Does Migrating a Monolithic System to Microservices Decrease the Technical Debt?

Valentina Lenarduzzi, Francesco Lomio, Nyyti Saarimäki, Davide Taibi

CloWeE - Cloud and Web Engineering group
Tampere University, Finland

Abstract

Background. The migration from a monolithic system to microservices requires a deep refactoring of the system. Therefore, such a migration usually has a big economic impact and companies tend to postpone several activities during this process, mainly to speed up the migration itself, but also because of the demand for releasing new features.
Objective. We monitored the technical debt of an SME while it migrated from a legacy monolithic system to an ecosystem of microservices. Our goal was to analyze changes in the code technical debt before and after the migration to microservices.
Method. We conducted a case study analyzing more than four years of the history of a twelve-year-old project (280K Lines of Code) where two teams extracted five business processes from the monolithic system as microservices. For the study, we first analyzed the technical debt with SonarQube and then performed a qualitative study with company members to understand the perceived quality of the system and the motivation for possibly

Email addresses: `valentina.lenarduzzi@tuni.fi` (Valentina Lenarduzzi), `francesco.lomio@tuni.fi` (Francesco Lomio), `nyyti.sarimaki@tuni.fi` (Nyyti Saarimäki), `davide.taibi@tuni.fi` (Davide Taibi)

postponed activities.

Results. The migration to microservices helped to reduce the technical debt in the long run. Despite an initial spike in the technical debt due to the development of the new microservice, after a relatively short period of time the technical debt tended to grow slower than in the monolithic system.

Keywords: Technical Debt, Architectural Debt, Code Quality, Microservices, Refactoring

1. Introduction

Migration to microservices has become very popular over the last years. Companies migrate for different reasons, for example because they expect to improve the quality of their system or to facilitate software maintenance [1].

Companies commonly adopt an initial migration strategy to extract components from their monolithic system as microservices, making use of simplified microservices patterns [1][2]. As an example, it is common for companies to initially connect the microservices directly to the legacy monolithic system and not to adopt message buses. However, when the system starts to grow in complexity, they usually start re-architecting their system, considering different architectural patterns [1][2][3]. The migration from a monolithic system to microservices is commonly performed on systems that are being actively developed. Therefore, in several cases, the development of new features is prioritized over refactoring of the code, which generates technical debt (TD) every time an activity is postponed [4][5] and increases the cost of software maintenance.

Companies migrate to microservices to facilitate maintenance [1]. However, recent surveys have confirmed that maintenance costs increase after migration [1][6]. Therefore, the goal of this paper is to understand whether the overall code TD of a system increases or decreases after migration to microservices, and whether the type of TD varies after the migration.

In this work, we report a case study where we monitored the code TD of an SME (small and medium-sized enterprise) that migrated their legacy monolithic system to an ecosystem of microservices. The company was interested in evaluating their TD with SonarQube[1], which is one of the most commonly used tools for analyzing code TD. They asked us to monitor the evolution of TD in a project they are developing, during the migration to microservices. The analysis focused on the three types of TD proposed by SonarQube: reliability remediation cost (time to remove all the issues that can generate faults), maintainability remediation cost (time to remove all the issues that increase the maintenance effort), and security vulnerability remediation cost (time to remove all the security issues). Moreover, we implemented a qualitative study by conducting a focus group with two development teams, the software architect, and the product manager. The goal was to deeply understand the causes of the changes in the distribution of the types of TD issues and the motivations for any postponed activities.

To the best of our knowledge, only a limited number of studies have investigated the impact of postponed activities on TD, especially in the context of microservices [7]. This work will help companies to understand how TD grows and changes over time while at the same time opening up new avenues for future research on the analysis of TD interest.

This paper is structured as follows: Section 2 briefly introduces the background and related work on microservices and TD. In Section 4, we

[1]SonarQube: www.sonarqube.org

present the case study design, defining the research questions and describing the study context with the data collection and the data analysis protocol. In Section 5, we show the results we obtained, followed by a discussion of them in Section 6. In Section 7, we identify the threats to the validity of our study, and in Section 8, we draw conclusions and provide an outlook on possible future work.

2. Background

In this Section, we will first describe the background on microservices and technical debt (TD). Moreover, we will describe SonarQube and the method adopted to calculate TD.

2.1. Microservices

Microservice architecture has become more and more popular over the last years. Microservices are small, autonomous, and independently deployed services, with a single and clearly defined purpose [8].

The independent deployment provides a lot of advantages. They can be developed in different programming languages, they can scale independent of other services, and they can be deployed on the hardware that best suits their needs. Moreover, because of their size, they are easier to maintain and more fault-tolerant since the failure of one service will not break the whole system, which is possible in a monolithic system. In addition, as microservices are cloud-native applications, they support the IDEAL properties: Isolation of state, Distribution, Elasticity, Automated management, and Loose Coupling [8]. Moreover, microservices propose vertical decomposition of applications into a subset of business-driven services. Each service can be developed, deployed, and tested independently by a different development team using different technology stacks. The development responsibility of a microservice belongs to a single team, which is in charge of the whole development process, including deploying, operating, and upgrading the service when needed.

2.2. Technical Debt

The concept of TD was introduced for the first time in 1992 by Cunningham as *"The debt incurred through the speeding up of software project development which results in a number of deficiencies ending up in high maintenance overheads"* [4]. McConnell [9] improved the definition of TD to *"A design or construction approach that's expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now (including increased cost over time)"*. In 2016, Avgeriou et al. [10] defined TD as *"A collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. TD presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability"*.

Different approaches and strategies have been suggested for evaluating TD. Nugroho et al. [11] proposed an approach for quantifying debt based on the effort required to fix TD issues, using data collected from 44 systems as their basis. Seaman et al. [12] proposed a TD management framework that formalizes the relationship between cost and benefit in order to improve software quality and support the decision-making process during maintenance activities. Zazworka et al. [13] investigated automated identification of TD. They asked developers to identify TD items during the development process and compared the manual identification with the results from an automatic detection. Zazworka et al. [14] examined source code analysis techniques and tools to identify code debt in software systems, focusing on TD interest and TD impact on increasing defect- and change-proneness. They applied four TD identification techniques (code smells, automatic static analysis issues, grime buildup, and modularity violations) on 13 versions of the Apache Hadoop open-source software project. They collected different metrics, such as code smells and code violations. The results showed a positive correlation between some metrics and defect- and change-proneness, such as Dispersed Coupling and modularity violations. Guo et al. [15] investigated the TD cost of applying a new approach to an ongoing software project. They found higher start-up costs, which decreased over time.

2.2.1. Technical Debt Measurement

Different commercial and open-source tools can be used to measure TD, including CAST[2], Coverity Scan[3], SQUORE[4], Designite[5], and others.

In this work, as required by our case company, we adopted SonarQube, as it is one of the most commonly used TD measurement tools, adopted by more than 120K users[6]. Moreover, SonarQube is also open-source, while the other well-known competitors have a commercial license.

