Università degli Studi di Firenze

Scuola di Ingegneria - Dipartimento di Ingegneria dell'Informazione

Tesi di Laurea Magistrale in Ingegneria Informatica

Analysis of the Evolution of Code Technical Debt in Microservices Architectures

Candidato
Kevin Maggi

Relatori
Dott.Ric. Roberto Verdecchia
Prof. Enrico Vicario

Correlatore
Dott.Ric. Leonardo Scommegna

Anno Accademico 2022/2023

to my family

and to Elisa

Acknowledgment

First of all, I would like to express my gratitude to my supervisors, Doctor Roberto Verdecchia and Professor Enrico Vicario, for giving me the opportunity of doing this thesis project, for offering me all the support necessary to work at my best, and for having transmitted to me even just an infinitesimal of their knowledge and their skills; I also deserve thanks to my co-supervisor, Doctor Leonardo Scommegna, for the help given to me in understanding the theory on which this work is based and in overcoming the difficulties I encountered.

Many thanks also to Angelo and Leonardo who, despite the remote methods that have characterized part of our journey, have always been ready to offer me their help during the difficulties.

In the end, special thanks to all my family, my girlfriend Elisa and her family and my friend Matteo for the fundamental moral support that they have given to me over these years.

Abstract

Context For a long time, and sometimes also nowadays, in literature it was talked about microservices architectures as a growing emerging architectural approach; but now this architecture, although still growing, is widespread both in practice and in academic and industrial interest. Despite that, to date, no one has conducted a study to investigate how Technical Debt evolves in software-intensive systems based on such architecture.

Goals This study aims to understand how Technical Debt evolves in microservices architectures both in terms of overall evolution trend and related to number of microservices that compose the system.

Method I select a sample of 13 open-source projects, through a multi-step mixed manual-automated filtering of the results of some queries on GitHub, with an interesting evolution in terms of number of microservices. Then I adopt a research method based on repository mining and automated source code analysis complemented with manual code inspection. The raw data is in the end statistically analyzed to get the results.

Results From the analysis it is possible to observe that periods of development without significant Technical Debt variations are present, but generally the Technical Debt tends to grow in time; moreover the variations can happen regardless of the development activity performed in a commit, although some are naturally more inclined to introduce Technical Debt. Technical Debt and number of microservices seem to be correlated in the

majority of cases, although not always. Furthermore adding (or removing) a microservice has a similar impact on Technical Debt regardless of the number of microservices already present.

Conclusions Adherence to microservices architecture principles might help code quality management by keeping Technical Debt compartmentalized within microservices and hence more manageable. Anyway developers should pay attention to Technical Debt they may introduce, because it can grow regardless the development activities they conduct. Keeping Technical Debt not growing during the evolution of a microservices architecture could be possible, but its growth while the system becomes bigger and more complex might be inevitable.

Sommario

Contesto A lungo, e talvolta ancora adesso, in letteratura si è parlato di architetture a microservizi come un approccio architetturale emergente in crescita; ma oggigiorno questa architettura, sebbene ancora in crescita, è diffusa sia nella pratica che nell'interesse accademico e industriale. Nonostante ciò, a oggi, nessuno ha condotto uno studio che investigasse come il Technical Debt evolve nei software-intensive system basati su tale architettura.

Obiettivi Questo studio mira a capire come il Technical Debt evolve nelle architetture a microservizi sia in termini di tendenza complessiva, sia in relazione al numero di microservizi che compongono il sistema.

Metodo Ho selezionato un campione di 13 progetti open-source, tramite un filtraggio multi-step misto manuale-automatizzato dei risultati di alcune query di GitHub, con un'evoluzione nel numero dei microservizi interessante. Dopodiché ho adottato un metodo di ricerca basato su repostory mining e analisi automatizzata del codice sorgente completato con un'ispezione manuale del codice. I dati grezzi sono stati infine analizzati statisticamente per ottenere i risultati.

Risultati Dall'analisi è possibile osservare che sono presenti lunghi periodi di sviluppo senza variazioni del Technical Debt, ma generalmente questo tende a crescere nel tempo; inoltre le variazioni possono avvenire a prescindere dall'attività di sviluppo eseguita nel commit, sebbene alcune siano naturalmente più inclini a introdurre Technical Debt. Il Technical

Debt e il numero dei microservizi sembrano essere correlati nella maggior parte dei casi, sebbene non sempre. Infine aggiungere (o rimuovere) un microservizio ha un impatto simile sul Technical Debt a prescindere dal numero dei microservizi già presenti.

