients still poses challenges to the system scalability. To address the growing number of clients and the increasing system complexity, developers started to split web services into several logical parts (Sahoo, 2009). Each of those parts handles a slice of the business logic and can be scaled separately. This approach offers less overhead for large web systems, but also requires service orchestration and communication (Aalst, 2013). It is known as service-oriented architecture (SOA) (Gold et al., 2004), a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains (OASIS, 2006). SOA implies that the system is implemented as a set of logically independent services, communicating using standardized protocol (OASIS, 2006). Several different implementations of SOA and communication protocols incompatible with each other, which made the adoption difficult. In addition, SOA services were usually extensive enough to make deployment and maintenance more difficult than in a single web service (Fox and Patterson, 2012). However, SOA enables software companies to divide their software development between different teams with different backgrounds, and sometimes those teams may be geographically distributed. This approach is known as global software development (GSD) (Herbsleb and Moitra, 2001) and has successfully been adopted by the software development community.

Computational resource virtualization has enabled software engineers to use remote resources more easily and to reduce maintenance cost (Linthicum, 2010). A virtual machine offers the same environment for development, testing and production, and it may be configured once and then replicated to several physical servers. Service-oriented architecture, based on virtualized resources, has led to the emergence of microservice architecture.

Several large companies have successfully implemented the microservice architecture and published results indicating that microservice architecture may reduce development and maintenance costs. For example, Netflix has presented its Platform as a Service (PaaS) infrastructure solution and legacy service integration (Bryant, 2014).

Microservices are built around business capabilities and deployed independently using special deployment automation (Lewis and Fowler, 2014). There are various solutions for container virtualization and microservice automation, including those presented by, for example, Vamp (2014), Docker (Docker, 2014; Merkel, 2014), and Mesosphere (2014). The key difference between a microservice architecture and other distributed computing approaches is the ability to create new microservices: in the microservice architecture, only the infrastructure can create new instances of microservices to handle the changing load. In other distributed computing approaches, intelligent agents, for example, can create new agents if the task requires this. Microservice platforms usually include automatic scaling mechanisms and the gathering of performance metrics, but they lack testing features that are necessary for complex microservice applications.

2.1.1 Cloud computing

The idea of cloud computing was first mentioned in 1996, and this idea is close to utility computing (Feeney et al., 1974). Utility computing is a concept of resource delivery on demand, just like water and electricity. It means, that users should have easy network access to remote pools of configurable computational resources on demand (Bohn et al., 2011).

Cloud computing may be defined as a parallel and distributed system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one unified computing resource or more based on service-level agreements established through negotiation between service providers and consumers (Buyya et al., 2009). This definition highlights two features of cloud computing: virtualization and dynamic provisioning. Virtualization is the abstract representation of computing resources that enables a single physical computational node to run several different operating system instances (Barham et al., 2003; Buyya et al., 2009). Dynamic provisioning implies that computational resources – for example, CPU time, RAM, and disk space – are available on demand and paid upon use (Lu and Chen, 2012; Buyya et al., 2009). From the customer point of view, cloud computing may be defined as the applications delivered as services over the internet and the hardware and system software in the data centers that provide those services (Armbrust et al., 2009). This definition mostly concentrates on service provisioning but ignores underlying technical aspects, such as the virtualized infrastructure.

The National Institute of Standards and Technologies (NIST) defines cloud computing as a model for enabling ubiquitous, convenient, on-demand network access to a shared

pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction (Bohn et al., 2011; Mell and Grance, 2011). The NIST definition distinguishes three basic service models (Mell and Grance, 2011):

1. *Software as a Service (SaaS)* represents the consumer's capability to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through a client interface, such as a web browser. The consumer does not manage or control the underlying cloud infrastructure including a network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific configuration settings (Bohn et al., 2011; Mell and Grance, 2011; Olsen, 2006).

2. *Platform as a Service (PaaS)* is the capability to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including a network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment (Bohn et al., 2011; Mell and Grance, 2011).

3. *Infrastructure as a Service (IaaS)* is a service model that implies that customers control computational resources, storage, the network, and other resources. For example, customers can install and run custom software, such as operating systems, frameworks, or applications. A customer can control and choose the provided services, such as firewalls and DNS settings. The physical network, host operating systems, physical storage, and other physical infrastructure are controlled by the cloud provider (Bohn et al., 2011; Mell and Grance, 2011).

Cloud computing service models are not limited to SaaS, PaaS, and IaaS. Some studies define additional service models: for example, Software Testing as a Service (STaaS) (Aalst, 2010), or even Everything as a Service (XaaS) (Duan et al., 2015), but these models usually fall under the scope of the three basic ones.

Cloud computing is difficult to implement without virtualization technologies (Yau and An, 2011). Virtual resources allow to abstract the underlying logic from the environment and bypass the physical limitations of the hardware, for example, the geographical location. In addition, virtual machines may be used on a higher level of abstraction. For instance, PaaS allows to host virtual containers and adopt such a system for a changing load with container duplication (Rhoton and Haukioja, 2011). In practice, containers are often understood as a lightweight equivalent of virtual machines. Containers make available protected portions of the operating system – it means that containerization is based on isolated root namespaces, supported by the operating system kernel. Two containers running on the same operating system do not know that they are sharing resources because each has its own abstracted network layer, processes, and so on (Merkel, 2014). Containerization technology is closely linked with the term 'DevOps'. DevOps represents the combination of software development (*dev*elopment) and system administration (*op*erations) (Ebert et al., 2016). Containerization technology enables developers

to use the same environment during development, testing, or production deployment, and DevOps also widely used in microservice development (Balalaie et al., 2016). DevOps, cloud computing, and SOA were linked even before microservice architecture introduction (Hosono et al., 2011). All of these technologies together aimed to shorten software delivery time; in other words, continuous delivery (Pawson, 2011) – an approach that is intended for the manual delivery of new versions of the software.

Cloud computing reduces hardware expenses and overheads in software development, especially in small projects, because a customer can rent only the resources that they need, and pay only for the resources that they actually used (Leavitt, 2009; Marston et al., 2011). This paradigm is known as pay-per-use (Durkee, 2010). Generally, cloud computing makes the software engineering process more agile and cost efficient (Patidar et al., 2011), but imposes more requirements on the developed software in exchange (Grundy et al., 2012; Sodhi and Prabhakar, 2011). For example, SaaS applications require greater effort in testing because they may deal with millions of customers and should be able to fulfill their requirements (Riungu-Kalliosaari et al., 2013).

2.1.2 Microservice architecture

In this study, we understand microservice architecture as a subset of SOA because microservice systems meet all of the requirements, imposed to SOA (Papazoglou and Georgakopoulos, 2003): they are reusable, have a standardized service contract, are loosely coupled, encapsulate business logic, are composable, are autonomous, and are stateless. In addition, it is also possible to test dedicated microservices using existing SOA testing mechanisms or tools.

Microservice architecture is usually compared with the more common three-tier architecture. The application server in the three-tier architecture usually works in one operating system process. Such a system is easy to develop and maintain, but it is generally hard to scale for a changing load (Sneps-Sneppe and Namiot, 2014). Microservice systems are more complex to develop, but the microservice architecture is intended to be scalable. As the system consists of small services, infrastructure can create new instances of the most loaded microservices, creating less overhead and requiring less maintenance because microservices rely on containerization technology instead of virtualization (Kang et al., 2016). This has led several large companies to adopt microservices (Thönes, 2015). For example, Thönes (2015) describes reasons that made Netflix, a large media provider, to refactor its whole infrastructure according to the microservice style. Netflix decided to shift the accidental complexity from the software to the infrastructure because there are currently many ways to manage accidental complexity at the infrastructure level: programmable and automated infrastructure, cloud services, and so on. Netflix is an example of the successful implementation of the microservice system and illustrates how to change the existing infrastructure to the microservice one. In companies such as Netflix, eBay, and Amazon, the microservice approach their IT solutions less expensive to maintain as well as makes them more stable and predictable (Thönes, 2015). Amazon originally started with a monolith web service with a single database aimed for online shopping but later faced performance issues especially during holidays, when the number of customers rose. The wave of clients could be handled by increasing the amount of hardware, but this hardware was not in use during the working days. To solve this issue,

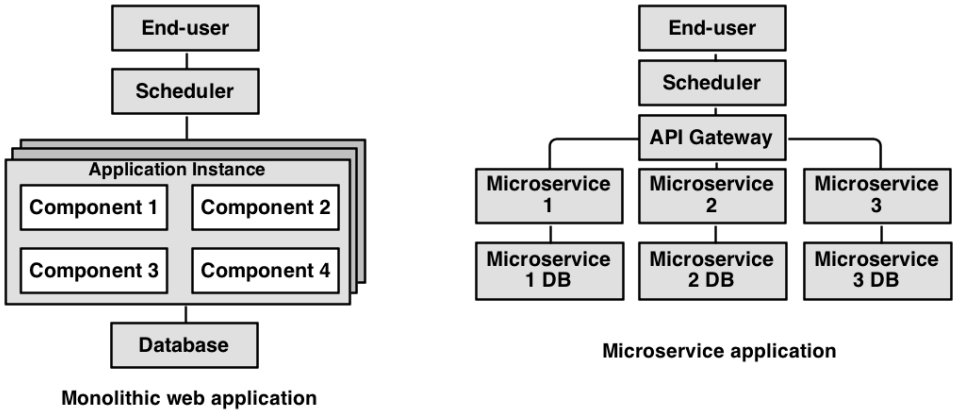

Figure 2.4: Differences between monolithic and microservice architectural styles

Amazon decided to rent out its computational resources when they were not in use, thus launching Amazon Web Services (AWS) (Rhoton and Haukioja, 2011). AWS was the first large-scale and commercially successful application for the provision of distributed resources that enabled the adoption of a microservice architecture.

Figure 2.4 shows the differences between three-tier architecture (also called monolith) (Figure 2.4, left) and microservice architecture (Figure 2.4, right). The application server in the monolith applications often works as one process and handles business logic, Hypertext Transfer Protocol (HTTP) requests from the user, communicates with the database and generates some output, for example, Hypertext Markup Language (HTML) or JavaScript Object Notation (JSON) responses. To handle an increasing load, the system administrator creates duplicates of the application server process. Such an approach imposes high overheads because process duplication also involves pieces of code that are not highly loaded. For example, if the hardware is not able to handle a large number of requests to the Component 1, the whole application instance should be duplicated, while in microservice architecture, it is possible to duplicate only the microservice that implements the required functionality (Figure 2.4). Changes to code require the redeployment of all of the application server processes, which may be difficult.

Lewis and Fowler (2014) mention that it is generally difficult to maintain a good modular structure when developing large applications. Therefore, companies have to divide a large application server physically into smaller modules and develop them separately. Such an approach also facilitates companies to outsource the development and maintenance of some modules while keeping the service contract untouched. Such an approach was introduced by Jeff Bezos in the early 2000s and called "You build it, you run it" (Vogels, 2006). All modules are hosted and deployed separately and can be adjusted in an agile manner to conform to a changing load. This is one of the main reasons for large companies to adopt a microservice architecture (Mauro Tony, 2015).

The fine-grained architecture of microservice systems has also been influenced by the agile development paradigm (Beck et al., 2001), and the microservice architecture follows agile

principles, such as customer satisfaction, adaptation to changing requirements, and close relationships between business and software engineers (Zimmermann, 2017).

2.2 Software testing techniques

Software quality assurance and testing focus on understanding the quality of the software under specific requirements (Kaner, 2006). Software quality is generally difficult to define, and there are several different definitions for it (Blaine and Cleland-Huang, 2008; Kitchenham and Pfleeger, 1996; Sommerville, 2004). In this study, we understand software quality as the capability of a software product to satisfy stated and implied needs when used under specified conditions (ISO/IEC, 2005). Software quality has two aspects: external and internal software quality. External software quality is the ability to demonstrate the intended behavior, and internal software quality mostly targets static parameters, such as architecture and structure (ISO/IEC, 2005).

Web services introduce new challenges for stakeholders involved in test activities, i.e. developers, providers, integrators, certifiers, and end-users (Canfora and Di Penta, 2009). These challenges are linked with the architecture, operational environment, and increasing number of customers. Unlike desktop applications, web services, including microservice-based systems, work in a special environment, and their quality assurance may involve more activities than that of stand-alone software. Cloud-based applications face more challenges in testing, including an on-demand test environment, scalability, and performance testing, testing security and measurement in clouds, integration testing in clouds, on-demand testing issues and challenges, and regression testing issues and challenges (Zech et al., 2012). These issues are mostly linked with the nature of the clouds: cloud-oriented applications are difficult to deploy and test locally, and the quality of the cloud infrastructure should be defined by the cloud provider in a service-level-agreement (Gao et al., 2011).

Generally, software testing methods may be divided into two categories: black box testing and white box testing (Kasurinen et al., 2009; Kit and Finzi, 1995). The key difference between these techniques is the amount of information of the system structure. This means that black box testing techniques do not use the information about the system's internal data processing and use only information such as system specification, service contract, and a common sense. In contrast, white box testing techniques can access the internal logic of the application to find comparable test cases, for example, boundary values linked with the business logic of the developed application. Black box testing and white box testing may be associated with external and internal software quality, respectively. In real application testing, both categories are used. The ISO/IEC 25000 (ISO/IEC, 2005), ISO/IEC 25010 (ISO 25010, 2011), ISO/IEC 29119 (ISO 29119, 2013) and IEEE 1012 (IEEE, 2012) standards describe different quality and software testing aspects, but standards dedicated to microservice testing do not exist. Therefore, to derive the microservice testing methodology, we need to analyze different distributed computing testing approaches and their similarities with the microservice architecture, including their advantages and disadvantages.

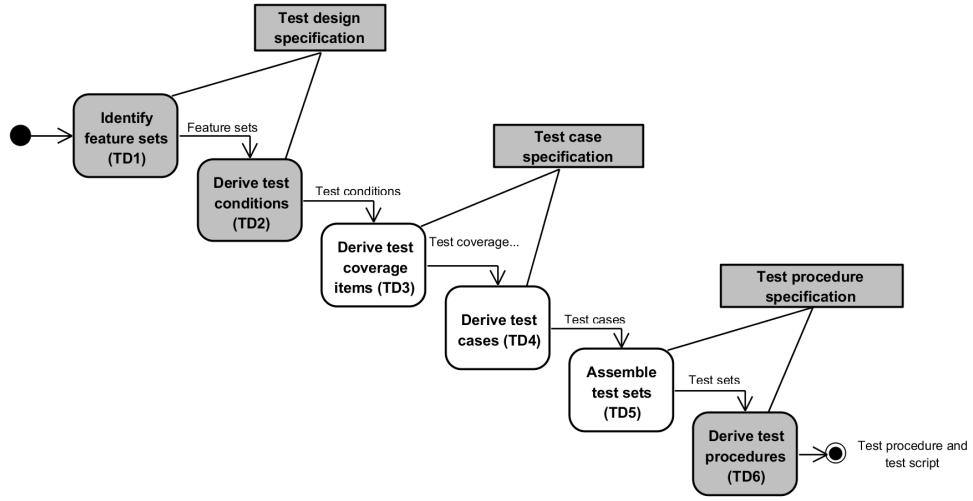

Figure 2.5: Test design and implementation process (modified from ISO/IEC 29119-4)

2.2.1 Software testing standards

Software testing and quality assurance processes may be defined and explained differently. For example, Heiser describes the software development cycle as a set of the following steps: requirement specification, preliminary design, detailed design, coding, unit testing, integration testing, and system testing (Heiser, 1997). This process is very general and ignores challenges linked with distributed systems, deployment, and a virtual or cloud environment. The ISO/IEC 29119-2 standard describes a general software testing process that can be applied to the wider range of software. This process consists of six steps: identify feature sets (TD1), derive test conditions (TD2), derive test coverage items (TD3), derive test cases (TD4), assemble test sets (TD5) and derive test procedures (TD6) (Figure 2.5).

The ISO/IEC 29119 (ISO 29119, 2013) standard describes different test levels and different test types. As an example, in this study, we observe the component, integration, and system levels, and performance, security, and functional testing types. To test microservice systems, we use the information described in ISO/IEC 29119 as a starting point, but we also follow microservice features described by Fowler and Lewis. This study concentrates on TD3, TD4. and TD5 because other testing process steps should be defined according to the domain and application features.

To apply existing testing standards to the testing of microservice applications, we need to analyze test levels and types and then choose examples of levels and types that explain microservice testing. For the chosen levels and types, we pick possible test techniques, that can be implemented in the microservice testing service. In our study, we selected the component, integration, and system levels as test level examples for microservice testing (Figure 2.6). Then, we implemented the selected test levels with corresponding features

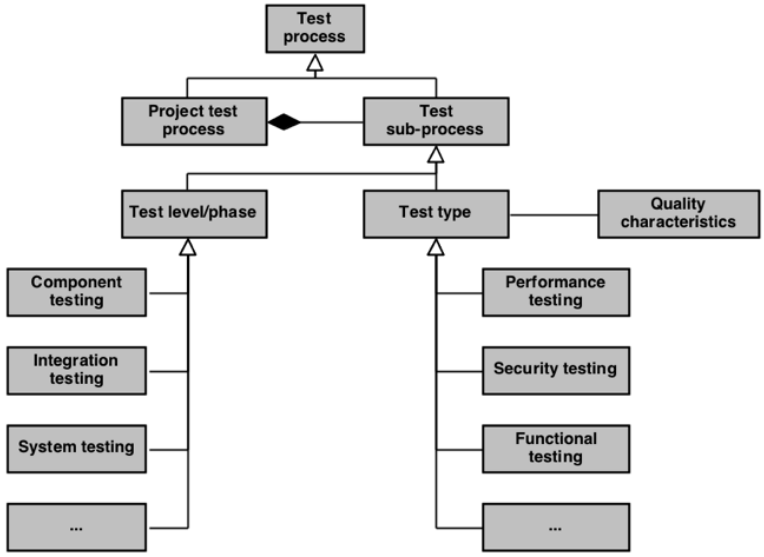

Figure 2.6: Test levels and types (ISO/IEC 29119-4 software testing standard, modified)

within the microservice testing context. ISO/IEC 29119 (ISO 29119, 2013) describes more test levels, but the described test levels were chosen to illustrate the applicability of the microservice testing service to the microservice testing. The quality characteristics of microservice applications and individual microservices can be derived from the requirements and functions of the system. The ISO/IEC Service Quality Requirements and Evaluation (SQuaRE) standard (ISO/IEC, 2005) lists eight quality characteristics and divides them into sub-characteristics. The quality characteristics and sub-characteristics were used to select examples of microservice test types and further microservice test techniques. As examples, we selected performance testing, security testing, and functional testing.

2.2.2 Distributed systems testing

Microservice architecture shares similarities with multi-agent and actor approaches. The multi-agent system is the system of multiple communicating intelligent agents (Jennings, 2000). Wooldridge (1997) defines an agent as an encapsulated computer system that is situated in some environment and that is capable of autonomous action in that environment in order to meet its design objectives, but in practice, the intelligent agent is understood as a programmatic entity that performs processing on a local or remote computational node. On the other hand, the actor approach considers distributed computing as a set of primitives called actors that can make local decisions, send and receive messages from other actors, and create new actors if needed (Tasharofi et al., 2012).

Even though the approaches above share many similarities with microservice architecture, there are still many differences. For example, actors can create new actors to fulfill their responsibilities, but in the case of a microservice, only the infrastructure can create new microservices. Agents usually work at a lower level of abstraction than microservices. In addition, microservices are virtual containers at the physical level, and an internal container infrastructure should also be tested. Multi-agent systems testing implies that the application should be tested at different levels – the component, agent, integration, and multi-agent or system levels (Nguyen et al., 2011). Dividing the testing process into levels makes the testing process easier because it is divided into several independent steps. Each microservice may be considered as a dedicated software entity and tested respectively. In comparison with actors, microservice architecture has more differences, as well as in their testing. Actors can create other actors. In addition, actors usually maintain their own state, while in a microservice architecture statefulness is usually considered as a bad practice (Tasharofi et al., 2012). Differences are provoked by the differences in the applications of those approaches, and therefore, actor systems are usually tested using workflow testing techniques (Altintas et al., 2004).

2.3 Summary

Chapter 2 describes the background and scope of this study. It includes an overview of the existing testing approaches and standards, as well as several distributed computing approaches and their differences with the microservice architecture. Microservice architecture shares similarities with actor-oriented architectures, service-oriented architectures, and agent programming, but it is not implemented at a higher level of abstraction. To formulate the approach to microservice testing, we decided to combine different sources and demonstrate the implementation of several test levels and types as an example of the microservice testing.

Chapter III

Research problem, methodology and process

In this chapter, we describe the research problem we want to solve, establish the research question, and review possible research methods and choose the most appropriate one. Then, we describe the application of the chosen research method. The selection of the research method is based on the research question. We use the description of the microservice architecture by Lewis and Fowler (2014) as a basis to establish the research question. Then, we use the research method taxonomy described by Järvinen (2012) to choose the appropriate research method.