SonarQube calculates several metrics such as number of lines of code and code complexity, and verifies the code's compliance against a specific

[2]CAST Software https://www.castsoftware.com/ Last Access: August 2019

[3]Coverity Scan. https://scan.coverity.com. Last Access: August 2019

[4]SQUORE. https://www.squoring.com/en/produits/squore-software-analytics/ Last Access: August 2019

[5]Designite. http://designite-tools.com Last Access: August 2019

[6]SonarQube. http://www.sonarqube.org Last Access: August 2019

set of "coding rules". If the analyzed source code violates a coding rule or if a metric is outside a predefined threshold (also called "quality gate"), SonarQube generates an "issue". Issues are problems that generate TD and therefore should be solved.

SonarQube has separate rule sets for the most common development languages such as Java, Python, C++, and JavaScript. For example, SonarQube version 7.5 includes more than 500 rules for Java.

Rules are classified as being related to reliability, maintainability, or security of the code. Reliability rules, also named "bugs", create TD issues that "represent something wrong in the code" and that will soon be reflected in a bug. Security rules, also called "vulnerabilities" or "security hotspots", represent issues that can be exploited by a hacker or that are otherwise security-sensitive. Maintainability rules or "code smells" are considered "maintainability-related issues" in the code that decrease code readability and modifiability. It is important to note that the term "code smells" adopted in SonarQube does not refer to the commonly known code smells defined by Fowler et al. [16], but to a different set of rules. SonarQube claims that zero false-positive issues are expected from the reliability and maintainability rules, while security issues may contain some false-positives[7]. The complete list of rules is available online[8].

SonarQube calculates TD using the SQALE method [17]. It is an ISO 9126 compliant method developed by DNV ITGS France [18]. The method is based on five categories [19]:

[7]SonarQube Rules: https://docs.sonarqube.org/display/SONAR/Rules Last Access: June 2019

[8]https://rules.sonarsource.com/java Last Access: August 2019

- Robustness: Application stability and ability to recover from failures.
- Performance efficiency: application responsiveness and usage of resources.
- Security: Systems ability to protect confidential information and prevent unauthorized intrusions.
- Transferability: Software understandability and especially the "ease with which a new team can understand the software and become productive".
- Changeability: Measures software adaptability and modifiability.

SonarQube calculates TD as:

$$\begin{aligned} TD_{(person-days)} =& cost\ to\ fix\ issues + cost\ to\ fix\ duplications + cost\ to\ comment \\ & public\ API + cost\ to\ fix\ uncovered\ complexity + cost\ to\ bring \\ & complexity\ below\ threshold \end{aligned} \tag{1}$$

SonarQube also identifies TD density as:

$$TDDensity = \frac{TD_{(person-days)}}{KLOC} \tag{2}$$

The previous formula is then instantiated three times to calculate:

- *Technical Debt*, considering maintainability-related rules (tagged as "code smells") that are supposed to increase the change-proneness of the infected code.

  Technical Debt was also called "SQUALE index" until SonarQube version 7.7. Starting from SonarQube version 7.8, it has been called

"Mantainability remediation effort". In this work, we refer to this type of TD as TD_M.

- *Reliability Remediation Effort*, considering reliability-related rules (rules tagged as "bug") that are supposed to increase the bug-proneness of the infected code. In this work, we refer to this type of TD as TD_R.
- *Security Remediation Effort*, considering rules related to security vulnerabilities (tagged as "vulnerability"). In this work, we refer to this type of TD as TD_S.

3. Related Work

In this Section, we report on the most relevant related work on microservices migration and TD.

Companies migrate to microservices in order to ease their software development by improving maintainability and decreasing delivery time [1][6]. However, migration to microservices is not an easy task. Companies commonly start this migration without having any experience with microservices, and only in few cases do they hire a consultant to support them during the migration [1][6].

Several processes exist that can be adopted while migrating to microservices [1][20]. Fritzsch et al. [20] analyzed works from the literature and classified the reported refactoring approaches used to migrate from monolithic systems to microservices. They highlight that not all of the existing refactoring approaches are practically applicable, nor do they provide adequate tool support and metrics to verify the results of the migration. In our previous work [1], we also classified the different migration processes adopted by companies, highlighting the complex migration steps and the need of support from an experienced software architect, at least for identifying the architectural guidelines and helping to initiate the migration. Lu et al. [21] identified other migration strategies, distinguishing between the big bang migration, where companies replace their legacy system in one swoop, and incremental migration, where companies do the replacement step by step. All these works ([20][1][21]) agree that the migration to microservices generally increases TD, due to need to rewrite the vast majority of the code.

As for possible issues that can generate TD in microservices-based systems, in our previous works [22][23] we identified a set of microservices-specific anti-patterns and "bad smells" that can cause TD in microservices-based systems. Bogner et al. [24] extended our work, creating a public catalog of anti-patterns.

Some recent empirical studies have investigated the impact of TD in microservices-based systems. Bogner et al. [25] ran an industrial survey investigating the approaches adopted by industry to prevent the accumulation of TD, reporting that companies do have problems to prevent TD due to architectural erosion, mainly because of the lack of automated quality control at the architectural level. Moreover, in a subsequent study, Bogner et al. [26] performed another industrial survey, interviewing 17 practitioners to explore the evolvability assurance processes applied and the usage of tools, metrics, and patterns in microservices-based systems. They reported that architectural issues, and especially postponed architectural decisions, are the most harmful type of TD. Moreover, they also reported that their participants did not apply any architectural or service-oriented tools or metrics. As a result, they recommend applying static analysis tools and architectural analysis tools to keep track of the software quality and especially of the architectural issues in the systems. In their study, de Toledo et al. [7]

performed an exploratory case study on a large industry company while it was refactoring an existing microservices-based system, removing issues in the communication layer. They focused their investigation on the TD related to the communication layer of the system, indicating the large number of point-to-point connections between services as the major issue and noting the presence of business logic in the communication layer, which increased the dependency between services. Similarly to this study, we performed a case study on a single company, but we focused on the migration process itself, analyzing the system in all its aspects before and after the migration. Márquez and Astudillo [27] proposed an approach for identifying architectural TD in monolithic systems before the migration in order to satisfy the requirements of the new microservices-based system. Their proposal includes a set of tactics and patterns for better contextualizing the different types of architectural TD.

Unlike these previous works, we are investigating the impact of TD before and after the migration, considering both the TD calculated by SonarQube and the technical issues reported by the developers.

4. Case Study Design

We designed our empirical study as a case study based on the guidelines defined by Runeson and Höst [28]. In this Section, we will describe the case study design, including the goal and the research questions, the study context together with the case and subject selection, the data collection, and the data analysis procedure.

4.1. Research Questions

In this case study, we compared technical debt (TD) and its trend in a project before and after migration to microservices. The study was performed based on the following research questions (**RQs**):

RQ1: Is the TD of a monolithic system growing with the same trend as in a microservices-based system?