Conclusioni Aderire ai principi dell'architettura a microservizi può aiutare a gestire la qualità del codice tenendo il Technical Debt compartimentalizzato tra microservizi e quindi più gestibile. Tuttavia gli sviluppatori devono porre attenzione al Technical Debt che potrebbero introdurre, perché questo può crescere indipendentemente dalle attività di sviluppo condotte. Mantenere il Technical Debt non crescente durante l'evoluzione di un'architettura a microservizi potrebbe essere possibile, ma una sua crescita quando il sistema diventa più grande e complesso potrebbe essere inevitabile.

Contents

Chapter 1

Introduction and Motivation

In the last years microservices architectures (MSAs) have become more and more popular, thanks to their many benefits, including very high scalability and flexibility, and microservices independence along the whole software lifecycle: development, testing, deployment and maintenance. Certainly maintenance phase is one of those with major benefits: indeed it strongly depends on the software quality of the system and studies [8] show that the perception of resulting software quality in MSAs is really positive; this makes the maintenance phase easier to deal with.

Despite many benefits, obviously, MSAs present also challenges, like the consistency management and the additional effort required for integration and system testing. Also the potential loss of the bigger architectural picture has to be mentioned among the difficulties introduced by microservices approach: developers adopt sub-optimal implementation expedients that, while providing temporary benefits, tend to make future developing intervention harder. It is exactly these temporary expedients that increase the Technical Debt (TD) and, as consequence, decrease the software quality (see Infobox 1.1). Indeed TD is one of the factors in software development that can lead, if left growing, to lower development speed and higher number of bugs.

At this point it is natural to ask if, from a software quality perspective, MSA has more pros or cons with respect to other architectures, in particular the monolithic one. TD (and in particular Code TD on which this study focuses) has been treated by literature extensively from years ago [4,5], while

microservices are a relatively new phenomenon so the academic interest is still growing [6, 7]. In light of this it should be surprising that, although both Code TD and MSA are popular topics in literature, only few studies, to date, have focused the relation between the two subjects; and also more amazing that, at the best of my knowledge, no one has ever investigated quantitatively in depth how Code TD evolves in MSAs.

More on... Technical Debt Infobox 1.1

According to [2], the software engineering community converges on defining Technical Debt 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; impacting internal system qualities, primarily maintainability and evolvability.*

In literature various types of TD have been introduced [3]: Architectural TD, Code TD, Design TD and much more. Code TD is one of the more widespread types considered and it refers to aspects found in the source code that affect negatively its legibility making it more difficult to be maintained; in other words it regards issues related to bad coding practices.

In a recent work [1] co-authored with my supervisors and co-supervisor I have preliminarily considered a case study that shows us its Code TD increasing over time, despite many long more or less constant periods; and moreover it was possible to see that it increases/decreases same way regardless the number of microservices, so linearly with respect to it.

This study aims to extend the investigation to other systems to confirm or refute the aforementioned preliminary results. Understanding how Code TD evolves in microservice-based systems is necessary to know how it can be

effectively managed in these architectures, to better support the long-term success of software intensive systems based on such architectural style.

Thesis structure. This work is structured as follows: in Chapter 2 I illustrate some studies similar or somewhat related to this one, then in Chapter 3 I move to this study quickly introducing the experiment design based on Mining Software Repository (MSR), before to go deep in the microservices detection mechanism designed on purpose in Chapter 4; in Chapters 5 and 6 the experiment execution and results are described; Chapter 7 is about which threats to validity can have influenced the results and if they can have effectively affected them significantly; at the end in Chapters 8 and 9 I explain how this study's results impact the practice and I sum up the study and hint some possible future works.

Chapter 2

Related works

Assunção *et al.* [9] conducted a large empirical study on 11 open-source software (OSS) projects that aimed to understand how microservices evolve in time; in particular they considered the evolution from 3 different points of view: technological (technologies that enable the development of microservices), services (business logic of the system) and miscellaneous (aspects highly specific of the system). As a result they found that rarely the evolution is service-based, being the vast majority of commits related to technical changes. Since both technical and services evolutions can have impact to different extent on the TD, this study is complementary to that one, going to define a more complete overview.

By considering the literature that instead focuses on TD in MSAs I can identify, to the best of my knowledge, only few studies.

The works of Lenarduzzi *et al.* [10, 11] might potentially be the most similar to this one. They investigated the effect of migration from monolithic to microservices architecture on TD on two study cases obtaining opposite results: while in one of them the TD after the migration grows faster, in the other the increase is slower. Anyway both the studies agree on the fact that TD in each microservice, although an initial very fast growth, tends to stabilize and to grow slower than the overall system. This study anyway differs from these two for at least two reasons: (i) I do not focus on migration from monolith to microservices and (ii) I consider a large number of study cases (some of them also bigger than the small two, respectively 4 and 5 microservices, considered by Lenarduzzi *et al.*).

By inspecting the other related literature, TD in microservices seems to be investigated primarily from a qualitative point of view.