3.1 Research problem and its shaping

The demand for software testing grows from year to year, and software faults generate remarkable losses for companies (Sogeti, 2016), and it is clear that software testing increases the resulting quality of the software product (Tassey, 2002). On the other hand, software engineers may choose the microservice architecture for high load web services, but there is no standard for such an implementation. Therefore, different microservice systems may be implemented in a different way, including protocols, formats, and infrastructure automation tools. This also leads to the conclusion that each company performs microservice testing in its own way, depending on the infrastructure, deployment mechanism, and other factors, including the software development and maintenance budget. It also means that microservices are usually not publicly available. This concerns also microservice testing tools and explains why publicly available microservice testing service is a novel artifact and needs to be implemented from scratch based on empirical observations. To select the proper research method, we first need to understand the research problem and the domain. Microservice architecture is a relatively new concept, and there are few studies concerning microservice testing. Our research problem is to develop a microservice testing methodology, and an artifact, it's novel software implementation. This research problem leads to a research question that can be formulated as follows: Is it possible to create a system that can be used as a general tool for microservice testing?

3.2 Research methods

To choose the proper research method, we need to analyze the research problem and the corresponding research methods. Figure 3.1 shows the taxonomy of the research methods, described by Järvinen (2012). This taxonomy distinguishes mathematical methods from other methods because they work with the formal languages, for example, algebraic units and other abstract entities, that are not directly linked with any real objects. Then, the taxonomy distinguishes methods by the research question of the method. Approaches studying reality contain two classes of research methods: studies that aim to stress reality, and studies that are stressing the utility of the artifacts – something made by human beings. Research stressing what is reality also have two subclasses – conceptual-analytical approaches and approaches for empirical studies, which include theory-testing approaches and theory-creating approaches. Conceptual-analytical studies deal with basic terms and definitions behind theories; theory-testing approaches deal mostly with experiments that can also be called field tests; theory-creating approaches include case studies, grounded theory, etc. These approaches aim to create a theory based on empirical observations. Artifact-centered studies are divided into two classes: artifact-building approaches and artifact-evaluation approaches, but proper design science research includes both (Hevner et al., 2004). Design science research requires engineers and stakeholders to impose specific requirements on the artifact, and the artifact should be evaluated to determine whether it fulfills the imposed requirements or whether it solves the original problem. Design science research is based on preliminary empirical studies to determine the requirements and identify the problem, but the design science research process concentrates on artifact creation and evaluation (March and Smith, 1995). This study focuses on the creation and evaluation of a novel artifact, and therefore, we use design science as the research method.

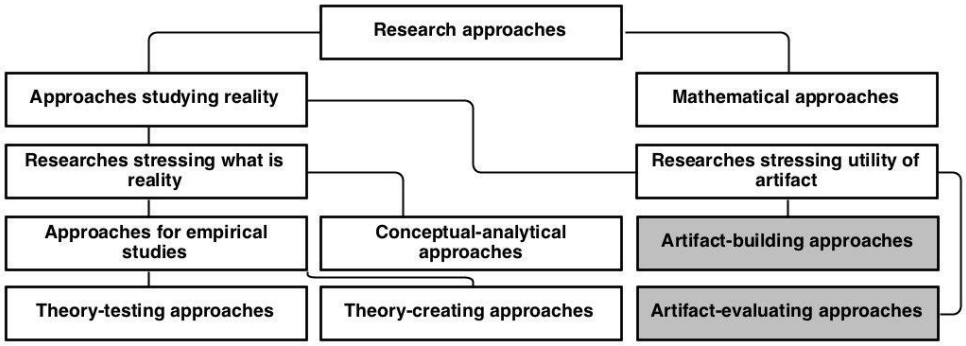

Figure 3.1: Järvinen's taxonomy of research methods (modified)

3.3 Research process

In this study, we follow the regular design science process, described in Figure 3.2. This process, described by Peffers et al. (2007), implies, that the research is divided into six

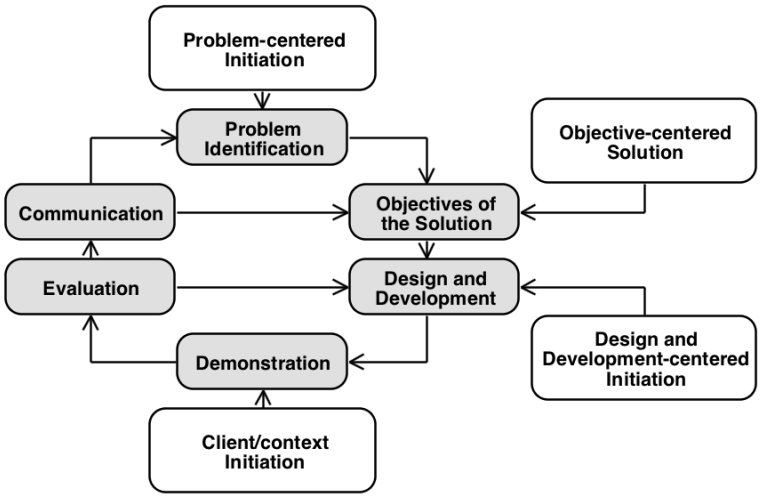

Figure 3.2: Design science research method process model by Peffers et al. (modified)

steps: problem identification and motivation, objectives and solution definition, design and development, demonstration, evaluation, and communication. The design science process may be initiated from different entry points, for example, it might be initiated from a client or problem context. In this study, we decided to choose problem-centered initiation and started from the analysis of different testing approaches. Then, we defined the objectives and a possible solution as a set of requirements for a microservice testing methodology. The design and development phase consisted of the methodology description, architecture derivation, and following software implementation. The implemented testing service was published on an open source website and evaluated using example microservices.

3.4 Related publications

The microservice architecture was described by Lewis and Fowler (2014), and it means that this approach is relatively novel. A number of studies investigate microservice architecture applications, but only a few deals with microservice applications testing. The latter are mostly theoretical, which is why it is difficult to adopt those testing techniques in practice. For example, Ford (2015) describes basic ideas regarding microservice systems development, monitoring, and testing approaches, but this study lacks practical implementation.

Savchenko and Radchenko (2014) describe a prototype for distributed application development. This platform was implemented as an infrastructure to support distributed applications that consist of small independent components, implemented around business capabilities. Those components can communicate only using a built-in message passing

interface, and therefore, fulfill the external contract. This platform implements microservice architecture logic on the low level of abstraction and is intended for use in business capabilities automation.

A general process of software testing can be developed based on ISO/IEC and IEEE standards. The IEEE Standard for System and Software Verification and Validation (IEEE, 2012) describes the general process of verification and validation for a wide range of software products. Verification includes activities associated with general quality assurance – for example, inspection, code walkthrough, and review in design analysis, specification analysis, etc. Validation usually aims to check whether the system meets the imposed requirements or solves the original real-world problem. The IEEE 1012 V&V standard defines software validation as a process of a component or system testing in order to check, does the software meet original requirements (Geraci et al., 1991). Validation consists of several levels: component testing, integration testing, usability testing, functional testing, system testing, and acceptance testing. In practice, validation is usually known as dynamic testing, while static testing is associated with verification, so in this study, we mostly focus on the validation process.

3.5 Summary

This chapter described the research problem, research question, and research process that was used in this study. Table 3.1 and Figure 3.3 summarize the research phases of the whole study.

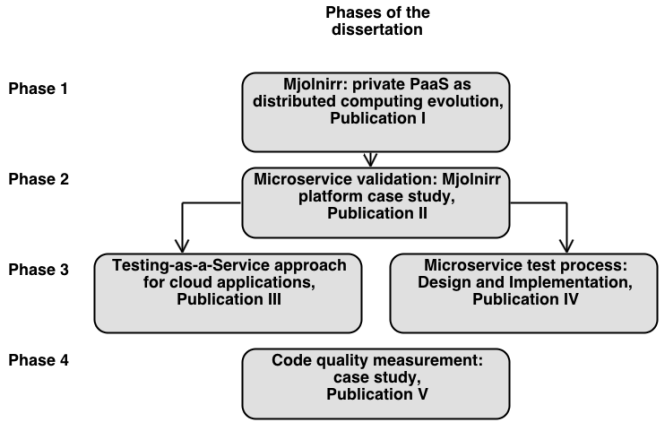

Figure 3.3: Research phases and publications

Table 3.1: The research phases

Phase	**Phase 1**	**Phase 2**	**Phase 3**	**Phase 4**
Research question	How to create a flexible solution for a business infrastructure?	How to test microservice systems?	How to build microservice testing service and evaluate it?	How to build an early-warning system for a wide range of software?
Research method	Design science	Design science	Design science	Design science
Reporting	Publication I	Publication II	Publication III, Publication IV	Publication V

Chapter IV

Overview of the publications

This chapter presents an overview of the most important results expressed in the publications of this dissertation. Five publications, attached as an appendix, contain the results in detail. All publications have been published separately in peer-reviewed scientific conferences and a journal. This chapter briefly discusses each of the publications, including their research objectives, main results, and their relation to the whole study.

4.1 Publication I: Mjolnirr: A Hybrid Approach to Distributed Computing. Architecture and Implementation

4.1.1 Research objectives

The objective of this study was to design and implement a private PaaS solution called Mjolnirr. This solution aimed at business infrastructure automation using a modular approach. The modular approach reduces the maintenance and development costs, and dedicated modules can be reused in other applications. The approach we investigated and described within the development of the Mjolnirr platform fits the microservice definition because the developed platform operates fine-grained and isolated components that communicate using a message bus. Microservice systems are based on a containerization technology, while components in Mjolnirr are based on a Java virtual machine. Mjolnirr platform development highlighted possible problems in the quality assurance of microservice-like systems.

4.1.2 Results

In this study, we investigated possible problems in business automation and existing solutions and offered a new solution to the problems. The study also presents an implementation of the Mjolnirr platform that meets business automation needs. The main features of the described platform are an advanced messaging system and support of distributed computing at the level of architecture.

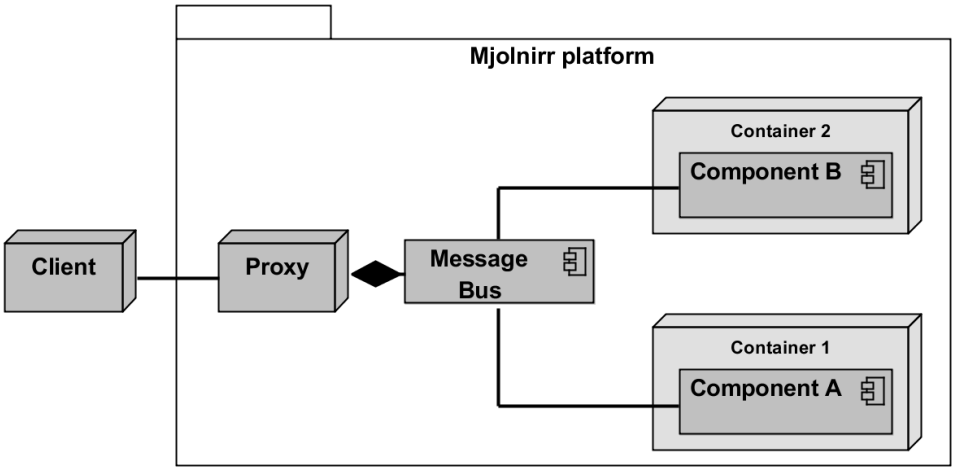

Figure 4.1: Mjolnirr platform architecture

The Mjolnirr platform is intended to meet the following requirements:

- *Costs* should be reduced by using popular and well-maintained open source projects as well as widely-used programming languages. In addition, the Mjolnirr platform has the ability to work not only on dedicated servers but on unallocated resources on personal computers within an organization. The use of idle resources may provide savings in server hardware.
- *Application development* should be facilitated with a popular language and integrated software development kit (SDK).
- *New resources and legacy applications* should be integrated with the help of built-in modular application architecture.

The Mjolnirr platform architecture (Figure 4.1) was developed to meet the requirements listed above. It consists of four basic component classes: *Proxy*, *Container*, *Component*, and *Client*. The Proxy acts as a gateway of the platform. It provides and controls access to the internal resources, manages the communication between applications, maintains the message bus, and hosts system services: a user authentication module, a shared database access module, a distributed file system module, etc. The Container is the entity that is responsible for physical resource allocation and applications hosting. A Mjolnirr installation may have several containers on different computational nodes, for example, a server node or personal computer. The Container provides a virtualized infrastructure for the applications and abstracts the custom applications in real hardware. It is important to note that the term "container" in the context of the Mjolnirr platform is not the same as, for example, the Docker container because it does not provide isolated namespace

and operating system capabilities. Mjolnirr containers rely on Java Virtual Machine to run custom Java applications within the dedicated network. The Component is a custom application developed by third-party developers. The Component provides business logic and is usually built around a single business capability. Optionally, the Component may have an HTTP-based interface accessible to clients. The Client is an external application that accesses the applications through the proxy.

4.1.3 Relation to the whole

This study was the starting point in understanding the practical needs of microservice systems research. The Mjolnirr platform is not a microservice platform implementation, but it partially follows the definition by Lewis and Fowler (2014) and can be used to evaluate microservice testing techniques. The Mjolnirr platform was implemented in accordance with a microservice paradigm and using this example, we found that such systems are difficult to test. During this study, only a few microservice infrastructure implementations were available, and that is why we used the Mjolnirr platform as an example of the microservice platform in later studies.

4.2 Publication II: Microservices validation: Mjolnirr platform case study

4.2.1 Research objectives

In the previous study, we studied the deployment and maintenance flow of systems that consists of a set of small independent components. In addition, during the year 2014, Lewis and Fowler published their research about microservice architecture. In our study, we focused on possible techniques for microservice testing. We decided to analyze the testing techniques for different distributed computing approaches and describe a possible methodology of microservice testing.

4.2.2 Results

This study presents the approach to the microservice testing and uses the ISO/IEC 29119 (ISO 29119, 2013) and ISO 25010 (ISO 25010, 2011) standards as starting points. We modified the generic test design and implementation process defined in ISO/IEC 29119 and described the microservice testing methodology. This methodology considers several features of microservices that are not specific to other software development approaches. In addition, we described the possible implementation of the testing system that was based on the Mjolnirr platform, portrayed in Publication I.

To depict the microservice testing methodology, we chose component, integration, and system testing levels as an illustrative example. To perform microservice testing at those levels, we need to analyze those levels and highlight special features in the microservice testing context. Then, we use the highlighted features to find the most appropriate test techniques.

1. *Microservice component testing.* Component testing of microservices consists of independent testing of the individual microservices to meet functional requirements. This type of testing is based on the formal description of the functional requirements imposed on each microservice, including requirements of the input and output data. Functional component testing considers a microservice as an isolated component, and testing can be conducted locally. The development of a software system in accordance with the microservice style means developing individual and independent microservices that interact exclusively using open protocols, as follows from the microservice definition by Lewis and Fowler (2014). Therefore, each microservice is an independent piece of software that needs to be developed and tested independently of the other microservices during the component testing. Testing a single service may be considered equivalent to component testing (Canfora and Di Penta, 2009). On the other hand, the microservice can be a complex software system which consists of several software components (local storage, web server, etc.) encapsulated in a container. Also in the case of third-party software, components of such an ensemble inside a container must be validated. The microservice inside the container is available only through its external interface. Hence, we can consider the microservice as a black box (Canfora and Di Penta, 2009). Therefore, in the component testing of microservices, it is necessary to test the compliance of the interface of the microservice with specifications, and the entire microservice should be tested using its external interface.

2. *Microservice integration testing.* As expressed by Lewis and Fowler (2014), a microservice system is built around business requirements and consists of many different microservices. The microservice system can be dynamically changed using new instances to meet varying conditions, for example, service unavailability. In such a scenario, it becomes crucial to test services for interoperability, i.e. to test service integration (Canfora and Di Penta, 2009). The integration testing involves testing of different types of communications between the microservices. It includes, for example, the testing of communication protocols and formats, the resolution of deadlocks, shared resource usage and messaging sequences (Jüttner et al., 1995). To find corresponding test techniques, we need to track the messages transmitted between the microservices and build a messaging graph. For this purpose, we need to know the origin and destination of the messages. A microservice interface definition offers this information as the test basis. With the interface definition, we can track the correct and incorrect messages to specific microservices. Further, we can filter and generate messages to predict the behavior of the microservice system in the production environment. Integration testing can also be used to audit the microservice system.

3. *System testing.* System testing means the testing of the whole microservice system regardless of its internal structure (Geraci et al., 1991). It can be, for example, a web-service and can thus be tested with web service test techniques. For example, it is possible to perform a generic application program interface (API) testing at the system level as well as at the component level. However, there is a difference at the component level: individual components are not reachable outside of the microservice environment, whereas the whole system is. Therefore, at this level, we should ensure not only functional suitability but also, for example, the security of

the entire system and its external interface. During system testing, the microservice system can be considered as a black box (Geraci et al., 1991). Therefore, test techniques which can be applied at this level do not consider the internal structure of the system. Microservice system testing can be widely covered by generic test techniques, including web-service testing.

To choose the appropriate test techniques, we first choose the quality characteristics and sub-characteristics. In this study, we selected security, performance, and functional suitability as our object quality characteristics and the equivalent testing types: security, performance, and functional suitability testing (Figure 2.6). We applied the mapping between the ISO/IEC 25010 (ISO 25010, 2011) quality characteristics and sub-characteristics, and the test design techniques (ISO 29119, 2013). By applying this mapping, we can find appropriate test techniques for the quality characteristics and sub-characteristics (Table 4.1). The mapping is an example and needs to be modified due to the special features of the applications, and also the microservice architectural style may entail extra quality requirements. Table 4.1 provides an example of the microservice quality characteristics, sub-characteristics and related test techniques in connection with the test types, security testing, performance testing, and functional testing.

Table 4.1: Examples of quality characteristics and sub-characteristics mapped to test design techniques, according to the ISO/IEC 29119-4 software testing standard (modified)

Quality characteristic	Sub-characteristics	Test design techniques
Security	Confidentiality Integrity Non-repudiation Accountability Authenticity	Penetration testing Privacy testing Security auditing Vulnerability scanning
Performance	Time behavior Resource utilization Capacity	Performance testing Load testing Stress testing Endurance testing Capacity testing Memory management testing
Functional suitability	Functional completeness Functional correctness Functional appropriateness	Boundary value analysis Equivalence partitioning Random testing Scenario testing Error guessing Statement testing Branch testing Decision testing Data flow testing

4.2.3 Relation to the whole

This study was the initial part of our microservice testing research. In this paper, we formulated the basic rules for microservice and microservice systems testing. We presented the methodology that can be applied to Mjolnirr testing – the example of the microservice platform. In this study, we also reformulated our research question into: "How to test microservice systems?". This change was motivated by a rising interest in the idea of microservices, but it was not clear at that moment how to test microservice systems.

4.3 Publication III: Testing-as-a-Service Approach for Cloud Applications

4.3.1 Research objectives

This study aimed at the practical implementation of the microservice testing service prototype. In addition, we applied the term 'Testing-as-a-Service' to such an approach for the first time.

4.3.2 Results

This study presented an example implementation of the microservice testing service. The prototype had limited applicability and implemented only three test techniques: REST API testing, performance testing for web services, and UI testing for single page web applications. The prototype was not yet evaluated in this study, but we explained the workflow when applying it to microservice testing.

In building the implementation, testing activities were selected from the test level and test type examples. The microservice testing service example includes the following activities:

1. component testing of the microservice source code;
2. component testing by microservice self-testing, where 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 functional suitability under the load;
6. integration testing of functional suitability to test the microservice's interactions.

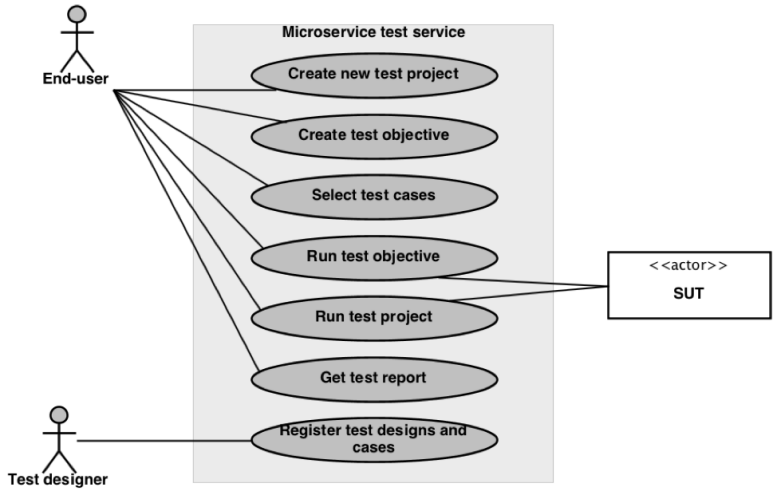

Figure 4.2: Microservice testing service use case

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

The microservice testing service was implemented as a web application. It enables end-users to run their preconfigured tests automatically or with a suitable trigger. The system allows to create and configure test projects, set test objectives and install new test designs and cases. Test designs and cases may be associated with an application type – a predefined set of techniques most suitable for the application. For example, a web service, implemented using Ruby on Rails (2018), should be tested with one set of test designs and cases, while a command-line application for infrastructure usage should be tested with a different set of test designs and cases. The microservice testing service offers test designs and cases by application type if the test cases have been implemented earlier. Figure 4.2 describes the main roles of the users.