RQ2: Does the distribution of TD Issue types (bugs, code smells, and security vulnerabilities) change after the migration to microservices?

4.2. Case and Subject Selection

The Case Company. The company is an Italian Small and Medium-Sized Enterprise (SME) with 50 developers, developing different business suites for tax accountants, lawyers, and other related businesses. The developed systems are used by more than 10K practitioners in Italy.

The System under Migration. The company is migrating a bookkeeping document management system for Italian tax accountants. The goal of the system is to make it possible to manage the whole bookkeeping process, including management of the digital invoices, sending the invoices to the Ministry of Economic Development, and fulfilling all legal requirements. The system is currently being used by more than 2000 tax accountants, who need to store more than 20M invoices per year.

The company needs to frequently update the system, as the annual tax rules usually change every year. The Italian government normally updates the bookkeeping process between December and January, which involves not only changing the tax rate but also modifying the process of storing the invoices. However, tax declarations can be made starting in March/April of each year. Therefore, in the best case, the company has two to four months to adapt their software to the new rules in order to enable tax accountants to work with the updated regulations from March/April. Up to now, the company needed to hire a consultancy company to help them during these three months of fast-paced work.

The system is being developed in Java, deployed in the Microsoft Azure Cloud, and delivered to the customers as a web application, together with a desktop application to support document uploading and synchronization. The monolithic system was composed of 280K lines of code and had been developed for more than twelve years. The server-side of the monolithic system was developed with a Model-View-Control pattern, while the web application was developed with a set of Java servlet and jsp pages. Data was stored on an SQL server database.

The project is being developed by three teams: two teams composed of four developers each, and one team of five consultants commonly hired from December to May to help with adapting the system to the new tax rules.

The Motivation for Migration. The system has been growing every year and as a result, the code has gotten harder to understand and especially to modify. The CEO reported that before the migration, changes took a significant amount of time to implement, as they had to modify the code in several places. Moreover, every time new developers arrived in the company, they took a long time to understand the system and to be able to implement changes.

As a result of the aforementioned issues, the company decided to migrate to microservices to facilitate maintenance of the system and to distribute the work to independent groups by separating each business process, eliminating the need for synchronization between teams, and increasing velocity.

The Migration Process. The company started discussing the migration to microservices in April 2016. The software architect, together with the support of two consultants with experience in migration to microservices, analyzed the feasibility and the potential usefulness of the migration, estimating costs and evaluating the availability and need of resources.

They decided to migrate, despite an estimated high initial cost due to the migration overhead, as microservices often increase development velocity and enable teams to work independently, therefore reducing the hassle during the very short time they have for implementing the new features every year. Moreover, they decided to hire two consultants with experience i the development of microservices and experience in decomposing monolithic systems into microservices.

The company froze the development of the system between May and December 2016, implementing only critical bug fixes. During this time frame, the company sent the developers to training courses on microservices, and the software architect, with the support of the two consultants, started the analysis of the monolithic system and planned the migration.

They planned the migration to be implemented in three main steps.

Step 1 **Identification of decomposition options, architectural guidelines, and migration plan.** The software architect and the consultants sliced the system into independent microservices. First they analyzed the internal dependencies with Structure 101[9]. Then they identified decomposition options based on the different services corresponding to the business capabilities and proposed them together with a set of architectural guidelines that all the extracted microservices should follow. As an example, they decided that microservices

[9]Structure101 Software Architecture Environment - http://www.structure101.com

should not communicate directly with each other but should use the publishsubscribe pattern to communicate through the RabbitMQ message bus[10]. Moreover, they also decided to temporarily use RabbitMQ as API-Gateway. Finally, they involved all of the team members in the prioritization of the microservices, discussing which services should be developed with the highest priority. The priorities were assigned based on different criteria. In some cases, services had high priority because they had a lot of bugs, and the re-implementation as a microservice would enable the developers to completely re-develop from scratch and thus to fix the issues. In other cases, microservices were prioritized based on their business priority. The only major constraint imposed by the company was that during the period from December to April, only one service could be migrated, as the highest priority had to be on adapting the system to the new tax laws.

Step 2 **Implementation of the first microservice**. The company decided to implement some of the new regulations in the first microservice, instead of implementing them in the monolithic system and then migrating them later. The first microservice was based on a low-risk component, which would have made it possible to move all changes to the monolithic system if major issues had arisen.

Step 3 **Implementation of the other microservices**. Based on the results of the implementation of the first microservice, the other teams gradually started the implementation of other microservices, based on the plan proposed in Step 1.

[10]RabbitMQ Message Broker. Online: https://www.rabbitmq.com

Before the implementation of the first microservice, the consultants provided a skeleton of a sample microservice and supported the company in setting up a continuous delivery pipeline using Gitlab-CI[11].

In January 2017, a team ~~started the migration by extracting a set of features related to a business process into a new microservice~~ composed by four internal developers, and the two consultants started implementing the first microservice. The consultants had more than five years of experience with microservices, while the four internal members had no experience with microservices, even though they were experienced developers (two with more than 20 years of experience in Java, and two with five years of experience in Java.. However, before the migration, the company sent the developers to a training course on microservices.

We monitored the migration process until September 2018. At that point, the teams had five business processes as independent microservices, which had been extracted from the monolithic system.

4.3. Study Execution and Data Collection

The study was performed in two steps. First, we performed an automated TD analysis in order to collect data on the TD of the system before and during the migration. Then we conducted a focus group to ask for feedback on the results and on how the developers perceived the TD after the migration.

Automated TD Analysis.

The TD data was obtained by analyzing the system's commits using SonarQube[12] (version 7.0). For the analysis, we used SonarQube's standard

[11]https://about.gitlab.com/product/continuous-integration/

[12]https://www.sonarqube.org

quality profile and analyzed each commit two years before the migration and after the start of the migration.

We analyzed the TD provided by SonarQube considering the distribution of the following types of TD issues:

- **TD_M**: SonarQube "Technical Debt", also called "Maintainability Remediation Effort" (issues classified by SonarQube as "Code Smells")
- **TD_R**: Maintainability Remediation Effort (issues classified as "Bugs")
- **TD_S**: Security Remediation Effort (issues classified as "Security Vulnerabilities")

Focus Group. To confirm the results of the automated TD analysis and gain more qualitative insights on the results, we performed a focus group as the second step. The focus group was based on a face-to-face semi-structured interview. The goal was to discuss whether the migration to microservices had been beneficial and whether the participants had experienced any of the expected benefits. The focus group was moderated by one of the authors. ~~Each participant reported their answers on Post-it notes and attached them to a whiteboard. Then each participant described their answer and grouped the answer with similar ones with the help of the group discussion.~~ During the focus group, we asked four questions:

Q1: *What benefits and issues have you perceived after the migration to microservices?* Please write the issues on the red Post-it notes and the benefits on the yellow ones.
The goal of this question was to understand whether the results might be biased because of some issues in the application of the microservices pattern.

Q2: *Has the quality of the code increased after the migration to microservices? [Yes, No]*
This question aimed at understanding the perception of quality by the developers before and during the migration.
Q3: *Has the TD decreased after the migration to microservices?* [Yes/No]. Before asking this question, the moderator introduced the notion of TD. The goal of this question was to understand the perceived overall TD from the developers' point of view.
Q4: *Have you postponed any technical activities before or during the migration to microservices?* Please report the postponed activities (or group of activities) before and during the migration on the Post-it notes.

~~After the participants had agreed on the postponed activities, the moderator asked them to classify the activities based on one of the ISO 25010 quality characteristics (Functionality, Performance/Efficiency, Compatibility, Usability, Reliability, Security, Maintainability, Portability).~~

For Q1 and Q4, each participant reported their answers on Post-it notes and attached them to a whiteboard. Then each participant described their answer and grouped the answer with similar ones based on the group discussion.

For Q4, after the participants had agreed on the postponed activities, the moderator asked them to associate each activity with one of the ISO/IEC 25010 [29] quality characteristics (Functionality, Performance/Efficiency, Compatibility, Usability, Reliability, Security, Maintainability, Portability). The moderator highlighted seven areas on a large whiteboard and asked the participants to discuss to which category each activity belongs. The goal of this grouping was to complement the automated TD analysis, to answer RQ2, and to understand which type of activities were postponed before and during the migration.

4.4. Data Analysis

The whole focus group session was audio-recorded. To be able to do this, we obtained written consent from all the participants. The relevant parts were transcribed and then coded by the moderator of the focus group, and verified by one of the authors.

In order to answer RQ1, we analyzed the growth of the total TD in minutes before and after migrating to microservices. We compared the rate of growth of the TD, analyzing the overall TD (sum of TD_M, TD_R, and TD_S) and all the three TD issue types independently, applying linear regression to the data before and after migrating to microservices. The growth rates of the two regression lines were compared by inspecting the slope coefficients of the lines. The R^2 values for the regression lines were also determined. We complemented the results of RQ1 obtained by the automated TD analysis by analyzing the answers of the focus group to Q2 and Q3. The results were analyzed by comparing the positive and negative answers (true/false).

For RQ2, we determined the relative proportion of each TD issue type in the total TD of each commit. Then we determined whether the relative distribution of TD issue types was statistically different before and after migrating to microservices by applying the Mann-Whitney test. This is a non-parametric test for which the null hypothesis is that the distributions for both tested groups are identical and thus there is an even probability that a random sample from one group is larger than a random sample from the other group [30]. The results are considered statistically significant if the p-value is smaller than 0.01. Moreover, in order to determine the magnitude of the measured differences, we used Cliff's Delta, which is a non-parametric effect size test for ordinal data. The results were interpreted using guidelines provided by Grissom and Kim [31]. The effect size is small if $0.100 \leq |d| < 0.330$, medium if $0.330 \leq |d| < 0.474$, and large if $|d| > 0.474$.

We analyzed the answers from the answers of the focus group to Q4 in order to complement the results of the aforementioned analysis. Since the coding regarding the different software quality characteristics was performed by the participants, we only analyzed the results using descriptive statistics.

5. Results

In the analysis, we used the data provided by the automated TD analysis. The data we used stems from two separate time frames: before the migration (23rd August 2014 - 30th April 2016) and after the migration (3rd January 2017 - 20th September 2018). The time period between May and December 2016 is left out, as during that time the monolithic system was not being actively developed.

The focus group provided insights into the reasons for the evolution of TD. Eleven members of the company participated in the focus group: four developers from one team, five developers from another team, the software architect, and the project manager. The focus group lasted for two hours.

The participants of the focus group reported several benefits that resulted from the migration. The main benefits were decreased bug-fixing time and a large increase in the system's understandability. These were also the main initial expectations the company had had for the migration. Other minor benefits were the reduced need for synchronization between teams and the possibility to deploy new features without re-compiling and re-deploying the whole system.

However, regarding the main issues, the project manager and the software architect highlighted higher development costs, possibly because the developers used microservices for the first time. The developers highlighted the complexity of connecting the different microservices to each other or to the monolithic system. The monolithic system had the advantage of local calls, while with microservices, they had to rely on a distributed system.

5.1. RQ1: Is the TD of a monolithic system growing with the same trend as a microservices-based system?

The data from SonarQube shows different behavior of the three TD types.

The overall TD (sum of TD_M, TD_R, and TD_S) of the monolithic system before the migration was lower than the overall TD right after the migration (Figure 1). Immediately after the introduction of a new microservice, the sum of the TD of the monolithic system and all the microservices grew faster compared to the growth of the TD before the migration.

As soon as a microservice became stable, the TD decreased significantly and started growing with a lower trend compared than the growth of the monolithic system before the migration. In conclusion, once the TD was stabilized after the extraction of a feature from the monolithic system as a new microservice, the TD trend grew much slower than before the migration.

It is worth noting that TD commonly increases immediately after the creation of a new microservice and decreases after a certain period of time.

The focus group partially confirmed the result. The participants considered the need for writing the code connecting the monolithic system to

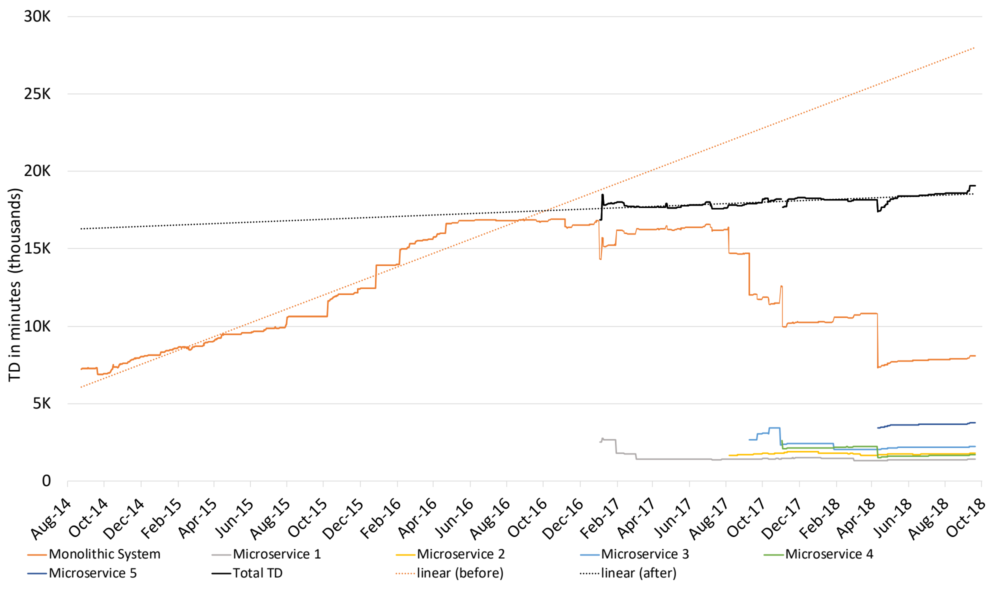

Figure 1: Overall TD evolution of the five microservices and the monolithic system (TD_M+TD_R+TD_S)