1. The end-user uses the service to test the SUT;
2. The test designer creates and registers test designs and cases in the testing service. Test designs and test cases shall meet the requirements of the testing service to be properly registered;
3. SUT is a microservice system.

The roles are interpreted as the actors of the service. The end-user can set the application's test level and type, select test designs, and test cases or configure new test cases to achieve the objectives, run separate tests or launch scripted test runs, and obtain test

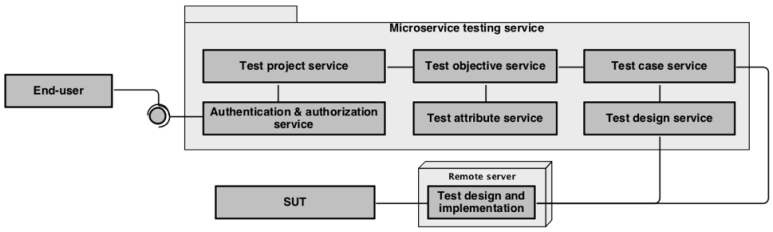

Figure 4.3: Microservice testing service architecture

reports. The reports contain detailed information of passed and failed test executions and reasons for them.

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. To solve this problem, the microservice testing service uses external containers that host different test designs and their implementations. A microservice testing service provides an HTTP Application Programming Interface (API) for easy integration of third-party test designs and cases. The testing service is implemented in accordance with the microservice architecture to support and allow the extension. Figure 4.3 depicts the proposed architecture.

The authentication and authorization service identifies users, their roles, and permissions using their credentials when the end-user enters the testing service. Rights are given according to the permissions. 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 which will be passed to underlying test objectives and cases to facilitate the test configuration.

The test attribute service provides a list of associated test designs and cases according to application attributes. An application attribute 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 also 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. The process of the test project creation and run may be described as follows:

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 test process, the end-user provides the final testing parameters, such as the microservice system location or programming language. The service executes the test cases.

The microservice testing service prototype has an HTTP API interface for external communication and a visual interface. The visual interface is intended for the configuration of test projects, objectives, test designs, test cases, names, parameters, the test execution order, etc. in the scope of the SUT. Using the HTTP API interface, the end-user can run tests to reach the test objectives. HTTP API also returns the test execution status, error log and other test-related information. HTTP API can be initiated using the visual interface. To create the test objective, the end-user specifies the name of the objective, describes it and chooses one or more available application attributes. 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. By 'static', we understand parameters that do not change from run to run, such as the 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 (Listing 4.1).