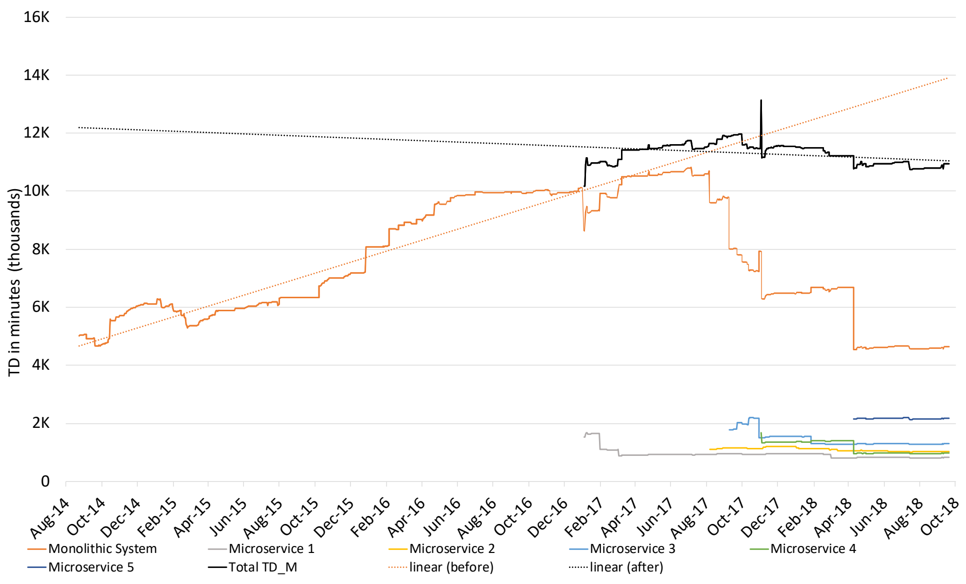

Figure 2: TD_M evolution of the five microservices and the monolithic system

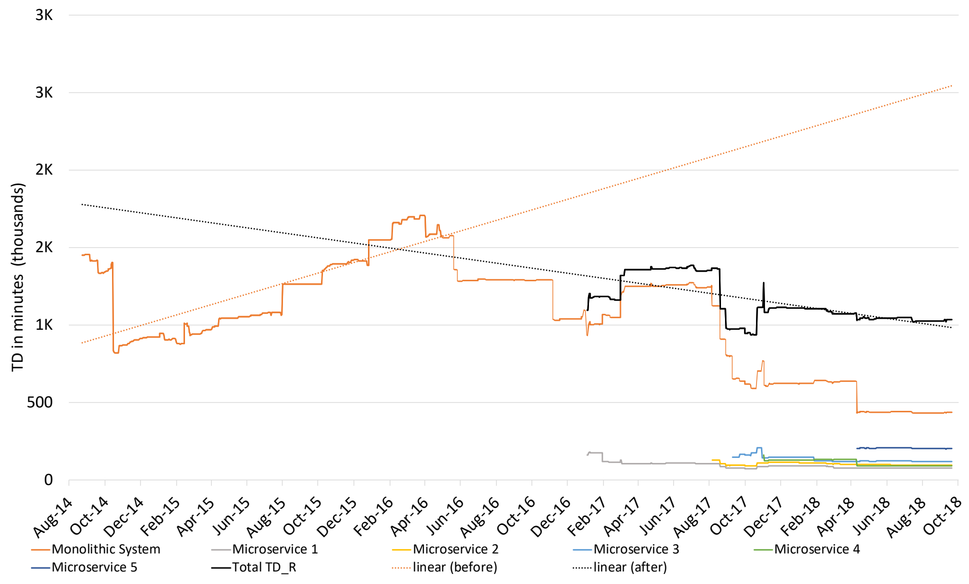

Figure 3: TD_R evolution of the five microservices and the monolithic system

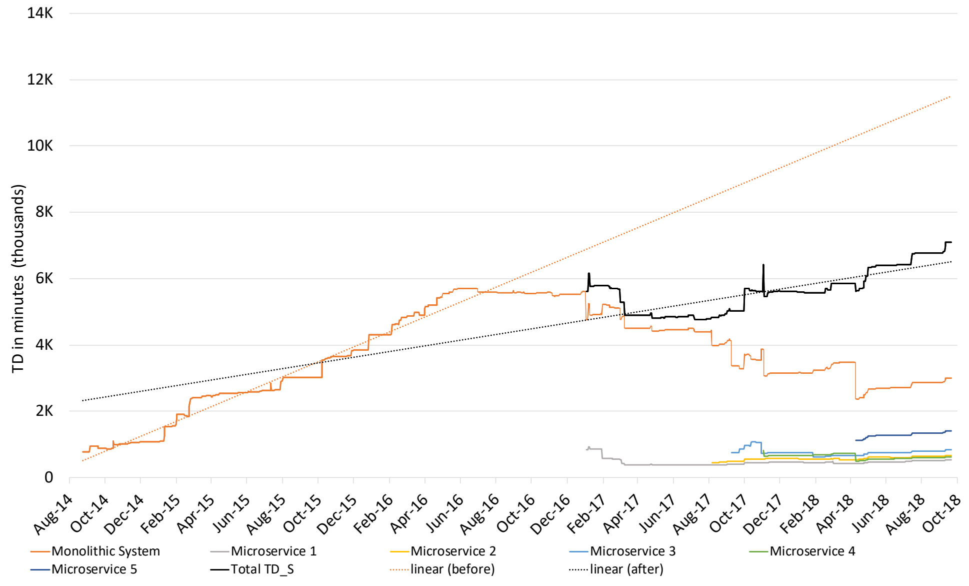

Figure 4: TD_S evolution of the five microservices and the monolithic system

the microservices as the main reason for the observed trend. The rationale behind this was that the development of a new microservice involves duplicating the existing service and adopting a large number of temporary solutions before the microservice becomes stable.

As for code quality (Focus Group - Q2), all of the participants confirmed that the code in the microservices is easier to read and modify. They also selected the created microservices as the components with the highest quality. They did so even though the migration process did not involve a complete rewrite of the code, as pointed out by the developers. Instead, they refactored only parts of the code and largely reused the existing code.

The migration also had an impact on testing the system. The developers reported that unit testing is easier in the microservices. However, integration testing is much more complicated, mainly because it is more complex to replicate the whole system locally on a developer's machine. No differences were reported for end-to-end testing. All the old Selenium tests[13] that were performed on the graphical user interface were executed without changes.

Regarding the amount of perceived TD after the migration (Focus Group - Q3), the developers, the project manager, and the software architect had different opinions. The software architect and the project manager perceived the increase of effort as a negative issue, since the overall development cost increased by more than 20% due to the cost of developing a distributed system. This result is in line with previous research work [6][1]. In addition, the software architect reported that several technical decisions were postponed during the migration process, as releasing new features requested by

[13]Selenium - Web Browser Automation. https://www.seleniumhq.org Last access: August 2019

customers was more important.

Below are several examples of postponed technical decisions. For example, in two microservices, the team kept the old SQL database, even though they had planned to migrate to a NoSQL database. Another postponed decision regarded the outdated libraries in the monolithic system. Upgrading the libraries to newer versions would have required several changes in the code, and the team decided to postpone doing this due to time constraints. The third important postponed activity was the refactoring of the code extracted for microservice 5 (See Figure 1). The code extracted from the monolithic system was planned to be deeply refactored before being used in production. However, the refactoring had to be postponed because of an urgent new feature request impacting microservice 5. The team reported that they had the choice of either implementing the new feature in the monolithic system and then applying all the changes to the almost migrated microservice, or implementing the new feature directly in the new microservice. Yet another important postponed decision was the development of an API-Gateway. The company used the RabbitMQ message bus as an API-Gateway, but they are planning to migrate to a proper API-Gateway in the future. Overall, despite all the voluntarily postponed activities, all the participants perceived the overall TD as decreased.

The developers confirmed that during the first months after the introduction of a new microservice, a lot of activities were postponed (Q1). In particular, they postponed the vast majority of quality-related aspects, except for unit and integration tests. This was done because they wanted to prioritize the delivery of the system in production. The most important type of postponed activities were those related to architectural decisions, mainly because they were not aware of common patterns and anti-patterns. As an example, the developers connected all microservices point-to-point, instead of using a message bus during the first period.

However, compared to the monolithic system, all the participants agreed that the activities postponed in the microservices will be much easier to implement in the microservices than they would have been in the monolithic system. They named the limited size of the microservices as the main reason for this.

When considering the growth of the reliability remediation effort presented in Figure 5 (TD_R) and the security remediation effort (TD_S), we can see a constant amount of TD_R after the migration, while TD_S increased much more after the introduction of each microservice.

Figures 1, 2, 3, and 4 show the linear regression lines fitted to the data before and after the migration to microservices. In particular, Figure 1 shows the overall TD evolution over time, while the other three figures visualize the individual TD related to vulnerability, reliability, and maintainability, respectively. The slope coefficients for the regression lines and their R^2 values are reported in Table 1. The table shows that the slope coefficient for the overall TD dropped significantly after migrating to microservices, and that the coefficient after migration was 89.76% smaller than when using a monolithic system. This implies a remarkable drop in the overall TD growth rate.

Looking at the distinct TD types, a vast difference can be noticed as well after the migration to microservices. In particular, TD_M and TD_R had a negative coefficient after the migration, indicating a decrease in TD after the adoption of microservices. Also, the coefficient of TD_S was reduced, resulting in a much slower TD growth rate.

Since the vast majority of business processes are still in the monolithic system after the migration, we expect the TD_M of the whole system to be lower than the TD_M of the monolithic system after all of the business processes have been migrated.

Table 1: TD slope coefficients before and after the migration to microservices.

		Type			
		TD_M	TD_R	TD_S	TD
Before migration	Slope coeff.	6.21	1.11	7.38	14.71
	R^2	0.81	0.55	0.97	0.93
After migration	Slope coeff.	-0.77	-0.54	2.81	1.51
	R^2	0.16	0.48	0.60	0.64

Table 2: The descriptive statistics for TD types before and after the migration and the results from the Mann-Whitney and Cliff's Delta tests.

		Type		
		TD_M	TD_R	TD_S
Before migration	mean	0.95	0.012	0.035
	median	0.95	0.012	0.037
	stdev	0.01	0.003	0.005
After migration	mean	0.97	0.004	0.028
	median	0.97	0.005	0.027
	stdev	0.01	0.001	0.005
Mann-Whitney	U	18,066	0	57,269
	p-value	0	0	0
Cliff's Delta	d	-0.91	1	0.70
	CI	-0.93 - -0.88	1.00 - 1.00	0.66 - 0.74

5.2. RQ2: Does the distribution of TD Issue types (bugs, code smells, and security vulnerabilities) change after migration to microservices?

When considering the distribution of the different TD types (RQ2) in Figure 5, in order to determine whether the change in the relative distributions between the different TD issue types was statistically significant, we applied the Mann-Whitney test for each type, reporting the results in Table 2. As the p-values are smaller than 0.01 for all types, we conclude that the relative distributions of TD issue types did change after migration to microservices. The results from the Cliff's Delta test are also presented in Table 2. All measured values of $|d|$ were greater than 0.474 and thus the effect size is considered large for all types. Also, the confidence intervals (CI) for d using 0.99 confidence are greater than 0.474.

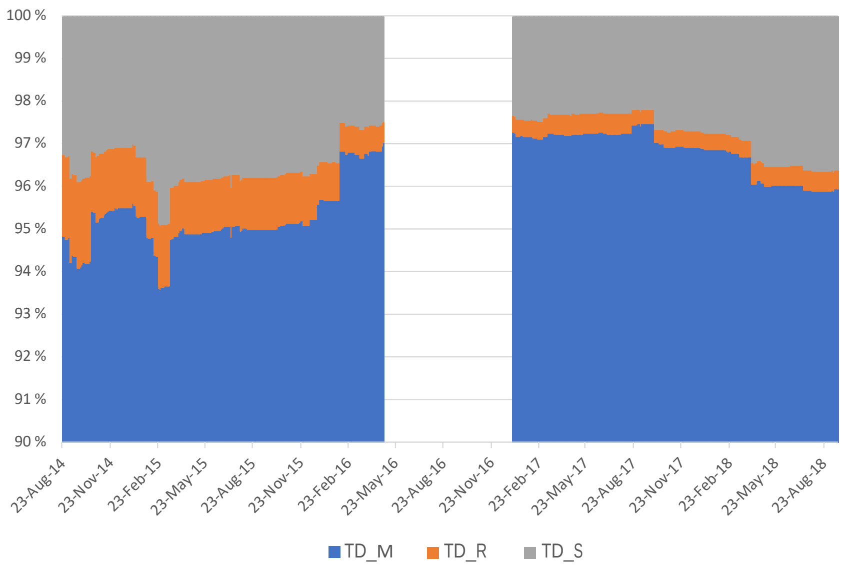

Figure 5: Relative distribution of TD issue types.

As for the focus group, the participants reported that a total of 29 activities were postponed during the development of the monolithic system, while 30 were postponed during the development of the microservices.

The participants assigned the activities to four groups based on their quality characteristics reported on the whiteboard: Performance/Efficiency, Reliability, Security, and Maintainability. Each activity was assigned to only one group. Details on the number of postponed activities are reported in Table 3.