Listing 4.1: 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"
 }
}
```

The HTTP request provides the necessary information for the test execution linked with the objectives except the 'static' parameters. Listing 4.1 displays the source code repository location, programming language, and location of the SUT.

4.3.3 Relation to the whole

This study illustrates the implementation of the microservice testing system. The implementation is based on the methodology, described in Publication II. It is generally difficult to perform a quantitative evaluation for this kind of a system. Microservice architecture is not as widely used as, for example, REST-based monolith web services because it

requires high effort for the initial implementation and it yields benefits in potentially high-loaded applications. In addition, microservices are mostly used in large companies within their proprietary infrastructure, and it makes the analysis of microservice systems more difficult. Publication III mostly focused on the architecture and implementation of the microservice testing service, whereas evaluation is conducted in later studies.

4.4 Publication IV: Microservice Test Process: Design and Implementation

4.4.1 Research objectives

In this study, we described the methodology of microservice testing. The methodology uses ISO/IEC/IEEE 29119 and ISO 25010 as starting points, as well as the existing testing techniques used with other distributed computing approaches. This study combines all of the results from the previous publications and evaluates the developed system.

4.4.2 Results

This study summarizes the studies of microservice testing. It indicates that the testing of a single microservice has few differences compared to the testing of any other web-service or SOA entity. A microservice system consists of a set of independent services that are deployed independently using an automated infrastructure. This affects features in the testing process. As a solution, we offer an example of architecture and implementation that may be included in the existing continuous integration infrastructure. The architecture is based on the microservice architecture because the microservice system may contain microservices implemented using different operating systems, frameworks, and programming languages. The system that we described in this study meets those requirements and offers a plug-in structure for different test techniques. The developed system was evaluated using an open source microservice with a flow that illustrates the integration of the microservice testing service with a continuous integration system.

The design science process includes the evaluation of the developed artifact. To evaluate the microservice testing service, we decided to use an open source microservice, implement a continuous integration process, and integrate the microservice testing service to this process. In addition, it is generally difficult to perform quantitative measurements for such a system. Therefore, we decided to implement a scenario that illustrates the applicability of the artifact. To estimate the applicability of the microservice testing service, we employed a Docker container which contains the microservice testing service implementation. A typical example of the evaluation was selected because the number of different test scenarios is unlimited. The evaluation was conducted on an example of an open source microservice. To demonstrate the typical workflow used in continuous integration with real-world microservices, we implemented the scenario Figure 4.4 shows. This scenario includes the microservice component, system, and integration testing, concurrently following the guidelines of ISO/IEC 29119 standards (ISO 29119, 2013) and implementing the collected observations. The scenario was implemented using two simple Ruby scripts that called command line interface and executed described commands. The success of the operation was determined using exit codes: 0 for a successful test and

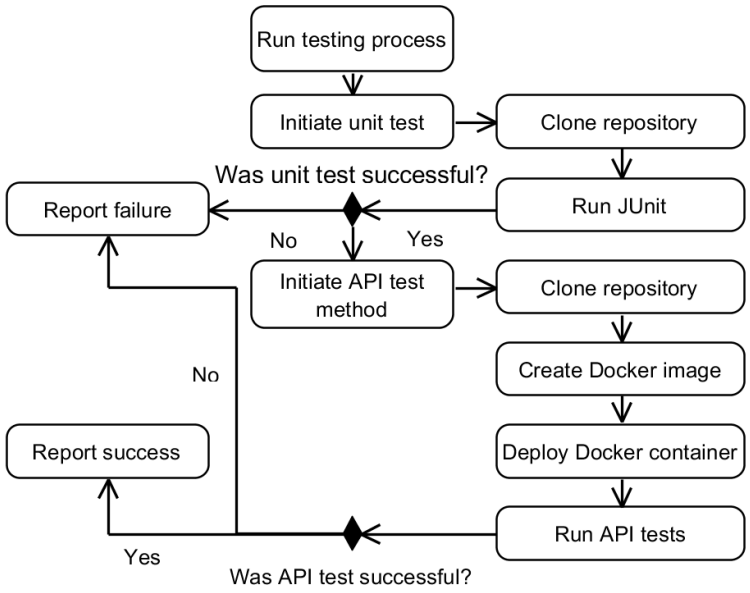

Figure 4.4: Workflow of the evaluation example

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. The 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. The integration example we provided in Publication IV showed a possible way of including the microservice testing service in existing infrastructure.

4.4.3 Relation to the whole

This study was a summary of microservice testing studies and combines previous results with system implementation and evaluation. We provided an overview of microservice testing service design, implementation, and evaluation according to the Design Science principle of Peffers et al. (2007). Publication IV answers the research question: "How to build a microservice testing service and evaluate it?". As an evaluation, we decided to demonstrate the applicability of the microservice testing service and illustrate it using the integration of this service into an existing continuous integration system. The evaluation showed that the developed microservice testing service can be used to organize continuous integration and delivery for a wider range of software without major changes. It is possible because of the distributed nature of the microservice testing service. As long as test designs and implementations are hosted on remote nodes, it is possible to apply the

system to different heterogeneous systems or mobile applications.

4.5 Publication V: Code Quality Measurement: Case Study

4.5.1 Research objectives

In this case study, we analyzed the sources of maintenance costs and ways to reduce those costs. In addition, this study presents an implementation of a system, that follows the principle of continuous testing to pinpoint possible sources of maintenance expenses as early as possible. This study is an example of the application of the microservice testing service to a wider range of software.

4.5.2 Results

This study introduces an overview of different maintenance metrics that could reduce the effort required in software maintenance. We used this overview to further develop the architecture of a system that is able to collect a large amount of data from the software and from all stages of the development process. The architecture of the microservice testing service was developed to be modular and extendable. We used the developed architecture to implement the continuous testing solution in the Maintain project. Maintain project is a Business Finland funded research project to reduce maintenance costs (http://www2.it.lut.fi/projects/maintain/). It was motivated by the rising maintenance cost in modern software development: according to the Gartner report (Kyte, 2012), maintenance costs may reach up to 92% of the total project expenses. We used the developed microservice testing service architecture and integrated it with a continuous integration flow in the Maintain project.

The Maintain software architecture is based on the microservice testing service architecture, but it also introduces new entities – *Probes* and *Analysers* (Figure 4.5).

- *Probe* is a program that gathers measurement data from software (static or dynamic). Each probe has an associated analyzer that analyzes new data from the probe;
- *Raw Data Storage* is a data storage that stores the raw data from the probes. Each probe may generate outputs in different formats, which is why Raw Data Storage is based on a document-oriented database that allows maintaining schemaless storage;
- *Analyzer* is a sub-program that receives the associated probe output from the Raw Data Storage and creates a report, based on this data;
- *Report Storage* is a data storage that stores reports from analyzers;
- *Report Visualizer* is a component that creates a visual representation of the report.

Maintain software uses existing interfaces and entities to accumulate data from probes instead of test design and implementation which were used in the microservice testing

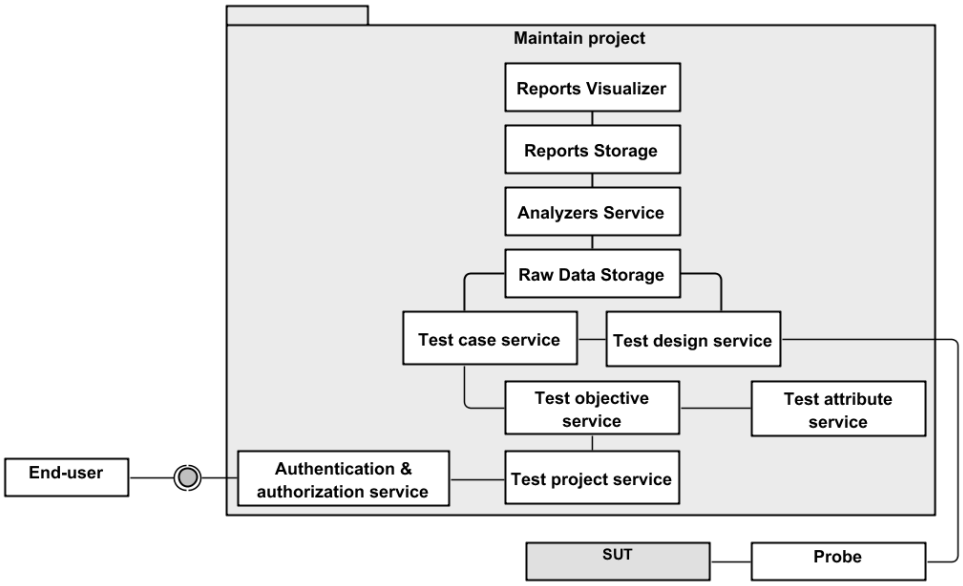

Figure 4.5: Maintain system architecture

service. In addition, we built an analyzing sub-system on top of the data gathering infrastructure provided by microservice testing service. Different probes may have different output formats – for example, a probe that calculates source code quality and a probe that monitors web page behavior and performance generate outputs in different formats.

The Maintain system was evaluated using a proprietary web application implemented with Ruby on Rails as a backend and CoffeeScript on top of React.JS as a frontend. The analyzed project is in the maintenance phase, and we decided to analyze historical data and compare Maintain system results with the feedback from the project manager, who managed the analyzed project. The application was used by five administrators and about 10000 users. Maintain system was deployed in Heroku cloud, while probes were running on a local PC. We gathered the code quality information for all previous code revisions to make the picture more consistent.

The Maintain project illustrates the general "health" of the analyzed application in the last code revision of the current working branch. Figure 4.6 shows the JavaScript (CoffeeScript) and HAML code quality. The project started as a pure backend solution, while frontend development started at the beginning of September 2016. A freelance designer was involved in the HAML development between November and December. As the graph shows, the HAML code quality decreased from September 2016 until December 2016, but after that it was stable. This behavior can be explained by the poor quality of the code that the freelancer produced and the deadline of the project, which was at the end of the year 2016. After the deadline, the active project development stopped. The project manager evaluated the results and stated that such an "early warning" system

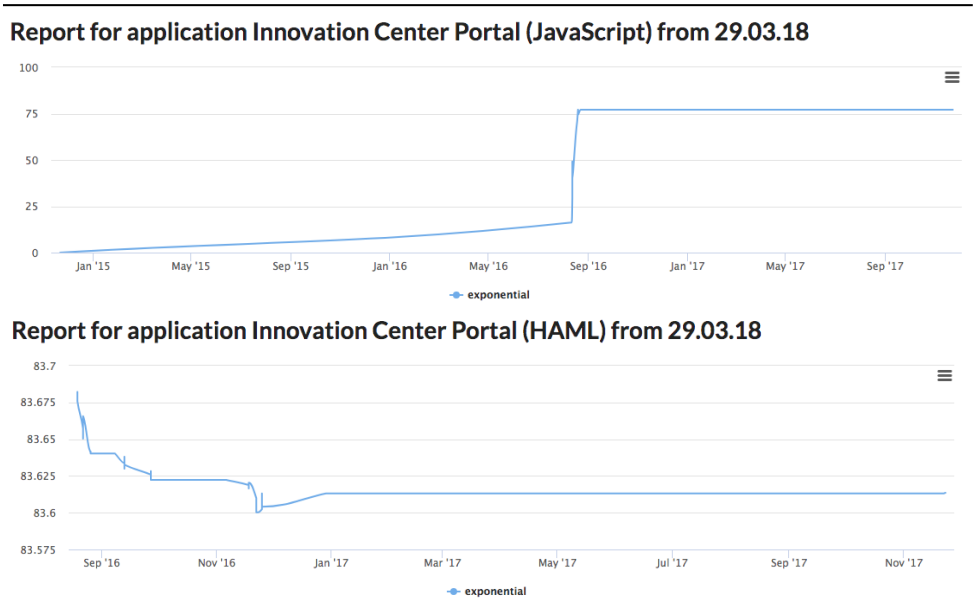

Figure 4.6: Project performance and code quality report

could notify the team about the possible problems in the frontend markup (HAML) and save some development resources. This evaluation shows that a microservice testing service can work together with continuous integration and continuous delivery systems and provide maintenance information on the SUT. Further pilot tests are ongoing with three operational systems. In the pilot tests, we gather historical information about old code revisions, analyze them within the Maintain project, and then conduct a set of interviews with the project managers involved in those projects. We use historical data to verify the reliability of the Maintain project analysis because the project manager can provide expert opinions on code quality issues and the problems it caused. This approach allows us to understand whether the analysis provided by the Maintain project confirms the subjective point of view of the project decision makers.

4.5.3 Relation to the whole

Maintain project evaluated the microservice testing service which was an integral part of the Maintain project architecture and implementation. Therefore, this study was also a pilot of the microservice testing service. The Maintain project employs the same ideas and architecture as a microservice testing service but to a wider range of different software. Combined with the analytic module, this provides real-time information about the code quality and software behavior.

Chapter V

Implications of the results

The microservice architecture is a relatively new approach to building highly-scalable applications to handle a massive external load. In this study, we understand s microservice as "SOA done right" (Lewis and Fowler, 2014): this means that one dedicated microservice meets requirements that are usually imposed on web services. This study focuses on the development, software implementation and evaluation of the microservice testing methodology.

When this study started, only a few studies had been conducted in the microservice field. The number of available publications was limited because different companies adopted a microservice architecture in different ways. Some companies reported their results in research papers, but their implementations were mostly proprietary and not available to the external researcher. Therefore, our study had a limited amount of information to start with. The microservice testing methodology was developed based on the existing software, service testing standards, and observations from existing solutions in the domain, for example, testing methodologies for actor-oriented systems, multi-agent systems, and web services. This study illustrates three examples of software quality characteristics and corresponding test types using three test levels. This illustration may be used to implement the continuous testing solution for microservice application in the real world. In addition, our study showed that a microservice testing service architecture has a modular structure and may be extended to meet a wider range of software. To demonstrate the applicability of the developed architecture, we decided to evaluate three different projects within the Maintain project piloting and compare the results of the Maintain project to the subjective feedback from the project manager.

As a first iteration of the evaluation process, we decided to analyze the historical data of the full-stack JavaScript application for seven weeks. We chose such a time range to make it easier to gather feedback from the project manager. On the first iteration of the analysis, we compared the code quality report directly to the feedback from the project manager (see Fig 5.1). Generally, results from the Maintain project show that code quality changes are linked with the project phase, but we also noticed several anomalies. To clarify those observations, we decided to concentrate on the dynamic of the code

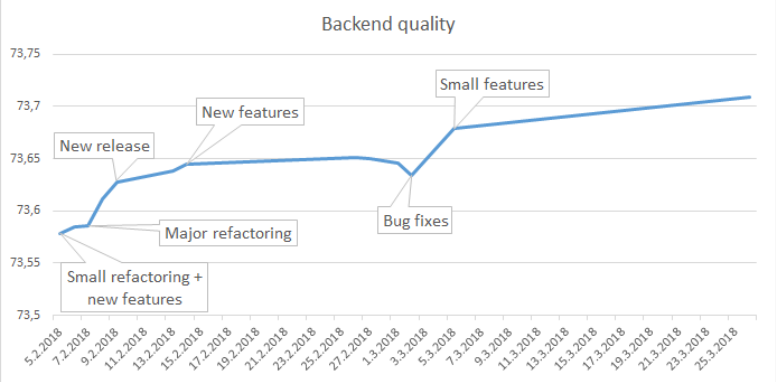

Figure 5.1: First step of the evaluation: raw data

quality change instead of its absolute value. This dynamic may be roughly described as a difference between the linear regression of the code quality measurements and the actual results. The outcome of this operation is presented on Fig. 5.2, and this led us to the conclusion that in order to create the early warning system for software code quality monitoring, we need to analyze code quality change dynamics.

To evaluate this theory, we decided to perform the same operations on the results from Publication V. Data was processed in the same manner as in the previous step and presented in Figure 5.3. This presentation highlights that all aspects of the analyzed application (markup, frontend, and backend) are linked to each other. Within this project, HAML markup was handled by a freelance designer, and the decline of the code quality for the whole project was initially provoked by the HAML part. As long as the HAML quality metric is based on the code smells, and JavaScript and Ruby metrics are based maintainability index, this link appears to be correct. In addition, the project manager also approved that freelance designer imposed several major problems that were later handled by the main team.

As the third step of evaluation, we decided to perform an analysis of the project that was not involved in previous studies. This time, the analysis took place over a longer period of time, but it was impossible to gather feedback from the project manager. Previous steps highlighted that the objective code quality change is linked with the development stage, so code quality should increase during refactoring or bug fixing and decrease during new features development. To check the observations within this project, we compared the code quality change with the data from the issue tracking system. To produce the metric that represents the development stage, we introduced a simple metric, based on issue type. For each day, the value is calculated as a sum of 1 (for bug) or -1 (for feature). Figure 5.4 compares this metric and the Maintain project report. The figure demonstrates that code quality have some correlation with the issue tracking system metric, but peaks are a little shifted. This shift may be explained by the missing data in the issue tracking system output: as a date, we used the creation date, so there was no guarantee that the bug was fixed the same day or week.

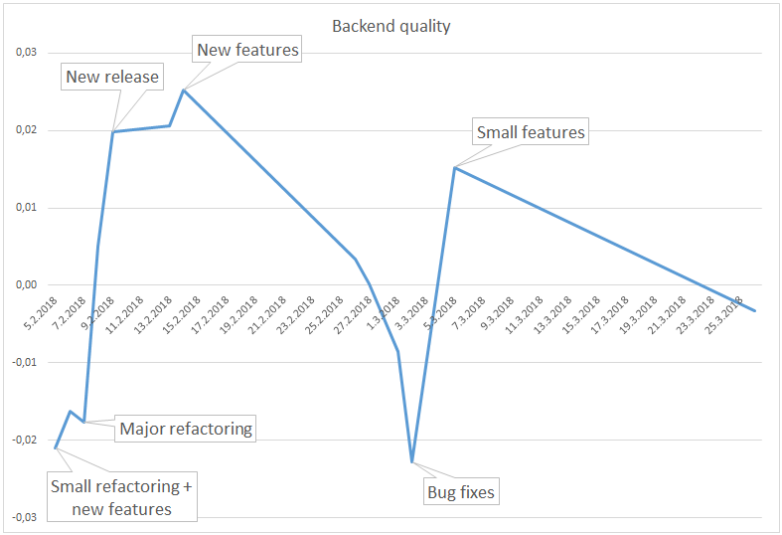

Figure 5.2: First step of the evaluation: normalized data

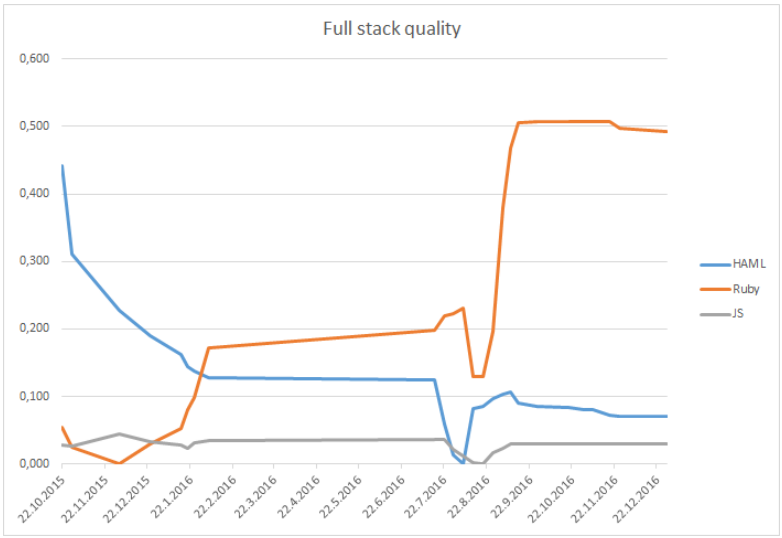

Figure 5.3: Second step of evaluation

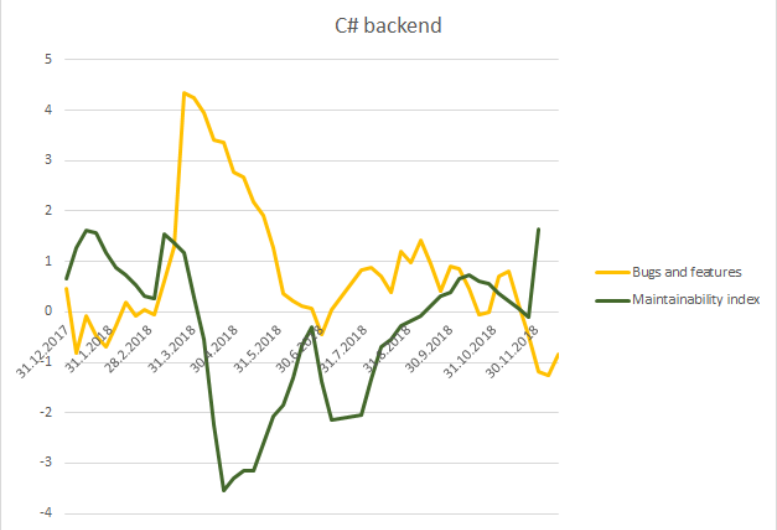

Figure 5.4: Third step of the evaluation

This piloting is an extended evaluation of the Maintain project within real organizations. As long as the Maintain project architecture is based on the microservice testing system architecture, we can consider the piloting as the evaluation of the microservice testing service in real conditions.

Chapter VI

Conclusions

In this study, we analyzed the microservice architecture and compared it with other distributed computing approaches, such as multi-agent systems and actor-oriented systems. Using this information, we offered a testing methodology that uses a modified software test process described in ISO 29119 standard (ISO 25010, 2011). The described testing methodology was implemented as a distributed system with a web interface and published as an open source project[1]. In addition, the testing system architecture uses microservices to implement external test designs and techniques to provide testing capabilities in a heterogeneous environment. An analysis of different distributed computing techniques showed that there are many similarities between microservice testing, SOA testing, and other distributed computing techniques, but the core difference is the microservice infrastructure and deployment automation. The microservice testing methodology was designed and implemented to be used together with different continuous integration systems – for example, Jenkins (2018) or TeamCity (2018) – popular tools to automate the testing process and release management. The described methodology was illustrated using the developed microservice testing service in the testing of a simple open source microservice application. As an evaluation, we used and reported results of the applicability of the microservice testing service in real software development projects.

As a case study, the microservice testing service was applied in the Maintain project, which focuses on the systematic collection and analysis of the different software metrics for maintenance purposes. The main goal of the Maintain project is the reduction of the effort required during the maintenance phase using an early warning system that may highlight future problems during the development phase and system operation. To meet this goal, we built a service that accumulates both static and runtime metrics from the developed software and then analyzes them using different analyzers. The analyzer output visualizes the current quality of the whole project and also highlights possibly problematic modules. The microservice testing service has a modular and extendable architecture, which is not platform-specific. In the Maintain project, measurements were done using probes. Probes are small embedded programmatic agents that gather metrics

[1]Source code available at https://bitbucket.org/skayred/cloudrift2

from the developed software and submit them to the Maintain project analyzer. Probes communicate with the Maintain core using HTTP API, making it possible to implement a new probe and associate it with new analyzers to meet the updated requirements, for example, a new programming language or framework. The Maintain project also uses a schemaless solution for the output of the probes. Therefore, the associated analyzer can access specific attributes of the analyzed data: for example, the analysis of the user interface and the analysis of code quality require different data, and the output of associated probes should have different formats. The Maintain project offers the metrics reports of the developed software, and it allows to use different software metrics in a real development process. Future development of the Maintain project includes the implementation of the probes for different languages and test levels. In addition, the Maintain project was evaluated in several software development companies to obtain the proof-of-concept of the correctness of the analyzer and the early-warning system.

References

Aalst, L.V.D. (2010). *SOFTWARE TESTING AS A SERVICE (STAAS) author: Leo van der Aalst based on the original publication in: PNSQC proceedings*. Technical report. url: `http://leovanderaalst.nl/SoftwareTestingasaService-STaaS.pdf`.

Aalst, W.V.D. (2013). Service mining: Using process mining to discover, check, and improve service behavior. *IEEE Transactions on Services Computing*. ISSN 19391374, doi:10.1109/TSC.2012.25.

Altintas, I., et al. (2004). *Kepler: an extensible system for design and execution of scientific workflows*. IEEE. doi:10.1109/SSDM.2004.1311241, ISBN 0-7695-2146-0, ISSN 1099-3371.

Armbrust, M., et al. (2009). Above the Clouds: A Berkeley View of Cloud Computing. *EECS Department, University of California, Berkeley*. ISSN 00010782, doi:10.1145/1721654.1721672.

Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2016). Microservices Architecture Enables DevOps : An Experience Report on Migration to a Cloud-Native Architecture The Architectural Concerns for Microservices Migration The Architecture of Backtory Before the Migration. *IEEE Software*. ISSN 07407459, doi:10.1109/MS.2016.64.

Barham, P., et al. (2003). Xen and the art of virtualization. In: *Proceedings of the nineteenth ACM symposium on Operating systems principles - SOSP '03*. ACM. ISBN 1581137575, ISSN 01635980.

Beck, K., et al. (2001). *Manifesto for Agile Software Development*. Web Page. url: `http://agilemanifesto.org/`.

Bennett, K., et al. (2000). Service-based software: The future for flexible software. In: *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*. IEEE. ISBN 0769509150, ISSN 15301362.

Blaine, J.D. and Cleland-Huang, J. (2008). Software quality requirements: How to balance competing priorities. *IEEE Software*. ISSN 07407459, doi:10.1109/MS.2008.46.

Bohn, R.B., et al. (2011). NIST cloud computing reference architecture. In: *Proceedings - 2011 IEEE World Congress on Services, SERVICES 2011*, pp. 594–596. IEEE. ISBN 9780769544618.

Bondi, A. (2000). Characteristics of Scalability and Their Impact on Performance. *Proceedings of the 2nd international workshop on Software and performance*, pp. 195–203. ISSN 158113195X, doi:10.1145/350391.350432, url: http://portal.acm.org/citation.cfm?id=350432{&}dl={%}5Cnpapers2://publication/uuid/CC882EDE-5736-49BB-A98D-5941E4FD8F09.

Brooks, F. (1987). No Silver Bullet: Essence and Accident of Software Engineering. *IEEE Software*, 20, p. 12. ISSN 0018-9162, doi:http://dx.doi.org/10.1109/MC.1987.1663532.

Bryant, D. (2014). Prana: A Sidecar Application for NetflixOSS-based Services. url: https://www.infoq.com/news/2014/12/netflix-prana.

Buyya, R., et al. (2009). Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*. ISSN 0167739X, doi:10.1016/j.future.2008.12.001.

Canfora, G. and Di Penta, M. (2009). Service-oriented architectures testing: A survey. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5413, pp. 78–105. Springer. ISBN 978-3-540-95887-1, ISSN 03029743.

Clemson, T. (2014). *Testing Strategies in a Microservice Architecture*. Web Page. url: https://martinfowler.com/articles/microservice-testing/.

Docker (2014). Enterprise Application Container Platform | Docker. url: https://docker.com/.

Duan, Y., et al. (2015). Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends. In: *Proceedings - 2015 IEEE 8th International Conference on Cloud Computing, CLOUD 2015*. IEEE. ISBN 9781467372879, ISSN 2159-6190.

Durkee, D. (2010). Why cloud computing will never be free. *Communications of the ACM*. ISSN 00010782, doi:10.1145/1735223.1735242.

Ebert, C., Gallardo, G., Hernantes, J., and Serrano, N. (2016). DevOps. *IEEE Software*. ISSN 07407459, doi:10.1109/MS.2016.68.

Erl, T. (2005). Service-Oriented Architecture: Concepts, Technology, and Design. *City*, p. 760. ISSN 0131858580, doi:10.1109/SAINTW.2003.1210138.

Feeney, G.J., et al. (1974). Utility computing. In: *Proceedings of the May 6-10, 1974, national computer conference and exposition on - AFIPS '74*, p. 1003. New York, New York, USA: ACM Press. url: http://portal.acm.org/citation.cfm?doid=1500175.1500370.

Fielding, R.T. (2000). CHAPTER 5 Representational State Transfer (REST). *Style DeKalb IL*, pp. 76–106. doi:1, url: http://www.ics.uci.edu/{~}fielding/pubs/dissertation/rest{_}arch{_}style.htm.

Ford, N. (2015). *Building Microservice Architectures*. Web Page. url: http://nealford.com/downloads/Building{_}Microservice{_}Architectures(Neal{_}Ford).pdf.

Fox, A. and Patterson, D. (2012). *Engineering long-lasting software : an agile approach unsing SaaS and Cloud Computing*. Strawberry Canyon LLC. ISBN 0984881212, 217 p.

Gallaugher, J.M. and Ramanathan, S.C. (1996). Choosing a Client/Server Architecture. *Information Systems Management*, 13(2), pp. 7–13. ISSN 1058-0530, doi:10.1080/10580539608906981, url: http://dx.doi.org/10.1080/10580539608906981{%}5Cnhttp://www.tandfonline.com/doi/abs/10.1080/10580539608906981.

Gao, J., Bai, X., and Tsai, W.T. (2011). Cloud Testing - Issues, Challenges, Needs and Practice. *Software Engineering: An International Journal*. ISSN 2168-6149, doi: 10.1001/jamaneurol.2015.1743.

Geraci, A., et al. (1991). *IEEE Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries*. IEEE. ISBN 1559370793, 1 p.

Gold, N., Mohan, A., Knight, C., and Munro, M. (2004). Understanding service-oriented software. *IEEE Software*. ISSN 07407459, doi:10.1109/MS.2004.1270766.

Grundy, J., Kaefer, G., Keong, J., and Liu, A. (2012). Software Engineering for the Cloud. *IEEE Software*. ISSN 0740-7459, doi:10.1109/SBES.2012.12.

Hassan, Q.F. (2011). Demystifying Cloud Computing. *The Journal of Defense Software Engineering*. ISSN 03621340, doi:10.1007/978-1-84996-241-4.

Heiser, J. (1997). An overview of software testing. In: *1997 IEEE Autotestcon Proceedings AUTOTESTCON '97. IEEE Systems Readiness Technology Conference. Systems Readiness Supporting Global Needs and Awareness in the 21st Century*, pp. 204–211. IEEE. ISBN 0-7803-4162-7, url: http://ieeexplore.ieee.org/document/633613/.

Helal, S., Hammer, J., Zhang, J., and Khushraj, A. (2001). A three-tier architecture for ubiquitous data access. In: *Proceedings of IEEE/ACS International Conference on Computer Systems and Applications, AICCSA*, vol. 2001-Janua, pp. 177–180. IEEE. ISBN 0769511651, ISSN 21615330.

Herbsleb, J.D. and Moitra, D. (2001). Global software development. *IEEE Software*. ISSN 07407459, doi:10.1109/52.914732.

Hevner, A.R., March, S.T., Park, J., and Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly*. ISSN 02767783, doi:10.2307/25148625.

Hosono, S., et al. (2011). Fast development platforms and methods for cloud applications. In: *Proceedings - 2011 IEEE Asia-Pacific Services Computing Conference, APSCC 2011*. IEEE. ISBN 9780769546247.

IEEE (2012). IEEE Standard for System and Software Verification and Validation IEEE Computer Society. *IEEE Std 1012-2012 (Revision of IEEE Std 1012-2004)*, 2012(May). doi:10.1109/IEEESTD.2012.6204026.

ISO 25010 (2011). *ISO/IEC 25010: 2011*. Technical report. ISBN 0738156639, ISSN 10774866, 1–25 p., url: http://www.iso.org/iso/iso{_}catalogue/catalogue{_}tc/catalogue{_}detail.htm?csnumber=35733.

ISO 29119 (2013). 29119-1:2013 - ISO/IEC/IEEE International Standard for Software and systems engineering âĂŤ Software testing âĂŤ Part 1: Concepts and definitions. *ISO/IEC/IEEE 29119-1:2013(E)*, 2013, pp. 1–64. doi:10.1109/IEEESTD.2013.6588537.

ISO/IEC (2005). *ISO/IEC 25000:2005 Software Engineering - Software product Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE*. ISO/IEC.

Järvinen, P. (2012). *On research methods*. Opinpajan kirja. ISBN 9789529923342.

Jenkins (2018). *Jenkins*. Web Page. url: `https://jenkins.io/`.

Jennings, N.R. (2000). On agent-based software engineering. *Artificial Intelligence*. ISSN 00043702, doi:10.1016/S0004-3702(99)00107-1.

Jüttner, P., Kolb, S., Naumann, U., and Zimmerer, P. (1995). Integration Testing of Object-Oriented Software. *Software Quality Week*.

Kaner, C. (2006). Exploratory Testing. *Quality Assurance International*.

Kang, H., Le, M., and Tao, S. (2016). Container and microservice driven design for cloud infrastructure DevOps. In: *Proceedings - 2016 IEEE International Conference on Cloud Engineering, IC2E 2016: Co-located with the 1st IEEE International Conference on Internet-of-Things Design and Implementation, IoTDI 2016*. IEEE. ISBN 9781509019618, ISSN 07407459.

Kasurinen, J., Taipale, O., and Smolander, K. (2009). Analysis of Problems in Testing Practices. In: *Software Engineering Conference, 2009. APSEC '09. Asia-Pacific*, vol. 29, pp. 309–315. IEEE. ISBN 9780769539096, ISSN 1530-1362.

King, J.L. (1983). Centralized versus decentralized computing: organizational considerations and management options. *ACM Computing Surveys*, 15(4), pp. 319–349. ISSN 03600300, doi:10.1145/289.290.

Kit, E. and Finzi, S. (1995). *Software testing in the real world : improving the process*. ACM Press. ISBN 0201877562, 252 p.

Kitchenham, B. and Pfleeger, S.L. (1996). Software quality: the elusive target. *IEEE Software*. ISSN 07407459, doi:10.1109/52.476281.

Kyte, A. (2012). The Four Laws of Application Total Cost of Ownership. *Gartner*. url: `https://www.gartner.com/doc/1972915/laws-application-total-cost-ownership`.

Leavitt, N. (2009). Is cloud computing really ready for prime time? *Computer*. ISSN 00189162, doi:10.1109/MC.2009.20.

Lewis, J. and Fowler, M. (2014). *Microservices*. Web Page. url: `http://martinfowler.com/articles/microservices.html`.

Linthicum, D.S. (2010). *Praise for Cloud Computing and SOA Convergence in Your Enterprise*. Addison-Wesley Professional. ISBN 9780136009221.

Lu, T. and Chen, M. (2012). Simple and effective dynamic provisioning for power-proportional data centers. In: *2012 46th Annual Conference on Information Sciences and Systems, CISS 2012*. IEEE. ISBN 9781467331401, ISSN 1467331392.

March, S.T. and Smith, G.F. (1995). Design and natural science research on information technology. *Decision support systems*. ISSN 01679236, doi:10.1016/0167-9236(94)00041-2.

Marston, S., et al. (2011). Cloud computing - The business perspective. *Decision Support Systems*. ISSN 01679236, doi:10.1016/j.dss.2010.12.006.

Mauro Tony (2015). Microservices at Netflix: Lessons for Architectural Design. url: `https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/`.

Mell, P. and Grance, T. (2011). The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology. *Nist Special Publication*, 145, p. 7. ISSN 00845612, doi:10.1136/emj.2010.096966.

Merkel, D. (2014). *Docker: lightweight Linux containers for consistent development and deployment*. Linux Journal. url: `http://dl.acm.org/ft{_}gateway.cfm?id=2600241{&}type=html{%}5Cnhttp://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment`. doi:10.1097/01.NND.0000320699.47006.a3, ISBN 1075-3583, ISSN 1075-3583.

Mesosphere (2014). The Premier Platform for Building Data-Rich Apps. url: `https://mesosphere.com/`.

Nguyen, C.D., et al. (2011). Testing in multi-agent systems. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6038 LNCS, pp. 180–190. Springer. ISBN 9783642192074, ISSN 03029743.

OASIS (2006). Reference Model for Service Oriented Architecture 1.0. OASIS Standard. *Public Review Draft 2*, 2(October), pp. 1–31.

Olsen, E.R. (2006). Transitioning to software as a service: Realigning software engineering practices with the new business model. In: *2006 IEEE International Conference on Service Operations and Logistics, and Informatics, SOLI 2006*. IEEE. ISBN 1424403189.

Papazoglou, M. and Georgakopoulos, D. (2003). Service-oriented computing. *Communications of the ACM*, 46(10), pp. 24–28. doi:10.1109/WISE.2003.1254461, url: `http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=1607964`.

Patidar, S., Rane, D., and Jain, P. (2011). Challenges of software development on cloud platform. In: *Proceedings of the 2011 World Congress on Information and Communication Technologies, WICT 2011*. IEEE. ISBN 9781467301251, ISSN 978-1-4673-0125-1.

Pawson, R. (2011). *Continuous Delivery: Reliable Software Releases through Build, Test and Deployment Automation*. Addison-Wesley Professional. ISBN 978-0-321-60191-9.

Peffers, K., Tuunanen, T., Rothenberger, M.A., and Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, 24(3), pp. 45–77. ISSN 0742-1222, doi:10.2753/MIS0742-1222240302, url: http://www.tandfonline.com/doi/full/10.2753/MIS0742-1222240302.

Rhoton, J. and Haukioja, R. (2011). *Cloud computing architected*. Recursive Press. ISBN 0956355617, 363 p.

Riungu-Kalliosaari, L., Taipale, O., and Smolander, K. (2013). Desired quality characteristics in cloud application development. In: *ICSOFT 2013 - Proceedings of the 8th International Joint Conference on Software Technologies*, pp. 303–312. Springer. ISBN 9789898565686, url: http://www.scopus.com/inward/record.url?eid=2-s2.0-84887036419{&}partnerID=40{&}md5=3e95c2ea27d06af6e5b88705f523334e.

Ruby on Rails (2018). *Ruby on Rails | A web-application framework that includes everything needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern*. Web Page. url: https://rubyonrails.org/.

Runeson, P. (2006). A survey of unit testing practices. *IEEE Software*, 23(4), pp. 22–29. ISSN 0740-7459, doi:10.1109/MS.2006.91, url: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1657935.

Sahoo, M. (2009). IT Innovations: Evaluate, strategize, and invest. *IT Professional*. ISSN 15209202, doi:10.1109/MITP.2009.128.

Savchenko, D. and Radchenko, G. (2014). Mjolnirr: A hybrid approach to distributed computing architecture and implementation. In: *CLOSER 2014 - Proceedings of the 4th International Conference on Cloud Computing and Services Science*. SCITEPRESS. ISBN 9789897580192.

Sinha, A. (1992). Client-server computing: current technology review. *Communications of the ACM*, vol.35, no, pp. 77–98.

Sneps-Sneppe, M. and Namiot, D. (2014). Micro-service Architecture for Emerging Telecom Applications. *International Journal of Open Information Technologies*. ISSN 2307-8162.

Sodhi, B. and Prabhakar, T.V. (2011). Assessing suitability of cloud oriented platforms for application development. In: *Proceedings - 9th Working IEEE/IFIP Conference on Software Architecture, WICSA 2011*. IEEE. ISBN 9780769543512.

Sogeti (2016). *WORLD QUALITY REPORT 2015-16*. Sogeti.

Sommerville, I. (2004). Software engineering (7th edition). *Engineering*. ISSN 1098-6596, doi:10.1111/j.1365-2362.2005.01463.x.

Stanley, M. (2017). *Software Sector Is Set for Strong Growth in 2017 | Morgan Stanley*. Web Page. url: https://www.morganstanley.com/ideas/software-sector-growth.

Tasharofi, S., et al. (2012). TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. *Lecture Notes in Computer Science*, pp. 219–234. doi:10.1007/978-3-642-30793-5_14, url: `http://link.springer.com/10.1007/978-3-642-30793-5{_}14`.

Tassey, G. (2002). *The Economic Impacts of Inadequate Infrastructure for Software Testing - Planning Report 02-3*. Technical report. ISSN 1043-8599.

TeamCity (2018). *TeamCity: the Hassle-Free CI and CD Server by JetBrains*. Web Page. url: `https://www.jetbrains.com/teamcity/`.

Thönes, J. (2015). *Microservices*. IEEE. doi:10.1109/MS.2015.11, ISBN 9781467372848, ISSN 07407459.

Vamp (2014). Vamp | Smart Application Releasing for Modern Cloud Platforms. url: `https://vamp.io/`.

Vogels, W. (2006). A Conversation with Werner Vogels. *Queue*. ISSN 15427730, doi:10.1145/1142055.1142065.

Wang, M. and Wang, H. (2006). From process logic to business logic - A cognitive approach to business process management. *Information and Management*, 43(2), pp. 179–193. ISSN 03787206, doi:10.1016/j.im.2005.06.001.

Wooldridge, M. (1997). Agent-based software engineering. *IEE Proceedings Software Engineering*. ISSN 13645080, doi:10.1049/ip-sen:19971026.

Yau, S. and An, H. (2011). Software engineering meets services and cloud computing. *Computer*. ISSN 00189162, doi:10.1109/MC.2011.267.

Zech, P., Felderer, M., and Breu, R. (2012). Towards a model-based security testing approach of cloud computing environments. In: *Proceedings of the 2012 IEEE 6th International Conference on Software Security and Reliability Companion, SERE-C 2012*. IEEE. ISBN 9780769547435.

Zimmermann, O. (2017). Microservices tenets. *Computer Science - Research and Development*. ISSN 1865-2034, doi:10.1007/s00450-016-0337-0.

Publication I

Savchenko D. and Radchenko G.
Mjolnirr: private PaaS as distributed computing evolution
2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, 2014, pp. 386-391.

Mjolnirr: private PaaS as distributed computing evolution

D.I. Savchenko* and G.I. Radchenko*
*South Ural State University, Chelyabinsk, Russia
dmitry@mjolnirr.com, radchenkogi@susu.ac.ru

Abstract - Cloud computing became very popular during the last few decades. It provides convenient access to remote computing resources for individual users and organizations. However, there are still security issues arise if the private data is transmitted to the public cloud for processing. This issue can be resolved with private cloud systems. In this paper, we propose the design and implementation of a Mjolnirr private cloud platform for development of the private PaaS cloud systems. It provides infrastructure for cloud applications development, including software developer kit and message brokering system. Proposed cloud platform can be deployed on resources of a distributed computing system, including idling resources of personal computers. For a developer, a Mjolnirr application is represented as a collection of independent components communicating by a message exchange communication protocol.

Keywords - PaaS, Mjolnirr, Cloud computing, distributed system, message passing

I. INTRODUCTION

Cloud computing became very popular computing paradigm during the last decade [15], [25]. This approach to resource provisioning offers convenient access to computing resources, which makes it easy to use them for custom services development.

Before the introduction of the cloud-computing concept, there was an another distributed computing concept called "grid computing" [11]. It is still used by the scientific community to solve extra-large and resource-intensive scientific tasks. The name "grid" originated from the metaphor of the power grid. However, unlike electricity devices, integration of new resources to the grid-computing environment is not trivial.

Meanwhile, PaaS (Platform as a Service) [9] solutions provide the ability to create custom applications [6]. However, the problem is that the most PaaS solutions are deployed on remote hosting platforms. The owner of the application may not know exactly where his information is stored, as well as he cannot be sure about the safety and security of his information [20]. These issues might be solved by the private PaaS platform [10].

In this paper, we propose the design and implementation of the private cloud platform called "Mjolnirr". There are several solutions available now that provide private IaaS (VMWare Private Cloud, OpenStack) and PaaS (Yandex Cocaine, AppFog, Stackato) levels of service. Comparing to these solutions, main features of our approach are integrated messaging subsystem, flexible workload management and native UNICORE [24] grid environment integration module.

This paper is organized as follows. In section II we present the concept of the Mjolnirr cloud platform. In section III we describe the results of the analysis of existing cloud platforms and compare them with the Mjolnirr platform. In section IV we describe the architecture of the Mjolnirr platform and the component communication protocol. In section V we describe the implementation of the Mjolnirr platform and demonstrate the process of custom applications development. In section VI we describe the platform performance evaluation. In section VII we summarize the results of our research and further research directions.

II. MJOLNIRR PLATFORM CONCEPT

The Mjolnirr private cloud platform provides development of a Java-based private PaaS cloud infrastructure. A Java application or library can be implemented as a Mjolnirr-based service. The Mjolnirr platform provides infrastructure for cloud applications development, including software developer kit, message-brokering system. For a developer, an Mjolnirr application is represented as a collection of independent components communicating by message exchange. This approach allows development of flexible and scalable cloud applications.

Mjolnirr also provide integration with the UNICORE grid environment through the DiVTB [18] platform. The DiVTB (Distributed Virtual Test Bed) platform provides a task-oriented approach for solving specific classes of problems in computer-aided engineering through resources supplied by grid computing environments. Thus, Mjolnirr can be used to provide infrastructure both for scientific projects with the grid systems and in a business infrastructure.

During our research, we should:

- analyze technologies of cloud systems and private cloud platforms development;
- develop Mjolnirr platform architecture;

The reported study was partially supported by Grant Fund for Assistance to Small Innovative Enterprises in Science and Technology: "UMNIK" Programme, research project No. №0000829 and by RFBR, research project No. 14-07-00420-a.

- implement Mjolnirr platform;
- evaluate the performance and scalability of the Mjolnirr platform.

III. Analysis of Existing Solutions

An investigation shows that C-level executives and IT managers in enterprise companies have issues with integration of cloud computing in their data processing [5]. One of the most serious concerns is the possibility of data breaches. A cloud provider can give access to the company's private data (accidentally or intentionally) or may bring harm to the data owner [14], [21].

For the prevention of such privacy threats, it is possible to use the encryption of data that is stored in the cloud [19], [22]. However, this approach is effective only when cloud is used just for storage. If the data is processed in the cloud, it become available decrypted in the memory of the host, where the processing occurs [4]. In addition to this drawback, the owner of the data does not control the location of his virtual machine, so it can be moved to the physical computer with the virtual machine that contains malware. It may cause the ban of the virtual machine or forfeiture of a computer containing these virtual machines.

Nowadays, there are two ways to ensure the data security in the cloud. The first way is called "trusted computing". It ensures security of virtual machines in the cloud [7]. Nevertheless, your data cannot be completely safe. For example, in present IaaS clouds, the virtual machine can be dynamically moved to the other host, but the concept of trusted computing provides security only for virtual machines running on the same host. Otherwise, the concept of TCCP (Trusted Cloud Computing Platform) [12] solves this problem by creating the safe environment for running virtual machines. In fact, neither of these approaches solves the problem of VMs placement on the same host with a malicious VM.

Another way to deal with the security issues is to deploy the cloud infrastructure on the private hardware. In this case, the owners themselves will protect all the data. However, buying and maintaining of the hardware is more expensive than rent of computing resources [9], [10].

The simplest way is to create a private cloud system, and there are several different products existing in this area. They can be divided into two categories according to the level of abstraction – infrastructure (IaaS) solutions (e.g. OpenStack and VMWare Private Cloud), and platform (PaaS) solutions (e.g. Yandex Cocaine, AppFog and Stackato).

OpenStack [17] is a popular open source package for private cloud systems provision. It provides infrastructure layer based on virtual machines abstraction. Software developers should install web servers, load balancers and other specific services manually. It offers great opportunities, but IaaS maintenance is expensive. VMWare Private Cloud [16] also provides IaaS-level solution, but on a commercial basis. VMWare Private Cloud offers advanced virtualization capabilities, as well as, the commercial support.

Yandex Cocaine and AppFog platforms provide the ability to create private PaaS solutions based on application containers [2], [3]. These platforms allow development of a "Heroku-like" [13] application hosting. They provide a number of built-in modules and a server infrastructure. Stackato [1] provides all advantages of the mentioned solutions and provide local application store. Some of them use Docker [8] containers to provide isolated environment to the applications, but Docker can be used only on Linux hosts. However, all of the above solutions consider custom applications as a monolith and ignore its internal structure that is why it is not possible to automatically and effectively balance the load on the individual subsystems applications. In addition to this drawback, none of these solutions considers end-user workstations as computing resources providers.

We decided to create a Mjolnirr cloud platform, which solves these problems as follows:

- Using popular open source libraries and programming languages, as well as the opportunity to work not only on the server platform, but also on the

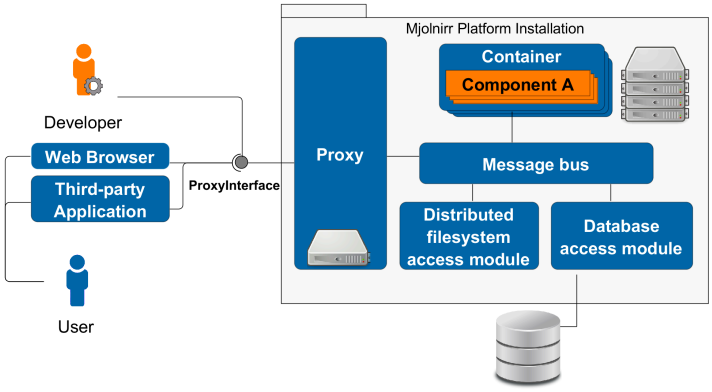

Figure 1: Mjolnirr platform architecture

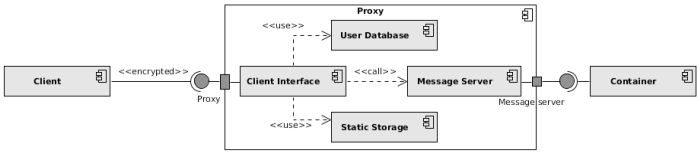

Figure 2: Mjolnirr proxy architecture

personal computers, using idling resources of the computer system, will provide cost reduction.

- Using popular programming languages, developer tools and SDK will provide ease of application development.
- Integration of new resources ease will be provided by the application architecture modularity and custom components reusability.

IV. ARCHITECTURE

Mjolnirr platform architecture is presented at the figure 1. The Mjolnirr platform consists of the following components.

- *Proxy* provides access to the cloud system for the external clients and manages the communication between cloud application components. Proxy is the only component that accessible from the external network;
- *Containers* are deployed on computing nodes and responsible for hosting of cloud applications components and message transmission. It can be deployed both on the personal computer and on the nodes of dedicated server;
- *Components* are custom applications, created to run in cloud environment. Each component has a unique name. UI components (applications) are multi-page applications, it means that applications has different representations for different actions, like different web-pages of one web-application;
- *Client* (browser or third-party application) is an application, which render user interfaces and provide user interaction with cloud applications, hosted in Mjolnirr platform. All clients should use encrypted channel and client certificates to communicate with the proxy.

A. Proxy

The Proxy (see fig. 2) provides access to Mjolnirr system from the external network. The Proxy performs the following actions:

- it stores and provides static resources (page layout descriptions, images etc.) of the deployed cloud applications in the Static Storage;
- it acts as a messaging server for the components of cloud applications;
- it handles client's requests to the Client Interface;
- it performs the authorization and authentication of users.

The external Proxy interface provides the following methods as RESTful API:

- *getUI (applicationName, pageName)* – returns the page layout description;
- *getResourceFile (applicationName, resourceName)* – gets the static file;
- *processRequest (componentName, methodName, args)* processes a client request, redirecting it to the first free suitable component. This method can be called directly from the application page (e. g. with JavaScript).

B. Container

The container (Fig. 3) provides cloud application component hosting. The container provides an API for remote components instances method invocation. The Mjolnirr installation can have any number of containers.

Any Mjolnirr-based application consists of independent components, which use a built-in messaging system, implemented in the basis of the Publisher-Subscriber pattern. The Proxy is responsible for message queue maintenance. The Message Server of the Proxy provides publisher-subscriber messaging service for cloud application components. Mjolnirr Containers subscribe to Message Channels that operates as a broadcast delivery – any message sent to the Message Channel will be transmitted to the subscribers of this channel. Each cloud application instance is subscribed on two types of Message Channels:

- Component Public Channel: every instance of the

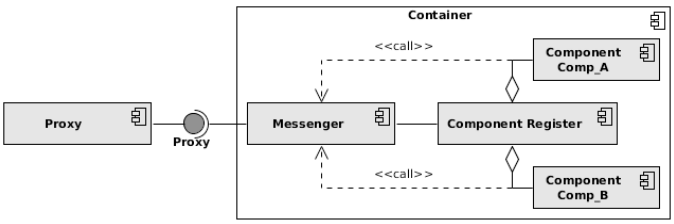

Figure 3: Mjolnirr container architecture

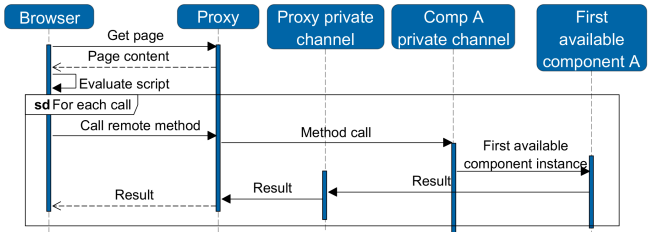

Figure 4: Mjolnirr container architecture

cloud application component is subscribed on this public channel. This a listener channel – when any message come to this channel, the appropriate component instance will be invoked.

- Instance Private Channel: provides direct communication between instances.

When container starts, it performs several initialization steps. The order of the container initialization:

- Container registers in the proxy database and receives the list of components to load in response;
- For each component from the list:
  - The container checks it's local cache, for each missing package container downloads it from the proxy;
  - Container runs the component;
  - Container subscribes on the Component Public Chanel and Instance Private Channel for the loaded component.

Container provides messaging API to the hosted component. Typical message transmission sequence looks like shown below (Fig. 4):

- when one component calls another component remotely, the container sends the call to a Component Public Channel (for example, when the *Comp_A* component calls the *Comp_B* component, the application *Comp_A*, will send the request to the channel *CPC_B*);
- The first available instance of the component in the cloud system processes the request;
- The response is returned to a private IPC channel of the instance that sent a message.

In addition, container has an opportunity to work in stand-alone mode. In this mode, the container does not support communication with other containers and acts as a stand-alone computing system (container and proxy at the same time).

C. Components

From the developer's point of view, Mjolnirr cloud application is a collection of independent components communicating by message exchange. Components represented as a package that contains the following information:

- manifest, that provides the interface of the component, including description of provided methods and their parameters;
- executables to handle incoming requests;
- static files, used in pages rendering (images, page layout descriptions and scripts) for UI provision.

Each component can be:

- Application Component: provides the user interface definition, scripts, styles and UI-specific actions. Optionally contains complex logic.
- Module Component: represents a single entity in the domain logic of the application. The Module Component provides data processing and storage, but doesn't provide interface and static files.

D. Clients

Mjonirr platform uses HTML and JavaScript to represent end-user interface in common Internet browser. But it's possible to consume Mjolnirr resources in third-party application because of using commonly used and standardized protocols.

V. Implementation

A. Components

Each custom component must have a manifest: an XML file that contains all the necessary information about the component's name, the name of the main class, list of methods and a list of parameters for each method. Sample manifest for the Calculator service is shown below.