It is worth noting that three out of four quality characteristics correspond to the TD categories proposed by SonarQube. Moreover, even though the results are only related to the number of postponed activities and not to the time needed to refactor them, we can still see an increased number of postponed activities related to security characteristics.

The developers reported that during the development of the microservices, they paid a lot of attention to the future maintainability of the system. A reduction in the number of postponed activities regarding maintainability-related aspects was therefore expected. This result is also in line with the results obtained from SonarQube, where the TD_M trend is inverted compared to the trend in the monolithic system. After the introduction of each microservice, the TD_M tended to decrease instead of increasing as before.

Considering security-related activities, the developers reported that the migration to microservices forced them to postpone more activities compared to when they developed the monolithic system. One of the main issues experienced by the developers was that there was a single sign-on approach between applications. The developers initially extended the authentication system of the monolithic system to provide authorizations to the individual services instead of implementing a system based on OAuth 2.0[14], as planned

[14]OAuth 2. https://oauth.net/2/

by the software architect. The postponement of this activity was due to the time pressure for delivering the new updates for the next tax year.

As for performance- and efficiency-related activities, the developers only postponed one activity during the development of the microservices. They reported that the performance had generally not been very important in the past, as the system never had important performance issues. The only exception was uploading of the PDF invoices, which is a task a tax accountant usually performs once a year in batch. Invoices need to be processed, sent to the Italian Ministry of Economy and Finance, and then stored permanently. The software architect and the development team planned to implement this process with a serverless function to enhance the performance and avoid over-sizing the whole infrastructure for an activity performed only once a year. However, time constraints forced the developers to postpone this activity to the next year as well.

Regarding reliability, the developers invested a lot of effort into getting a reliable system. However, they had to postpone some activities because of time pressure and technical reasons. Because of the NDA, we are not allowed to describe all the postponed activities.

Table 3: Activities postponed before and during the migration to microservices

	Before the Migration	**During the Migration**
Performance/Efficiency	3	1
Reliability	5	2
Security	10	18
Maintainability	11	9
Total	29	30

6. Discussion

The migration to microservices is a non-trivial task that requires deep re-engineering of the whole system. This heavily impacts on the whole project cost, but should also facilitate maintenance in the long run. Our case study on the migration to microservices provided both unexpected and expected results. On the one hand, we expected to see a significant decrease of the maintenance predictors, and thus a reduction of the remediation effort related to "code smells". On the other hand, we confirmed that, in the long run, the total code TD is growing slower after the migration to microservices.

The company highlighted that if the system had remained monolithic, they might have missed the deadlines of the annual updates of the system. This was caused by the constant grow of TD, as each year it took longer to adapt the system to the new tax rules. Therefore, despite the increased overall development costs, the migration was considered beneficial. The reasons for the extra costs were manifold. First of all, the developers had to deal with a new system architecture. They also had to consider various aspects such as enabling the legacy system to communicate via Enterprise Service Bus with the microservices, dealing with authentication issues as well as with process-related issues such as the introduction of the DevOps culture, including continuous building and delivery.

The extra cost was considered positive, as it included some activities that had not been performed in the past or that had been postponed. With the introduction of microservices, the company also introduced continuous delivery and increased the test coverage of the system in order to have higher confidence in the automatic builds.

The qualitative analysis also yielded three lessons learned that can help others reduce TD while migrating to microservices:

Lesson Learned 1: The creation of a microservice template is important. The initial increase of TD after the introduction of each microservice was probably due to the lack of a service template at the beginning. The second microservice was not easy to develop, but it was easier than developing the first one because of the availability of the template developed in the first microservice. We recommend investing more effort into the definition of a set of service templates, as this will dramatically ease the development of new microservices in the future.

Lesson Learned 2: The migration plan must be defined upfront, even if the company is using an agile methodology. A clearly defined migration strategy helps to avoid rework.

Lesson Learned 3: Do not postpone architectural decisions. Continuous architecture principles recommend postponing architectural decisions until they are really needed [32]. However, we observed that postponing architectural decisions causes the re-architecturing of the system to require significantly more effort than it would have required initially. An example was the usage of the lightweight message bus (RabbitMQ) as API-Gateway instead of the use of a proper API-Gateway. The SME of our case study is still using RabbitMQ as their API-Gateway. However, they are aware that implementing a proper API-Gateway at the beginning would have cost much less than migrating from RabbitMQ after two years.

7. Threats to Validity

In this Section, we will introduce the threats to validity following the structure suggested by Yin [33], reporting construct validity, internal validity, external validity, and reliability. Moreover, we will also debate the different tactics adopted to mitigate them.

Construct Validity concerns the identification of the measures adopted for the concepts studied in this work. We analyzed the TD using the model provided by SonarQube. Therefore, different tools and approaches might have provided different results. We are aware that other types of TD, such as requirements TD or architectural TD, can heavily impact the TD of a system. However, SonarQube was the only tool that we were allowed to use at the company. We are also aware that important postponed activities could have created a large amount of TD. We mitigated this threat by performing the focus group and by asking the teams to discuss other possible types of TD. A more thorough discussion on the amount of TD that these postponed activities will generate will be part of our future work.

Threats to *Internal Validity* concern factors that could have influenced the obtained results. The postponed activities were collected using a group discussion. It is possible that some developers did not want to publicly expose some activities they had postponed.

Threats to *External Validity* concern the generalization of the obtained results. The results of this paper are based on the monitoring of the development process of a single company. The results could be slightly different in other companies. However, based on previous studies on microservices, the developers confirmed that microservices increase maintainability, code readability, and system understandability [1]. Therefore, we expect that other systems could also benefit from a decrease of the TD when migrating to microservices.

Threats to *Reliability* refer to the correctness of the conclusions reached in the study. This study was a preliminary study, and therefore we applied simple statistical techniques to compare the trends of the TD before and after the migration. The results of the statistical technique applied are also confirmed by Figure 1. We are aware that more accurate statistical techniques for time series could have provided a more accurate estimate of the difference of the slopes. However, we do not expect that different statistical techniques would provide a contradictory result.

8. Conclusion

In this work, we compared the technical debt (TD) before and after the migration to microservices of a twelve-year-old software project (280K Lines of Code) developed by an Italian SME.

We conducted a case study, analyzing the code TD of a system under development over a period of four years (two years before and two years after the migration). Then we conducted a focus group to analyze the postponed activities in depth and to get more insights into the results. The first result revealed that TD grew 90% slower after the development of microservices. After the initial introduction of each microservice, TD grew for a limited period of time, mainly because of the new development activities. When the code of the microservice stabilized, TD decreased and started growing linearly, with a growing trend much lower than for the monolithic system.

Unexpectedly, when comparing the distributions of TD issue types before and after the introduction of microservices, important and statistically significant differences emerged. The proportion of SonarQube issues classified as bugs and security vulnerabilities decreased, while code smells (maintainability issues) increased. Since microservices are supposed to facilitate software maintenance, we expected a reduction of code smells.