```
<hive>
<application
  name="Calculator"
  classname="org.exmpl.Calculator">
  <methods>
   <method
     name="calculate"
     type="java.lang.String">
    <parameters>
     <parameter
       type="java.lang.String"/>
    </parameters>
   </method>
  </methods>
</application>
</hive>
```

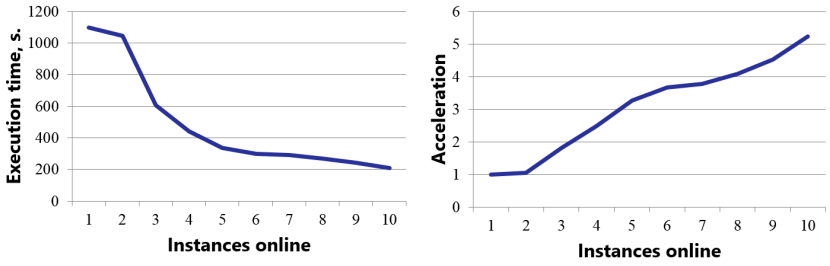

Figure 5: Performance evaluation

The container parses the manifest for each loaded component.

As stated above, the manifest must have the name of the main class as a fully qualified name of the facade class of the described components. Each facade class must have an implementation of the method "*initialize*". The container calls this method immediately after instantiating the component and passes the component *context there*. This context contains information about the working directory of the current application instance and the directory containing the configuration files.

B. *Messaging subsystem implementation*

Mjolnirr platform messaging subsystem is implemented on the basis of the open source HornetQ queue, which is built into the Proxy component. HornetQ was chosen for the following reasons:

- it can work in embedded mode;
- it provides high queue management performance [23].

The container provides messaging API to the components. Example of using messaging interface is shown below.

```
Communicator communicator =
        new HornetCommunicator();

YourResultClass result =
Communicator
     .sendSync(
        componentContext,
        "componentName",
        "methodName",
        argsList,
        YourResultClass.class);

communicator.send(
   componentContext,
   "componentName",
   "methodName",
   argsList,
   new Callback<YourResultClass>() {
       public void
       run(YourResultClass res) {}
   });
```

C. *UNICORE integration*

The Mjolnirr platform provides a UNICORE 6 integration module. This module uses DiVTB Server for communications with grid environment. It enables Mjolnirr applications to use UNICORE grid resources easily.

The concept of DiVTB allows users to split the supercomputing simulation in two phases - building experiment and launching the test stand. The main advantage of the DiVTB concept is the ability to use several times once set up the experiment.

UNICORE integration module provides the following methods:

- *uploadTestBed* – used for test bed archive uploading;
- *indexTestBed* – indexes specific text bed;
- *indexAllTestBeds* – indexes all available test beds;
- *createExperiment* – create experiment from one of the test beds;
- *startExperiment* – start experiment with specific parameters;
- *getStatus* – get experiment status (started, finished or failed);
- *getResults* – get experiment results.

VI. PERFORMANCE EVALUATION

Mjolnirr platform was evaluated using text-processing experiment. 1 gigabyte of text data was divided on 100 parts and distributed between available worker components for processing. Each worker divided text on words and counted frequency for each unique word. Pieces of work were distributed automatically – each worker polled Message Bus to receive new task. The results of the experiments are shown on the figure 5.

We used 1 virtual node with 2 CPU cores and 2048 MB RAM for proxy node and 10 virtual nodes with 1 CPU core, 512 MB RAM as worker nodes. Virtual machines were installed on the computing server node

with the following characteristics: 2 Intel Xeon X5680 CPU (6 Cores, 3.33 GHz) 12 GB DDR3 RAM.

Experiments have shown that the platform is stable. Average execution time on **10** containers was **219** seconds, comparing to **1107** seconds using 1 worker. Thus, acceleration of parallel word frequency counter task reached **5.3** on **10** worker nodes.

VII. CONLUSION

In this article, we described the design and implementation of a private cloud platform called Mjolnirr, which allows creating distributed cloud applications on private computing resources. Main features of the described platform are advanced messaging system and distributed computing support.

As a further development of the Mjolnirr platform we will investigate and implement application-level migration support, integration with the advanced resource monitoring systems, flexible adaptation to load changes, advanced system security and application store. The application store will reduce the number of duplicate software products and simplify the creation of individual business infrastructure to meet the needs of a particular company.

REFERENCES

[1] Activestate.com. *Stackato: The Platform for the Agile Enterprise ActiveState*. [Online] Available from: http://www.activestate.com/stackato [Accessed 22 Jan 2014].

[2] Api.yandex.com. *Cocaine. Cocaine technology description*. [Online] Available from: http://api.yandex.com/cocaine [Accessed 22 Jan 2014].

[3] Appfog.com. *AppFog - Platform as a Service for public clouds*. [Online] Available from: https://www.appfog.com/ [Accessed 22 Jan 2014].

[4] Armbrust M, Fox A, Griffith R, Joseph A, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, Others. A view of cloud computing. *Communications of the ACM*. 2010; 53 (4): 50--58.

[5] Avanade.com. *Global Study: Cloud computing provides real business benefit | Avanade*. [Online] Available from: http://www.avanade.com/us/about/avanade-news/press-releases/Pages/Global-Study-Cloud-Computing-Provides-Real-Business-Benefits-But-Fear-of-Security-and-Control-Slowing-Adoption-page.aspx [Accessed 22 Jan 2014].

[6] Chang W, Abu-Amara H, Sanford J. *Transforming enterprise cloud services*. Dordrecht: Springer; 2010.

[7] Christodorescu M, Sailer R, Schales D, Sg, Urra D, Zamboni D. Cloud security is not (just) virtualization security: a short paper. 2009;: 97--102.

[8] Docker.io. *Docker: the Linux container engine.* [Online] Available from http://www.docker.io/ [Accessed 03 Feb 2014]

[9] Fanning K, Centers D. Platform as a service: Is it time to switch?. *Journal of Corporate Accounting & Finance*. 2012; 23 (5): 21--25.

[10] Fortis T, Munteanu V, Negru V. Towards a service friendly cloud ecosystem. 2012;: 172--179.

[11] Foster I, Kesselman C. *The Grid 2*. Burlington: Elsevier; 2003.

[12] Garfinkel T, Pfaff B, Chow J, Rosenblum M, Boneh D. Terra: A virtual machine-based platform for trusted computing. 2003; 37 (5): 193--206.

[13] Heroku.com. *Heroku | Cloud Application Platform*. [Online] Available from: https://www.heroku.com/ [Accessed 22 Jan 2014].

[14] Kamara S, Lauter K. Cryptographic cloud storage. *Proceedings of Financial Cryptography: Workshop on Real-Life Cryptographic Protocols and Standardization*. 2010;: 136-149. [Accessed 22 Jan 2014].

[15] Mell P, Grance T. The NIST definition of cloud computing (draft). *NIST special publication*. 2011; 800 (145): 7.

[16] Peng J, Zhang X, Lei Z, Zhang B, Zhang W, Li Q. Comparison of several cloud computing platforms. 2009;: 23--27.

[17] Pepple K. *Deploying OpenStack*. Sebastopol, CA: O'Reilly; 2011.

[18] Radchenko G, Hudyakova E. Distributed Virtual Test Bed: an Approach to Integration of CAE Systems in UNICORE Grid Environment. *MIPRO 2013 Proceedings of the 36th International Convention*. 2013;: 183-188. [Accessed 22 Jan 2014].

[19] Ren L, Yu X, Fletcher C, Van Dijk M, Devadas S. Design Space Exploration and Optimization of Path Oblivious RAM in Secure Processors.. *IACR Cryptology ePrint Archive*. 2013; 2013: 76.

[20] Rhoton J, Haukioja R. *Cloud computing architected*. [Tunbridge Wells, Kent]: Recursive Press; 2011.

[21] Ristenpart T, Tromer E, Shacham H, Savage S. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. 2009;: 199--212.

[22] Santos N, Gummadi K, Rodrigues R. Towards trusted cloud computing. 2009;: 3--3.

[23] Spec.org. *All Published SPEC SPECjms2007 Results*. [Online] Available from: http://www.spec.org/jms2007/results/jms2007.html [Accessed 22 Jan 2014].

[24] Streit A. UNICORE: Getting to the heart of Grid technologies. *eStrategies*. 2009; 3: 8-9. [Accessed 22 Jan 2014].

[25] Vaquero L, Rodero-Merino L, Caceres J, Lindner M. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*. 2008; 39 (1): 50--55.

Publication II

Savchenko D., Radchenko G., and Taipale O.
Microservice validation: Mjolnirr platform case study
2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, 2015, pp. 235-240.

Microservices validation: Mjolnirr platform case study

D.I. Savchenko*, G.I. Radchenko* and O. Taipale**
*South Ural State University, Chelyabinsk, Russia
**Lappeenranta University of Technology, Finland
dmitry.savc@gmail.com, radchenkogi@susu.ac.ru, ossi.taipale@lut.fi

Abstract – Microservice architecture is a cloud application design pattern that implies that the application is divided into a number of small independent services, each of which is responsible for implementing of a certain feature. The need for continuous integration of developed and/or modified microservices in the existing system requires a comprehensive validation of individual microservices and their co-operation as an ensemble with other microservices. In this paper, we would provide an analysis of existing methods of cloud applications testing and identify features that are specific to the microservice architecture. Based on this analysis, we will try to propose a validation methodology of the microservice systems.

Keywords - Microservices, Services Oriented Architecture, Cloud computing, PaaS, testing, validation

I. INTRODUCTION

The microservice architecture is a cloud application design pattern that implies that the application is divided into a number of small independent services, each of which is responsible for implementing of a certain feature. Microservices can be considered as meta-processes in a Meta operating system (OS): they are independent, they can communicate with each other using messages and they can be duplicated, suspended or moved to any computational resource and so on.

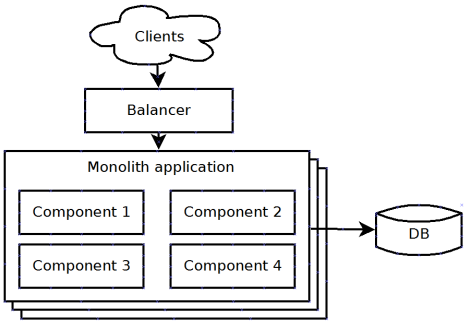

Figure 1. Monolithic system architecture

There are several examples of software systems, which were designed and implemented in accordance with microservice concept. Such companies as Netflix [19] and SoundCloud [3] use the microservice concept in their architectures, while Amazon [20] offers services like S3, which can be considered as microservices.

We can describe microservice architecture by comparing it with the monolithic architecture style, when an application built as a one big single unit (figure 1). The server-side monolithic application will handle HTTP requests, execute domain logic, retrieve and update data from the database, and select and populate HTML views to be sent to the browser. Such a design approach may result in that a change made to a small part of the application, requires the entire monolith to be rebuilt and deployed. Over time, it is often hard to keep a good modular structure, making it harder to keep changes that ought to only affect one module within that system [13].

In a case of microservice architectural style (figure 2), we can divide the application into a several independent components (microservices) and scale them independently. It is useful in a case of high-loaded systems and systems with lots of reusable modules.

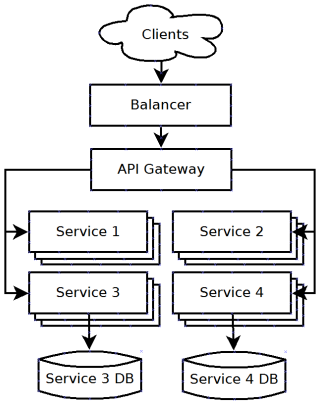

Figure 2. Microservice system architecture

Some developers and researchers believe that the concept of microservices is a specific pattern of implementation of Service Oriented Architecture [11]. In particular, the developers of the Netflix platform, which is considered the embodiment of the microservice architecture, named their approach of a design of software

The reported study was partially supported by RFBR, research project No. 14-07-00420-a and by Grant of the President of the Russian Federation No. MK-7524.2015.9

architecture as "a fine grained Service Oriented Architecture" [12]. However, the microservice pattern defines following specifics of software services development process [13]:

- microservices should use lightweight mechanisms for the communication (often an HTTP resource API);
- microservices should be built around business capabilities;
- microservices should be independently deployable by fully automated deployment machinery;
- there should be a bare minimum of centralized management of these services, architecture should provide decentralized service governance and decentralized data management.

However, the microservice approach is not an ultimate solution to deal with the system complexity. Using microservice approach, we are shifting the accidental complexity [2] from inside of the application out into infrastructure. That is why we should use special approaches to manage that complexity. Programmable infrastructure allows automating the microservices lifecycle management [18]; including automation of service deployment and termination, load balancing, service naming, continuous integration and delivery etc. Such support can be provided by a PaaS cloud platform, in order to scale the system automatically, manage it and provide middleware for the message communication. In addition, there should be implemented monitoring mechanisms to control the flow of the microservice application. It can be implemented as a specialized PaaS [16] solution or integrated in the existing PaaS.

Since the continuous integration is the only option of the implementation of the microservices architecture pattern, the question arises about the support this process on the methodological and software level. One of the most important stages of the process of continuous integration and deployment is validation of the developed software.

Our research goal is to develop the microservice validation methodology and a software support for microservice testing. To reach this goal, during our research we should solve the following tasks:

- to provide a review of existing software validation methods to identify testing features that are specific to microservice architecture;
- to develop a methodology for microservice systems validation, including approaches for component and integration testing;
- to develop a software solution that provides support for microservice systems testing on the basis of the proposed methodology.

In the framework of the current article, we would concentrate on the analysis of the current software validation standards and approaches and their relevance to the microservice architecture pattern. This paper is organized as follows. In Section II we will provide a review of the research and standards, related to the microservices testing and validation. In Section III we would try to identify special aspects of microservice validation, related to the component, integration and system testing. In Section IV we would propose microservice validation techniques for the Mjolnirr cloud platform prototype. In section V we summarize the results of our research and further research directions

II. Related research

Microservices concept is relatively new, so there are few materials in the field of microservice validation and testing right now. Basic information about the microservice approach is provided in the article [13] by James Lewis and Martin Fowler. In this article, they define the concept of microservice as well as the specific features of microservice architecture. Toby Clemson in his article [5] provides strong emphasis on the analysis of possible strategies of microservice systems testing, including certain types of microservice systems architecture; unit, integration, component and contract testing. Neal Ford provides a review of the proposed approaches on this subject is his work [6]. He describes basic ideas about microservice systems development, monitoring and testing. In addition, several commercial companies successfully moved to microservice architectures and published their result. For example in the paper [14] Netflix provide an overview of their PaaS system and mechanisms that they use to provide features of their platform to the components that initially was not designed to be used inside of the Netflix PaaS platform.

In our previous works, we described a prototype of distributed computing system called Mjolnirr [17]. This system works with isolated components, connected using message passing pattern. This system also can be considered as a microservice platform, because each component is isolated, fine-grained and has open standardized interface. In addition, automated scheduling and planning mechanisms have been implemented within the Mjolnirr platform in order to optimize the system load [15].

The ISO/IEC standards, ISO/IEC 25010 – Software Product Quality [7] and ISO/IEC 29119 – Software Testing [8], serve as the starting point of the study of microservice system testing and validation. Generally, we define testing as a combination of verification and validation [8]. In this research, we define microservice testing as microservice validation, because in general it is impossible to apply static testing methods to the compiled microservice.

As we mentioned before, we can consider microservices as a subset of Service Oriented Architecture (SOA). Thus, we can use SOA validation techniques in application to microservices systems. Some of existing approaches to SOA validation use SOA's BPEL business model for constraints-based SOA functional testing [10], other use white-box testing [1] to determine, is service covered with tests enough.

When we are talking about service testing and validation, we should decide, which quality characteristics are important for the product we are going to validate. Quality in use is defined in the ISO/IEC 25011 standard: “Quality in use of a service is the degree to which a

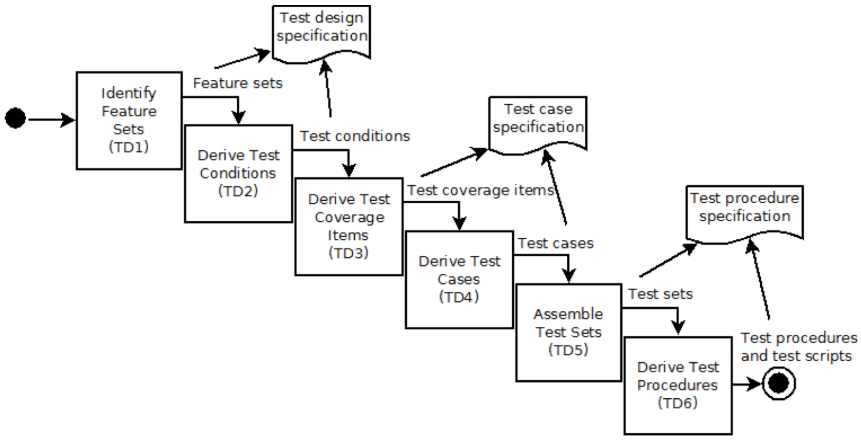

Figure 3. ISO/IEC/IEEE29119-2 Test Design and Implementation Process (modified)

service can be used by specific users to meet their needs to achieve specific goals with effectiveness, efficiency, satisfaction, freedom from risk and SLA coverage" [8]. ISO/IEC 25011 standard defines eight quality characteristics of services: Service Suitability, Service Usability, Service Security, Service Reliability, Service Tangibility, Service Responsiveness, Service Adaptability, and Service Maintainability. All of them, except Service Tangibility, can be applied to microservices. In addition, in a case of microservice system, we do not care about effectiveness and efficiency – these characteristic should be provided and measured by application programmer. However, satisfaction (stability of the whole system, quick response, etc.), freedom from risk and SLA coverage are important when we want to validate microservice system.

According to ISO/IEC 29119-4 standard, test design and implementation process consists of 6 steps – identifying feature sets (TD1), deriving test conditions (TD2), deriving test coverage items (TD3), deriving test cases (TD4), assembling test sets (TD5) and deriving test procedures (TD6) (see Fig. 3) [8]. To describe the microservice validation methodology, we should focus on the TD2, TD3, TD4 and TD6 steps.

III. Special Aspects of Microservice Validation

In this study, we define microservice as a software-oriented entity, which has following features:

- *Isolation* from other microservices as well as from the execution environment based on a virtualized container;
- *Autonomy* – microservices can be deployed, destroyed, moved or duplicated independently. Thus, microservice cannot be bound to any local resource because microservice environment can create more than one instance of the same microservice;
- *Open and standardized interface* that describes all available communication methods (either API or GUI);
- *Microservice is fine-grained* – each microservice should handle its own task.

Let us consider the most significant stages of the validation process, in relation both to the software as a whole and in particular microservice systems and define the features of the validation to be considered for microservice systems. By *unit validation* we understand individual validation of the microservice, by *integration validation* – validation of the interaction between individual microservices and by *system validation* – validation of the whole system (see Table 4).

System validation does not seem to offer features to microservices testing, since validation of the system level does not consider the internal structure of the application. The internal structure of the system can be considered as a black box, so there would be no difference in the system validation process for any application for which network resources access is needed.

A. Microservice unit validation

Functional Unit validation of microservice systems involves independent validation of individual microservices to meet the functional requirements. This

TABLE I. Featured Types and Levels of Testing

	Component	Integration	System
Functional	+	+	–
Load	+	+	–
Security	+	+	–
– - no features for microservice validation + - has features for microservice validation			

type of validation is based on a formal description of the functional requirements (including requirements as input and output data) imposed on each microservice. To provide the functional unit validation, microservice does not require to be deployed in a cloud environment, as this type of validation considers microservice as an isolated component.

Development of a software system in accordance with microservice approach is a process of development of individual independent microservices interacting exclusively based on open protocols. Thus, each microservice represents independent software product that should be developed and tested independently of the other microservices. Functional unit validation of the microservices can be divided into two stages (see Fig. 7):

1. unit, integration and system validation of the microservice source code or/and internal components (basic level);
2. self-validation of the microservice interface.

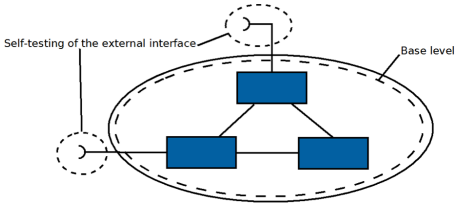

Figure 4. Functional unit validation of the microservice

Each microservice can be a relatively complex software system, which consists of several software components (local storage, web server, etc.) encapsulated in a container. Even in a case of ready-made software solutions, integration of such a component ensemble inside a container must be accompanied by validation the correctness of these components work together.

After that, it is necessary to test the interface provided by the individual microservice on its compliance with the specifications defined in the design. In this case, the entire microservice should be tested in general using its external interface.

Functional unit tests of a microservice should be conducted automatically on each microservice configuration change.

Load unit validation is aimed at the individual microservice validation at a certain load. Such validation may also be performed automatically for each microservice. Microservice load validation allows to identify resources that should be provided to the microservice container to ensure its correct work.

Security unit validation is aimed at the security and isolation validation of the individual microservice. As in the case of functional validation, microservice security does not end with the security of its individual classes. In component security validation we test the security of the whole microservice in the complex (for example, for typical vulnerabilities of web servers, injections, etc.). This stage of validation is performed without the presence of application programmer.

B. *Integration microservice validation*

Integration validation of microservice systems involves validation of all types of communications between the microservices. It includes validation of communication protocols and formats, resolution of deadlocks, shared resource usage and messaging sequence. To provide all of this validation techniques, we need to track all the messages, transmitted between microservices, and build the messaging graph. For this purpose, we need to know the messaging direction, message origin and destination. This information must be provided by microservice standard contract.

Functional integration validation ensures interoperability of the validation individual microservices. In this type of validation, we check for correctness of sending requests from one microservice to another, the sequence of interactions, microservice orchestration and choreography. In this type of validation, we can generate test cases in semi-automated mode – we need to know the communication sequence, and this data should be provided by the developer. Using this data, we can produce communication graph and test the system according to this graph.

Load integration validation is aimed at checking the microservice correctness under automatic deployment, backup and transfer, as well as microservice orchestration and choreography. This type of validation can be done automatically. Unlike functional integration validation, this type of validation includes loading of communication channels to discover the maximum load the microservice can handle before failure.

Integration security validation is aimed at validation of the security of communications between microservices, as well as checking for the interception of messages from outside or inside by anyone, besides the message recipient. Microservice specific validation should also take into account the particular environment in which the microservices are applied.

IV. VALIDATION OF MJOLNIRR-BASED MICROSERVICES

We have developed a prototype of the Mjolnirr microservice platform [17]. This platform supports microservices, implemented using JVM-compatible languages and Java-based containerization. Integration with Mjolnirr will be the first step of our validation framework implementation. This step includes Java-specific methods of software validation, such as jUnit for component validation. Let's describe, how validation implementation steps that we have highlighted in Chapter 2 could be implemented on the basis of Mjolinrr platform.

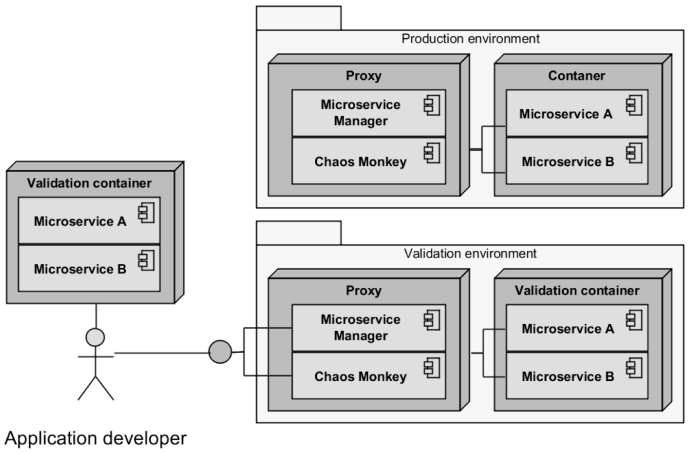

Figure 6. Validation of Mjolnirr-based microservices

TD2 (Derive test conditions) describes the testable aspects of components, in our case - microservices. In Mjolnirr, we can consider microservices and their calls as test conditions. As mentioned in Chapter 3, each microservice is isolated and autonomous and can be tested in a fully isolated JVM. Microservices are communicating using messages. Therefore, conditions, which determine quality of microservice system, are following: full isolation for component level, message communication for integration level and encapsulation for system level.

TD3 (Derive test coverage items) describes the attributes, derived from test conditions. Test conditions are microservices and their communications, therefore, test coverage items would be microservice methods and messages. Each microservice method and communication channel should be covered. Coverage items can be covered automatically or by application developer.

TD4 (Derive test cases) describes the test cases generation process. According to quality characteristics and ISO/IEC 29119-4 standard, we can decide, what test design techniques we should use and find exact technologies to implement those techniques. To perform a component-level testing, we can use only specification-based techniques, such as Boundary Value Analysis, Equivalence partitioning, Use case testing, Random testing and Scenario testing. In a case of component security testing, we can also scan microservice for typical vulnerabilities of the specific middleware, used in its infrastructure.

TD6 (Derive test procedures) describes the testing process. In Mjolnirr we use both automated and user-provided test cases, so we should cover all the coverage items, provided by microservice interface, and call all the test cases, provided by application developer. Test case can modify the microservice environment and state, therefore, microservice validation should be executed in an isolated environment, which should be purged after each set execution. As shown on figure 6, Mjolnirr has three sets of environments – special validation container for isolated component-level validation, validation environment for integration-level validation and production environment for end users. Validation environment and production environment has built-in Chaos Monkey service to ensure system stability. These three steps should be passed by each microservice to go to production.

To use those techniques, we need to have detailed microservice interface description, including input and output variables types, boundaries and syntax. Using this information, we can perform black box testing to each microservice in addition to the simple unit tests. Microservice interface definition described on figure 7.