The developers confirmed the overall results, perceiving reduced maintenance complexityand increased velocity. The overall development effort increased after the introduction of microservices because of the extra effort due to the re-development of the system, the connection of the legacy system to the new microservices, introduction of a distributed authentication mechanism, and many other activities not previously considered. However, the manager confirmed that the increased velocity and team freedom compensated for the required extra effort. For the company, it was especially important that the migration allowed them to remain on the market, releasing the annual tax rules update required by the government on time.

Future work will include an investigation of the impact of other types of TD during the migration to microservices. We aim to analyze the same projects using tools for detecting architectural smells. Moreover, we aim to investigate TD due to temporary architectural decisions. Our next goal is to understand how long different activities could be postponed before the benefit of postponing an activity is canceled out by the increased effort needed to refactor it. As an example, if an activity has an interest equal to zero (i.e., if the development/refactoring effort does not increase if postponed), it can be postponed until it is needed, whereas if an activity has a monthly interest of 10% (i.e., 10% extra interest per month), it should be refactored as soon as possible before becoming too expensive.

References

References

[1] D. Taibi, V. Lenarduzzi, C. Pahl, Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation, IEEE Cloud Computing 4 (2017) 22–32.

[2] D. Taibi, V. Lenarduzzi, C. Pahl, Architectural patterns for microservices: A systematic mapping study, in: 8th International Conference on Cloud Computing and Services Science (CLOSER), p. 8.

[3] H. Knoche, W. Hasselbring, Using microservices for legacy software modernization, IEEE Software 35 (2018) 44–49.

[4] W. Cunningham, The wycash portfolio management system, in: OOPSLA '92.

[5] Z. Li, P. Avgeriou, P. Liang, A systematic mapping study on technical debt and its management, Journal of Systems and Software 101 (2015) 193 – 220.

[6] J. Soldani, D. A. Tamburri, W.-J. V. D. Heuvel, The pains and gains of microservices: A systematic grey literature review, Journal of Systems and Software 146 (2018) 215 – 232.

[7] S. de Toledo, A. Martini, A. Przybyszewska, D. I. Sjøberg, Architectural technical debt in microservices: A case study in a large company, in: Proceedings of the 2019 International Conference on Technical Debt, TechDebt '19.

[8] S. Newman, Building Microservices, O'Reilly Media, Inc., 1st edition, 2015.

[9] S. McConnell, Managing technical debt, ICSE Keynote (2013).

[10] P. Avgeriou, P. Kruchten, R. L. Nord, I. Ozkaya, C. Seaman, Reducing friction in software development, IEEE Softw. 33 (2016) 66–73.

[11] A. Nugroho, J. Visser, T. Kuipers, An empirical model of technical debt and interest, in: Workshop on Managing Technical Debt, MTD '11, pp. 1–8.

[12] C. Seaman, Y. Guo, Chapter 2 - measuring and monitoring technical debt, volume 82 of *Advances in Computers*, Elsevier, 2011, pp. 25 – 46.

[13] N. Zazworka, R. O. Spínola, A. Vetro', F. Shull, C. Seaman, A case study on effectively identifying technical debt, in: Int. Conf. on Evaluation and Assessment in Software Engineering, EASE '13, pp. 42–47.

[14] N. Zazworka, A. Vetro', C. Izurieta, S. Wong, Y. Cai, C. Seaman, F. Shull, Comparing four approaches for technical debt identification, Software Quality Journal 22 (2014) 403–426.

[15] Y. Guo, R. Spínola, C. Seaman, Exploring the costs of technical debt management – a case study, Empirical Software Engineering 21 (2016) 159–182.

[16] M. Fowler, K. Beck, Refactoring: Improving the design of existing code, Addison-Wesley Longman Publishing Co., Inc. (1999).

[17] K. Mordal-Manet, F. Balmas, S. Denier, S. Ducasse, H. Wertz, J. Laval, F. Bellingard, P. Vaillergues, The squale modela practice-based indus-

trial quality model, in: 2009 IEEE International Conference on Software Maintenance, IEEE, pp. 531–534.

[18] J.-L. Letouzey, M. Ilkiewicz, Managing technical debt with the sqale method, IEEE software 29 (2012) 44–51.

[19] B. Curtis, J. Sappidi, A. Szynkarski, Estimating the principal of an application's technical debt, IEEE software 29 (2012) 34–42.

[20] J. Fritzsch, J. Bogner, A. Zimmermann, S. Wagner, From monolith to microservices: A classification of refactoring approaches, in: J.-M. Bruel, M. Mazzara, B. Meyer (Eds.), Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment, Springer International Publishing, Cham, 2019, pp. 128–141.

[21] N. Lu, G. Glatz, D. Peuser, Moving mountains practical approaches for moving monolithic applications to microservices, in: International Conference on Microservices (Microservices 2019).

[22] D. Taibi, V. Lenarduzzi, On the definition of microservice bad smells, IEEE Software 35 (2018) 56–62.

[23] D. Taibi, V. Lenarduzzi, C. Pahl, Microservices architectural, code and organizational anti-patterns, Springer Book on Microservices (2019).

[24] J. Bogner, T. Boceck, M. Popp, D. Tschechlov, S. Wagner, A. Zimmermann, Towards a collaborative repository for the documentation of service-based antipatterns and bad smells, in: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), pp. 95–101.

[25] J. Bogner, J. Fritzsch, S. Wagner, A. Zimmermann, Limiting technical debt with maintainability assurance: An industry survey on used techniques and differences with service- and microservice-based systems, in: Proceedings of the 2018 International Conference on Technical Debt, TechDebt '18, ACM, New York, NY, USA, 2018, pp. 125–133.

[26] J. Bogner, J. Fritzsch, S. Wagner, A. Zimmermann, Assuring the evolvability of microservices: Insights into industry practices and challenges, in: 35th International Conference on Software Maintenance and Evolution (ICSME).

[27] G. Márquez, H. Astudillo, Helping novice architects to manage architectural technical debt in microservices architecture, in: XIII Jornadas Iberoamericanas de Ingeniera de Software e Ingeniera del Conocimiento.

[28] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Softw. Engg. 14 (2009) 131–164.

[29] ISO/IEC, ISO/IEC 25010 System and software quality models, Technical Report, 2010.

[30] P. E. McKnight, J. Najab, Mann-whitney u test, The Corsini encyclopedia of psychology (2010) 1–1.

[31] R. J. Grissom, J. J. Kim, Effect sizes for research: A broad practical approach., Lawrence Erlbaum Associates Publishers, 2005.

[32] M. Erder, P. Pureur, Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World, Morgan Kaufmann, Amsterdam, 2016.

[33] R. Yin, Case Study Research: Design and Methods, 4th Edition (Applied Social Research Methods, Vol. 5), SAGE Publications, Inc, 4th edition, 2009.