```
{
  "name": "test_microservice",
  "inputs": [
    {
      "type": "integer",
      "min": 0,
      "max": 10
    },
    {
      "type": "string",
      "syntax": "he.?lo"
    }
  ],
  "outputs": [
    {
      "type": "integer",
      "min": 0,
      "max": 10
    }
  ],
  "input_connections": ["test_microservice2"],
  "output_connections": ["test_microservice3"]
}
```

Figure 7. Microservice interface description

In a case of integration testing we have knowledge about internal structure of the tested system. Therefore, we can use the testing techniques, mentioned before, and structure-based test design techniques, such as data flow testing, branch condition testing, etc. As shown on figure

5, microservice interface also describes input and output connections for each microservice, and we can use this knowledge to produce call graph and validate this structure.

Integration-level validation cannot be implemented on a container level, so this type of validation will be handled by Proxy node. Proxy will store the call graph, compare it with the desired structure, based on component interface. Also, Proxy will use different integration validation techniques, like Chaos Monkey [4] and random messaging, which are useful in a case of high load system.

V. CONLUSION

In this article, we described microservice testing features and validation techniques. In addition, specific test design techniques were mentioned. We have described the basics of microservice approach, provided and overview of methods and standards of software testing and validation. We described microservice validation features and proposed an approach to microservice validation. At last we proposed a brief overview of possible methods of implementation of microservice validation framework based on a Mjolnirr platform.

As a further development of this study, we will implement microservice validation framework, which will be able to perform automated and semi-automated validation to the microservices.

REFERENCES

[1] Bartolini C., Bertolino A., Elbaum S., Marchetti E. Whitening SOA testing. *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering.* 161—170.

[2] Brooks F.P. No Silver Bullet. Essence and accidents of software engineering. IEEE computer 20(4); 1987: 10-19.

[3] Calchado P. Building Products at SoundCloud—Part III: Microservices in Scala and Finagle. [Online] Available at: https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-3-microservices-in-scala-and-finagle [accessed 6.02.2015].

[4] Chaos Monkey. [Online] Available at: https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey [accessed 12.01.2015]

[5] Clemson, T. Testing Strategies in a Microservice Architecture. 2014. [Online] Available at: http://martinfowler.com/articles/microservice-testing/ [Accessed 10.01.2015].

[6] Ford N. Building Microservice Architectures. [Online] Available from: http://nealford.com/downloads/Building_Microservice_Architectures(Neal_Ford).pdf [accessed 13.02.2015]

[7] International Organization for Standardization (2014) 25010. Service Quality Requirements and Evaluation (SquaRE) – System and software Quality Model, 2014, Canada.

[8] International Organization for Standardization (2014) 29119. Software Testing, 2014, Canada.

[9] International Organization for Standardization (2015) 25011. Service Quality Requirements and Evaluation (SquaRE) – Service Quality Model, 2015, China.

[10] Jehan S., Pill I., Wotawa F. Functional SOA Testing Based on Constraints. *Proceedings of the 8th International Workshop on Automation of Software Test.* 33—39.

[11] Jones S. Microservices is SOA, for those who know what SOA is, 2014. [Online]. Available at: http://service-architecture.blogspot.ru/2014/03/microservices-is-soa-for-those-who-know.html. [Accessed: 17.11.2014].

[12] Kant N., Tonse T. Karyon: The nucleus of a Composable Web Service. [Online] Available at: http://techblog.netflix.com/2013/03/karyon-nucleus-of-composable-web-service.html [accessed 17.11.2014].

[13] Lewis J., Fowler M. Microservices. [Online] Available at: http://martinfowler.com/articles/microservices.html [accessed 17.11.2014].

[14] Prana: A Sidecar for your Netflix PaaS based Applications and Services. [Online] Available from: http://techblog.netflix.com/2014/11/prana-sidecar-for-your-netflix-paas.html [accessed 13.02.2015]

[15] Radchenko G., Mikhailov P., Savchenko D., Shamakina A., Sokolinsky L. Component-based development of cloud applications: a case study of the Mjolnirr platform. *Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia (CEE-SECR '14)*. Article 6, 10 pages.

[16] Rhoton J, Haukioja R. Cloud computing architected. [Tunbridge Wells, Kent]: Recursive Press; 2011.

[17] Savchenko D., Radchenko G. Mjolnirr: A Hybrid Approach to Distributed Computing Architecture and Implementation // CLOSER 2014. Proceedings of the 4th International Conference on Cloud Computing and Services Science (Barcelona, Spain 3-5 April, 2014), 2014. P. 445-450.

[18] Thones J. Microservices. Software, IEEE, Volume 24 Issue 3, 2015. P. 116-116.

[19] Tonse S. Microservices at Netflix. [Online] Available from: http://www.slideshare.net/stonse/microservices-at-netflix [accessed 6.02.2015].

[20] Varia J. Cloud architectures. White Paper of Amazon Web Services, 2008. [Online] Available at: https://media.amazonwebservices.com/AWS_Cloud_Architectures.pdf: 16. [accessed 17.11.2014].

Publication III

Savchenko D., Ashikhmin N., and Radchenko G.
Testing-as-a-Service approach for cloud applications
2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC), Shanghai, 2016, pp. 428-429.

2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing

Testing-as-a-Service Approach for Cloud Applications

Dmitry Savchenko
South Ural State University
System Programming Department
Chelyabinsk, Russia
savchenkodi@susu.ru

Nikita Ashikhmin
South Ural State University
System Programming Department
Chelyabinsk, Russia
ashikhmin.na@yandex.ru

Gleb Radchenko
South Ural State University
System Programming Department
Chelyabinsk, Russia
gleb.radchenko@susu.ru

ABSTRACT

Web-based systems test techniques developed rapidly over the past years. Continuous integration approach demanded rejection of manual testing methods, and transition to fully automated methods, covering all application from the source code of core services to the user interface and API correctness. Every year there are dozens of new approaches to cloud applications testing, many of which, despite their effectiveness, little known to ordinary developers and testers. We propose a Testing-as-a-Service solution to simplify the cloud applications testing by providing a catalog of applicable test techniques and automating testing processes.

CCS Concepts

•Software and its engineering → Cloud computing; Software testing and debugging;

1. INTRODUCTION

It is believed that writing documentation and writing tests - the two things software developers hate. Indeed, high-quality testing of complex applications, such as cloud services, requires competence in testing automation [1], knowledge about the state-of-the-art testing techniques (like Twitter Diffy, Chaos Monkey, Fuzzy Testing, etc.) and skills of applying these techniques. Furthermore, development of the cloud service testing methodology is often become a separate complex project that should be conduct concurrently with the main software development project [2]. For all of these reasons, software developers and project managers often pay insufficient attention to the comprehensive testing, sacrificing the quality for the sake of faster product delivery.

We propose a Testing-as-a-Service solution to simplify the cloud applications testing by providing a catalog of applicable test techniques and automating testing processes. Three main features of the proposed service can be distinguished.

- *Testing methods catalog:* Providing users a catalog of testing methods available for their applications. Based on the features of the application under test (like programming language, development framework, application type, etc.), the testing service offers to a user a set of ready-to-use suitable testing methods.
- *Extensibility:* each testing method is implemented as a separate microservice, easily integrated to the catalog of existing testing methods.
- *Pay-as-you-go:* using the cloud approach to implement the software testing provides to the user the combination of cost flexibility and built-in resource scalability.

UCC '16 December 6–9, 2016, Shanghai, China

DOI: 10.475/123_4

2. SYSTEM OVERVIEW AND SERVICE ARCHITECTURE

According to the features, defined in the introduction, we decided to implement cloud applications testing service as a web-application. Two actors are interacting with the testing service: tester and test techniques developer. **Tester** uses the service to test a cloud application. A tester can browse a catalog of available test techniques, create test projects, define test objectives, run tests and obtain test results. **Test techniques developer** implements test techniques in the testing service.

The interaction of the actors with the testing service is performed using the following key entities, defining features of the application under test and testing process.

Application attribute describes a feature of the cloud application under test. For example, a tester can define programming language, basic application framework or database type as an application attribute value. Each application attribute is associated with a set of test techniques, which provide testing for the applications with such attribute value.

Test technique is an atomic testing step. It aims the testing of the one single system aspect (like unit-testing, API interface testing, load testing, etc.).

Test objective is a set of test techniques focused on a testing of some aspect of the cloud application (e.g. testing of a specific component, interface, performance or stress testing and so on). While defining the test objective, the tester provides a set of cloud application properties (application attribute values) so the testing service could offer him a list of applicable test techniques. The tester picks and configures test techniques for the test objective, providing such information as source code repository URL, web server URL, test cases files, etc. When all test techniques are configured, the tester can run the test objective. When the test objective execution is finished, the tester can get the test report,

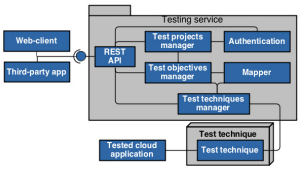

Figure 1: Testing service architecture

which contains detailed information about passed and failed tests.

Test project is a set of test objectives that represent a comprehensive cloud application testing solution. While creating a test project, a tester defines a set of static (configured once per project, like source code repository URL) and dynamic (must be set on each run) parameters. Test project parameters are passed to underlying test objectives and test techniques when the test project is initializing.

We can highlight the following components in the cloud applications testing service (see fig. 1):

- *Authentication and authorization* component identifies users, their roles, and permissions. It provides authentication through the login-password pair, SSH keys, OAuth accounts and authentication tokens (for third-party API users).
- *Test projects, objectives and techniques managers* provide Create, Read, Update, Delete (CRUD) actions for the Test projects, Test objectives, and Test techniques entities respectively.
- *Mapper* associates application attributes to the corresponding testing techniques.
- *Test technique* is independent service, implementing common REST interface providing test configuration, execution, and results delivery.

Let's describe the process of test configuration and execution (see fig. 2):

- The tester passes the authentication, creates a test project and a test objective, defining application attributes of the cloud application under test;
- According to provided application attributes, testing service offers a list of testing techniques that can be implemented for the application;
- The tester chooses test techniques he want to implement, defines parameters and sets the execution order of the test techniques;
- To initiate a testing process, the tester provides dynamic parameters of the test, executes the test project and receive the testing results.

3. IMPLEMENTATION AND FUTURE WORK

Testing service was implemented using Ruby on Rails framework and deployed to Heroku cloud platform for evaluation (available on http://cloudrift.x01d.com). It provides a REST API and a web interface for projects, objectives and test techniques configuration. Three test techniques were implemented - *API testing* using Ruby and Airborne; *load testing* using Ruby, Apache Benchmark, and ABCrunch; and *UI testing* for the Facebook React Framework using Node.JS and Carte Blanche.

Let us describe test objective initialization process. First, the tester defines static parameters for the created objective. In a case of API testing, he provides the cloud application domain. Next, the tester fills the tests list by choosing the "API testing" test technique. For each test technique, the tester provides API method path, HTTP verb, test parameters and excepted outputs as JSON. Testing service sends those parameters to the test technique service, and calls the API according to the provided parameters and matches the output with the expected one.

As a further development, we are going to enhance test techniques catalog, increase the number of covered tests techniques; implement automatic application attributes detection for the popular frameworks and programming languages; implement desktop and mobile applications testing support.

4. ACKNOWLEDGMENTS

The reported study was partially supported by RFBR, research project No. 14-07-00420-a and by Grant of the President of the Russian Federation No. MK-7524.2015.9

5. REFERENCES

[1] M. Rahman and J. Gao. A Reusable Automated Acceptance Testing Architecture for Microservices in Behavior-Driven Development. In *SOSE 2015 Symposium Proceedings,* pages 321-325. IEEE, 2015.

[2] D. Savchenko and G. Radchenko. Microservices validation: Methodology and implementation. In *CEUR Workshop Proceedings*, pages. 21–28. 2015, vol. 1513.

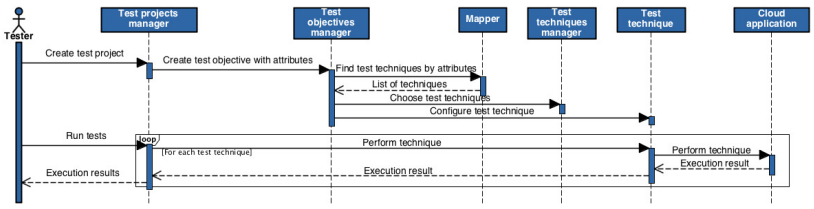

Figure 2: Configuration and testing process

Publication IV

Savchenko D., Radchenko G., Hynninen T., and Taipale O.
Microservice test process: Design and Implementation
International Journal of Information Technologies and Security (IJITS),
ISSN 1313-8251, vol. 10, No 3, 2018, pp. 13-24.

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@lut.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].

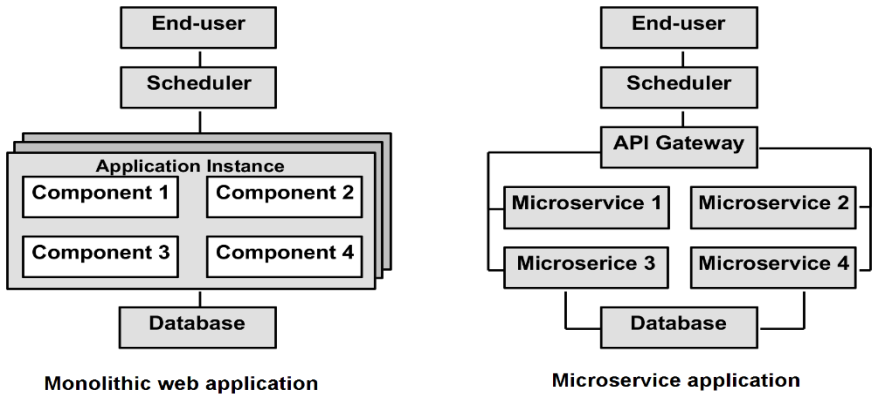

Fig 1. Differences between monolithic and microservice architectural styles

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.

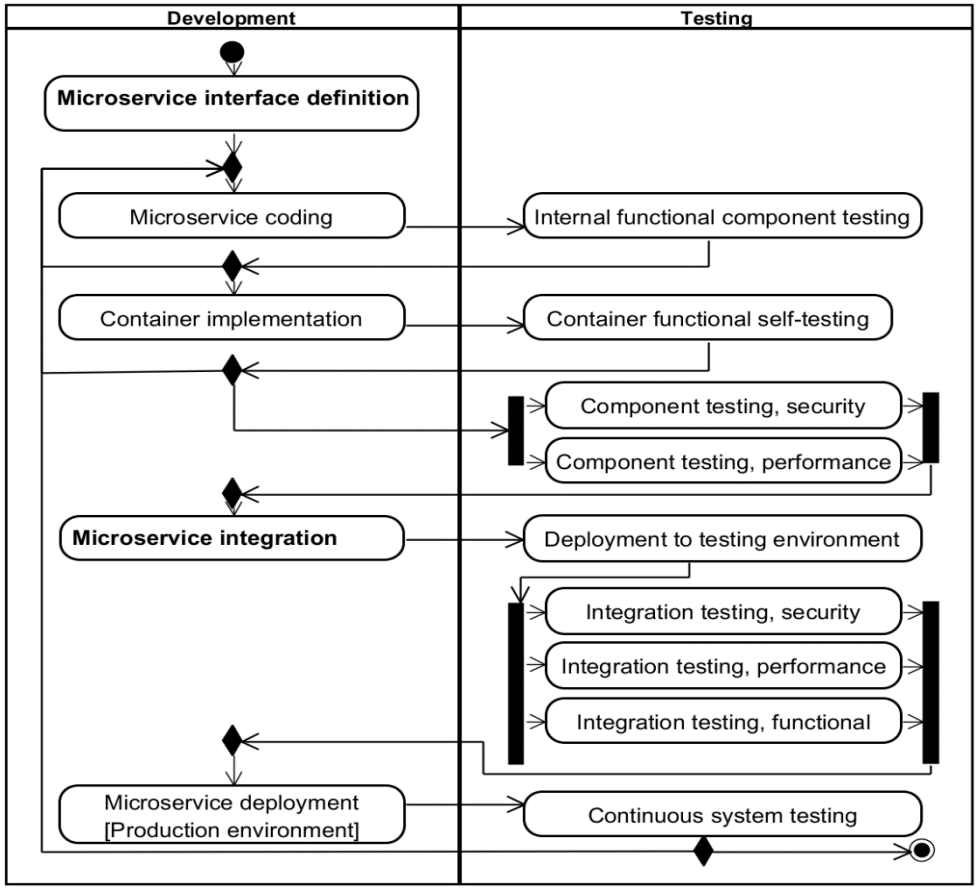

Fig. 2. Microservice test process

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.

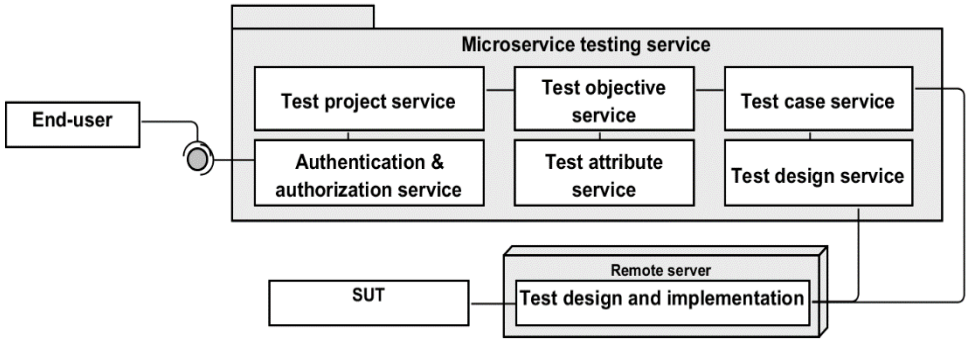

Fig. 3. Microservice testing service architecture

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.

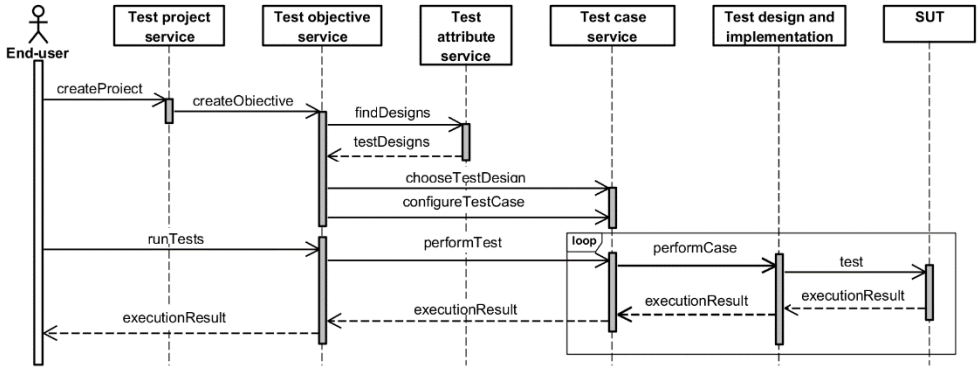

Fig. 4. Microservice testing service sequence diagram

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.

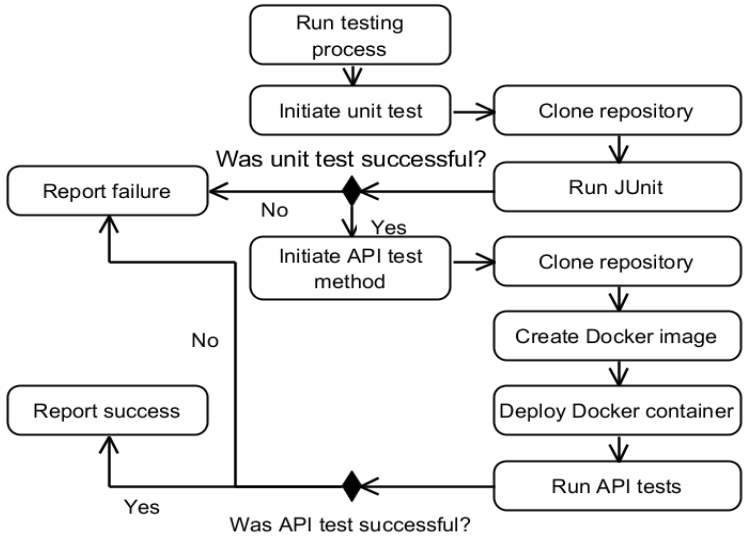

Fig. 5. Workflow in the evaluation example

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

Publication V

Savchenko D., Hynninen T., and Taipale O.
Code quality measurement: case study
2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, 2018, pp. 1455-1459.

MIPRO 2018, May 21-25, 2018, Opatija Croatia

Code quality measurement: case study

D. Savchenko*, T. Hynninen* and O. Taipale*
* Lappeenranta University of Technology, Lappeenranta, Finland
dmitrii.savchenko@lut.fi, timo.hynninen@lut.fi, ossi.taipale@lut.fi

***Abstract* - As it stands, the maintenance phase in the software lifecycle is one of the biggest overall expenses. Analyzing the source code characteristics and identifying high-maintenance modules is therefore necessary. In this paper, we design the architecture for a maintenance metrics collection and analysis system. As a result, we present a tool for analyzing and visualizing the maintainability of a software project.**

Keywords - maintenance, code quality

I. INTRODUCTION

Maintenance and upkeep is a costly phase of software life cycle. It has been estimated that maintenance can reach up to 92% of total software cost [1]. Code quality can be analyzed using various existing metrics, which can give an estimate on the maintainability of software. There are several tools and frameworks for assessing the maintainability characteristics of a project. Many tools are included in integrated development environments (IDEs), such as Eclipse metrics [2], JHawk [3] or NDepend [4]. As such the existing tools are specific to platform and programming language, providing quality analysis during development. Considering maintenance also includes activities post-release of a software product, it would be beneficial to perform quality measurement also in the maintenance and upkeep phase of life cycle.

One solution to the post-release monitoring are online data gathering probes, which can be inserted into production code to gather runtime performance data. In order to establish and sustain a commitment for maintenance measurement this work introduces a design for data collection and storage. In this paper we present an architecture for systematically collecting code metrics for maintenance. Additionally, the visualization and analysis of the metrics are explored.

In this study we will focus on the analysis of web-applications. This delimitation is due to the collection of runtime metrics as well as static metrics. The focus on web-applications provides a reasonably standardized measurement interface for runtime performance through the browser's web API. In this paper we also propose the design and implementation of the system called Maintain. The probes for gathering metrics in the system are implemented in both JavaScript and Ruby programming languages.

Rest of the paper is structured as follows. In Section 2, related work in analyzing software maintainability is introduced. Sections 3 and 4 presents the architecture and our implementation for a metrics collection and analysis system, which is main contribution of this work. Evaluation of the system's performance and utility is presented in Section 5. Finally, discussion and conclusions are given in Section 4.

This study was funded by the Technology Development center of Finland (TEKES), as part of the .Maintain project (project number 1204/31/2016).

II. RELATED RESEARCH

Software maintenance, as defined by ISO 14764 standard, is the "the totality of activities required to provide cost-effective support to a software system", consisting of activities both during development and post-release [5]. The analysis of software maintainability is by no means a novel concept. Motogna et al. [6] presented an approach for assessing the change in maintainability. In [6], metrics were developed based on the maintainability characteristics in the ISO 25010 software quality model [7]. The study presents how different object oriented metrics affect the quality characteristics.

A study by Kozlov et al. [8] distinguished that particular code metrics (data variables declared, McClure Decisional Complexity) have strong correlations with the maintainability of a project. In the work, the authors analysed the correlation between maintainability and the quality attributes of a Java-project.

In the study by Heitlager et al. [9] a practical model for maintainability is discussed. The study discusses the problems of measuring maintainability, particularly with expressing maintainability as a single metric (Maintainability index).

Studies where different evaluation methods are combined in order to get a more thorough view on the maintainability of a project have been conducted during the past decade. For example, Yamashita [10] combined benchmark-based measures, software visualization and expert assessment. In a similar vein, Anda [11] assessed the maintainability of a software system using structural measures and expert assessment. In general, these studies suggest that visualization systems providing developers and project managers with an analysis of the health of a software project can help distinguish problematic program components, and thus help in the maintenance efforts of software.

III. ARCHITECTURE

Maintain system architecture is presented at the figure 1. System consists of the following components:

- *Probe* is a program that gathers some valuable data from the software (static or dynamic). Each probe should have an associated analyzer;
- *Data Storage* – data storage that stores the raw data from the probes. It also has REST interface that receives the data from the probes;
- *Analyzer* is a program that gets the raw data from the associated probe and creates a report, based on this data;
- *Report Storage* – data storage that stores reports from analyzers;
- *Report Visualizer* is a component that creates a visual representation of the report.

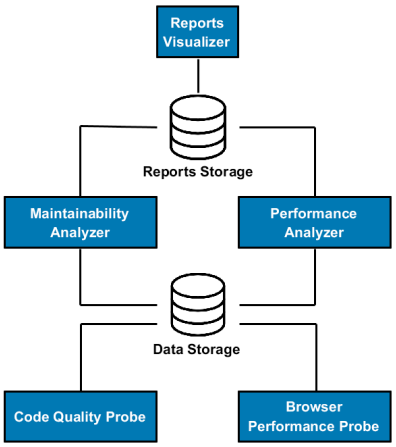

Figure 1. System architecture

Workflow of the system is centered around the Data Storage. Generally, it looks like this:

- Probes gather the information from the source code, it might be some static analysis results or dynamic performance data;
- Gathered and normalized data is sent to the Data Storage. Probe can have different data types, data structure is defined by analyzer;
- When new data is received by Data Storage, the associated analyzer is called. It requests the data from the Data Storage, produces report (object, that contains current status of the analyzed application aspect and a set of time series for the end user);

When end user requests the report, Reports Visualizer generates a visual representation of the time series, that were created by analyzers.

IV. Implementation

Maintain system was implemented using Ruby on Rails framework and hosted on Heroku cloud platform. Project details page is shown on Figure 3. This page provides the information about the current state of the project, that is described as a set of 8 scores, based on quality characteristics, described in ISO/IEC 25010 [7]. Those scores are visualized as a polar chart with 8 axis for each quality characteristic respectively. Score calculation is based on the report statuses – each report has an associated probe, and each probe has a set of associated quality characteristics. Quality characteristics are set by the project administrator.

System class structure is organized as pictured in figure 2. As system gathers the data using REST API, it is generally impossible to predefine all possible probes and probe types and set their quality characteristics in advance. That's why we decided to let user define the quality characteristics for probe when it is created or modified. Result score is based on statuses of last reports for each probe respectively.

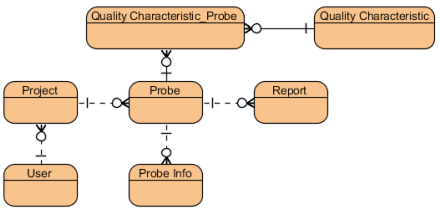

Figure 2. System entity-relationship diagram

A. Probes

As a case study, we have implemented four probes: HAML, JavaScript and Ruby code quality probes, and browser performance probe. JavaScript and Ruby code quality probes are based on maintainability index, which is calculated using the following formula:

```
maintainability = 171 -
            (3.42*Math.log(effort))-
            (0.23*Math.log(cyclomatic))-
            (16.2*Math.log(loc))
```

HAML maintainability index uses recursive formula, based on linter report:

```
Maintainability = a*maintainability
```

where **a** is 0.9 for linter error and 0.99 for linter warning

Code quality probes produce the following data for Data Storage:

```
{
    maintainability: M,
    revision: R,
    datetime: D,
    modules: Ms
}
```

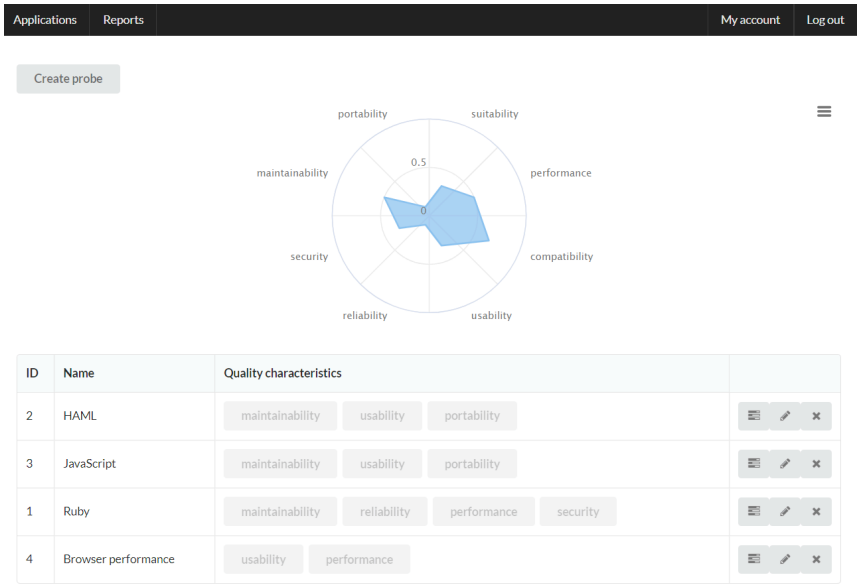

Figure 3. Project page example

where M is average maintainability index for the whole project, R is current Git revision, D is current date and time, and Ms is a list of maintainability index for project files and their names. Browser performance probe generates report in different format:

```
{
    page: P,
    timing: T,
    datetime: D
}
```

Where P is an URL of the current page (without query), T is the time between page load start time and DOM ready event time in milliseconds, and D is current date and time.

B. *Analyzers*

Currently we have implemented two different analyzers - maintainability analyzers for Ruby, JavaScript and HAML probes, and performance analyzer for browser performance probe. Workflow for maintainability analyzer works is described below:

- Data from Data Storage is grouped by days, maintainability index for each day calculated as a median of indices for day. If no data presented for day, analyzer sets the value for the previous day (fallback for weekends);
- List of maintainability indices are smoothed using exponential moving average method, those values are used as a time series for visualizer;
- Linear regression for last five days is used as a status of the project source code quality: if it is less than zero, then code quality is bad.

Workflow for browser performance analyzer is different:

- Performance data is grouped by five minutes, value for each section is calculated as a 95th percentile of all values for section;
- If values for all sections are less than 2 seconds, then browser performance is good.

V. MAINTAIN SYSTEM USAGE EXAMPLE

Maintain system was evaluated using a proprietary web application, that was implemented using Ruby on Rails as a backend, and CoffeeScript on top of React.JS as a frontend. This project is on maintenance phase, so we decided to analyze historical data and compare Maintenance system results with the feedback from the project manager, who managed the analyzed project. Application was used by 5 administrators and about 10000 users. Maintenance system was deployed in Heroku cloud, while probes were running on local PC, that had 1.8 GHz 2-core CPU and 4 Gb RAM. We gathered the code quality information for all the previous commits to make picture more consistent.

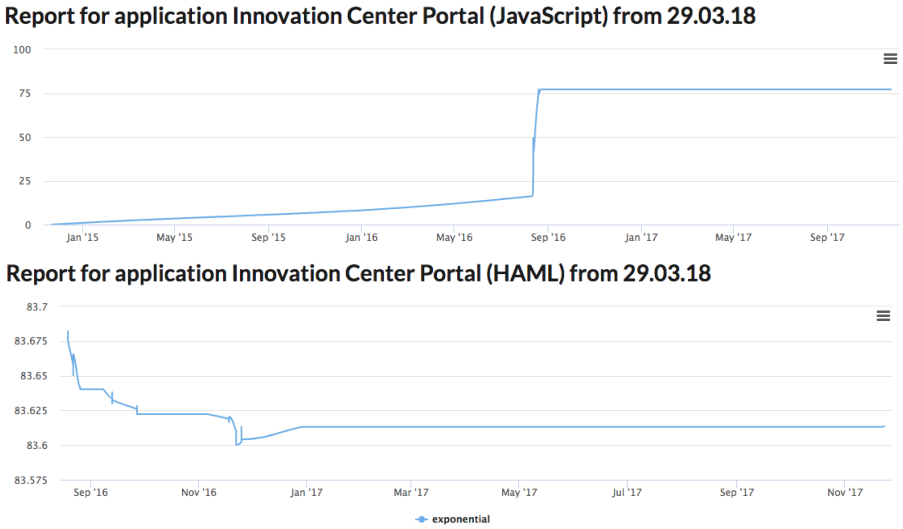

Figure 4. Project code quality measurements

Figure 3 illustrates the general 'health' of the analyzed application at the last Git revision at Master branch. Figure 4 shows the JavaScript (CoffeeScript) and HAML code quality. The project was started as a pure backend solution, while frontend development started at the beginning of September 2016. As shown in the graph, HAML code quality was decreasing from September 2016, until December 2016, then it was stable. This behavior can be explained by a deadline of the project, that was at the end of the year 2016. After the deadline, the project active development stopped. Project manager evaluated the results and stated, that such an 'early warning' system could notify the team and save some development resources.

VI. Discussion and Conlusion

The objective of this study was to facilitate the systematic collection and analysis of maintenance metrics, in order to reduce the effort required in the maintenance phase of software already during development. To realize the goal we designed and implemented an architecture for a system which can be used to collect both static and runtime metrics of a software project. We then implemented analysis tools to visualize these metrics, and display the most high-maintenance modules in a project repository.

The novelty of the presented work is the extendibility and modularity of the architecture. The architecture is not platform specific. New probes and corresponding analyzers can be added at any stage, using the REST API with any programming language or platform. The data storage and reporting system provide a common interface for the systematic collection of quality metrics, allowing the developers of a project to establish and sustain a commitment for quality measurement.

Providing a platform to establish the measurement commitment is important, because previous research shows that the quality assurance and testing practices of developers do not necessarily line up with measurement possibilities distinguished in academic research. For example, the recent study by Garousi and Felderer distinguishes that the industry and academia have different focus areas on software testing [12]. Likewise, Antinyan et al. show in [13] that existing code complexity measures are poorly used in industry. In this work, we used the maintainability index as an indicator for code quality, as it has been used in both academia and industry. In future, we should work on evaluating whether quality metrics presented in academic publications could be implemented into our system as probes providing reliable measurements.

Additionally, in future work we aim to develop more measurement probes in the system. We should evaluate the different metrics to distinguish which measurements provide the most useful information about software maintainability.

References

[1] "The Four Laws of Application, Total Cost of Ownership." Gartner, Inc., 2012.

[2] Eclipse Metrics Plug-in, http://sourceforge.net/projects/metrics. (accessed 5th Feb 2018).

[3] JHawk, http://www.virtualmachinery.com/jhawkprod.htm. (accessed 5th Feb 2018).

[4] NDepend "http://www.ndepend.com", (accessed 5th Feb 2018).

[5] ISO/IEC, "ISO/IEC 14764: Software Engineering - Software Life Cycle Processes - Maintenance." 2006.

[6] S. Motogna, A. Vescan, C. Serban, and P. Tirban, "An approach to assess maintainability change," in 2016 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), 2016, pp. 1–6.

[7] ISO/IEC, "ISO/IEC 25010: Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models." 2011.

[8] D. Kozlov, J. Koskinen, J. Markkula, and M. Sakkinen, "Evaluating the Impact of Adaptive Maintenance Process on Open Source Software Quality," in First International Symposium on

Empirical Software Engineering and Measurement (ESEM 2007), 2007, pp. 186–195.

[9] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the, 2007, pp. 30–39.

[10] A. Yamashita, "Experiences from performing software quality evaluations via combining benchmark-based metrics analysis, software visualization, and expert assessment," in 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015, pp. 421–428.

[11] B. Anda, "Assessing software system maintainability using structural measures and expert assessments," in Software Maintenance, 2007. ICSM 2007. IEEE International Conference on, 2007, pp. 204–213.

[12] V. Garousi and M. Felderer, "Worlds Apart: Industrial and Academic Focus Areas in Software Testing," IEEE Software, vol. 34, no. 5, pp. 38–45, 2017.

[13] V. Antinyan, M. Staron, and A. Sandberg, "Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time," Empirical Software Engineering, pp. 1–31, 2017.

ACTA UNIVERSITATIS LAPPEENRANTAENSIS

830. BELONOGOVA, NADEZDA. Active residential customer in a flexible energy system - a methodology to determine the customer behaviour in a multi-objective environment. 2018. Diss.

831. KALLIOLA, SIMO. Modified chitosan nanoparticles at liquid-liquid interface for applications in oil-spill treatment. 2018. Diss.

832. GEYDT, PAVEL. Atomic Force Microscopy of electrical, mechanical and piezo properties of nanowires. 2018. Diss.

833. KARELL, VILLE. Essays on stock market anomalies. 2018. Diss.

834. KURONEN, TONI. Moving object analysis and trajectory processing with applications in human-computer interaction and chemical processes. 2018. Diss.

835. UNT, ANNA. Fiber laser and hybrid welding of T-joint in structural steels. 2018. Diss.

836. KHAKUREL, JAYDEN. Enhancing the adoption of quantified self-tracking wearable devices. 2018. Diss.

837. SOININEN, HANNE. Improving the environmental safety of ash from bioenergy production plants. 2018. Diss.

838. GOLMAEI, SEYEDMOHAMMAD. Novel treatment methods for green liquor dregs and enhancing circular economy in kraft pulp mills. 2018. Diss.

839. GERAMI TEHRANI, MOHAMMAD. Mechanical design guidelines of an electric vehicle powertrain. 2019. Diss.

840. MUSIIENKO, DENYS. Ni-Mn-Ga magnetic shape memory alloy for precise high-speed actuation in micro-magneto-mechanical systems. 2019. Diss.

841. BELIAEVA, TATIANA. Complementarity and contextualization of firm-level strategic orientations. 2019. Diss.

842. EFIMOV-SOINI, NIKOLAI. Ideation stage in computer-aided design. 2019. Diss.

843. BUZUKU, SHQIPE. Enhancement of decision-making in complex organizations: A systems engineering approach. 2019. Diss.

844. SHCHERBACHEVA, ANNA. Agent-based modelling for epidemiological applications. 2019. Diss.

845. YLIJOKI, OSSI. Big data - towards data-driven business. 2019. Diss.

846. KOISTINEN, KATARIINA. Actors in sustainability transitions. 2019. Diss.

847. GRADOV, DMITRY. Experimentally validated numerical modelling of reacting multiphase flows in stirred tank reactors. 2019. Diss.

848. ALMPANOPOULOU, ARGYRO. Knowledge ecosystem formation: an institutional and organisational perspective. 2019. Diss.

849. AMELI, ALIREZA. Supercritical CO2 numerical modelling and turbomachinery design. 2019. Diss.

850. RENEV, IVAN. Automation of the conceptual design process in construction industry using ideas generation techniques. 2019. Diss.

851. AVRAMENKO, ANNA. CFD-based optimization for wind turbine locations in a wind park. 2019. Diss.

852. RISSANEN, TOMMI. Perspectives on business model experimentation in internationalizing high-tech companies. 2019. Diss.

853. HASSANZADEH, AIDIN. Advanced techniques for unsupervised classification of remote sensing hyperspectral images. 2019. Diss.

854. POPOVIC, TAMARA. Quantitative indicators of social sustainability applicable in process systems engineering. 2019. Diss.

855. RAMASAMY, DEEPIKA. Selective recovery of rare earth elements from diluted aqueous streams using N- and O –coordination ligand grafted organic-inorganic hybrid composites. 2019. Diss.

856. IFTEKHAR, SIDRA. Synthesis of hybrid bio-nanocomposites and their application for the removal of rare earth elements from synthetic wastewater. 2019. Diss.

857. HUIKURI, MARKO. Modelling and disturbance compensation of a permanent magnet linear motor with a discontinuous track 2019. Diss.

858. AALTO, MIKA. Agent-based modeling as part of biomass supply system research. 2019. Diss.

859. IVANOVA, TATYANA. Atomic layer deposition of catalytic materials for environmental protection. 2019. Diss.

860. SOKOLOV, ALEXANDER. Pulsed corona discharge for wastewater treatment and modification of organic materials. 2019. Diss.

861. DOSHI, BHAIRAVI. Towards a sustainable valorisation of spilled oil by establishing a green chemistry between a surface active moiety of chitosan and oils. 2019. Diss.

862. KHADIJEH, NEKOUEIAN. Modification of carbon-based electrodes using metal nanostructures: Application to voltammetric determination of some pharmaceutical and biological compounds. 2019. Diss.

863. HANSKI, JYRI. Supporting strategic asset management in complex and uncertain decision contexts. 2019. Diss.

864. OTRA-AHO, VILLE. A project management office as a project organization’s strategizing tool. 2019. Diss.

865. HILTUNEN, SALLA. Hydrothermal stability of microfibrillated cellulose. 2019. Diss.

866. GURUNG, KHUM. Membrane bioreactor for the removal of emerging contaminants from municipal wastewater and its viability of integrating advanced oxidation processes. 2019. Diss.

867. AWAN, USAMA. Inter-firm relationship leading towards social sustainability in export manufacturing firms. 2019. Diss.

ISBN 978-952-335-414-2
ISBN 978-952-335-415-9 (PDF)
ISSN-L 1456-4491
ISSN 1456-4491
Lappeenranta 2019