Toledo *et al.* [12] investigated in a multiple case study the Architectural TD (ATD) in microservices from a qualitative point of view. The results of 25 interviews identify ATD issues, their negative impact and common solutions to repay them. In a similar work by Toledo *et al.* [13] ATD in the communication layer was considered, through a qualitative analysis of documents and interviews related to a big case study. These two studies differ from this one both for the subject, ATD instead of Code TD, and for the research method, qualitative instead of quantitative.

Bogner *et al.* [14] studied how the sustainable evolution of 14 microservice-based systems was ensured, conducting 17 semi-structured interviews. The main focus was not on TD (differently from this work), but it emerges to be one of the main issues that undermine sustainable evolution. Moreover, albeit some tool-based DevOps processes were mentioned, the study was based on a qualitative research method. Bogner *et al.* in a different work [15] asked to 60 software professionals how TD can be limited through maintainability assurance. Results indicate that using systematic techniques benefits software maintainability, but often in practice these tools are not used. Also this study, again, adopts a qualitative approach and it does not account the TD evolution.

A more systematic scrutiny of the literature on TD in microservices was conducted by Villa *et al.* [16]. Basing on the analysis of the 12 primary studies they selected, the intuition on which this research is based, namely the absence of studies focusing the evolution of TD in microservice-based systems, is somehow confirmed. Moreover, from their results, Architectural TD, Code TD and Design TD appear to be the most common types of TD reported in microservices contexts. Such result reflects the general trend observed in developer discussion as highlighted by Kozanidis *et al.* [17], that

obtains the same result from the analysis of TD related questions on Stack Overflow. This result provides further support to the focus of this work, that is the evolution of Code TD in MSAs.

Chapter 3

Study Design

In this Chapter I document the research design of the study, in terms of Research Goal (Section 3.1), Research Questions (Section 3.2) and in the end Research Process (Section 3.3). Experiment Execution, instead, will be treated in chapter 5.

3.1 Research Goal

The goal of this study is to conduct an investigation into the evolution of Code TD in software-intensive systems based on MSA. By considering the Goal-Question-Metric approach of Basili *et al.* [18], the goal of this study can be formulated as follows:

> ***Analyze*** *software evolution*
> ***For the purpose of*** *studying trends and characteristics*
> ***With respect to*** *Code Technical Debt*
> ***From the viewpoint of*** *software engineering researchers*
> ***In the context of*** *microservice-based software-intensive systems.*

The choice of focusing on Code TD rather than other types, like ATD, is guided by multiple factors: (i) Code TD is one of the most frequent types of TD appearing in microservice-based systems [16], (ii) differently from other types of TD, Code TD is supported by a vast range of consolidated tools, largely used both in academic research and industrial practice [19] and (iii) for the previous reason, it allows this study to consider an heterogeneous set of experimental objects.

3.2 Research Questions

Basing on the just formulated goal of the work it is possible to derive the main Research Question (RQ) on which this study is based, that can be formulated as follows:

> **RQ**: *How does Code Technical Debt evolve in a microservice-based software-intensive system?*

This RQ covers the overall goal of the study, expressing the intent to study the evolution of Code TD in MSA. In order to be more systematic, it is possible to decompose it into sub-RQs, each one considering one of the different facets of TD evolution this research aims to investigate. More specifically it has been decomposed into the two following RQs, each associated to one or more hypothesis testing.

> **RQ$_1$**: *What is the evolution trend of Code Technical Debt in a microservice-based software-intensive system?*
>
> $\mathbf{H_0^{1.1}}$: *TD evolution does not change in time*
>
> $\mathbf{H_a^{1.1}}$: *TD evolution changes in time*
>
> $\mathbf{H_0^{1.2}}$: *TD evolution does not present periodic trend*
>
> $\mathbf{H_a^{1.2}}$: *TD evolution presents periodic trend*

With RQ$_1$ I aim to understand the overall evolution trend of TD in microservice-based systems, e.g. if it is constant through the evolution of software system, if it shows a growing trend or if a periodic trend can be noted. It comes naturally to conjecture an increasing trend (as resulted in [10, 11], although in different measures) with appreciable periodic trend (i.e. seasonal periods where developers are more/less prone to incur in TD, like before/after holidays).

RQ$_2$: *Is there a relation between Code Technical Debt evolution and number of microservices?*

H$_0^2$: *TD evolution does not depend on number of microservices*

H$_a^2$: *TD evolution depends on number of microservices*

With RQ$_2$ I aim to understand if there is a relation between the evolution of TD and the number of microservices that compose the system. About this I could conjecture that, due to sub-optimal implementation choice, as the number of microservices grows, TD grows at an higher rate (i.e. TD is in superlinear or even exponential relation with the number of microservices).

3.3 Research Process

In this section I go deeper into the description of the research process followed to answer to the RQs. I describe which dataset has been selected, how it has been analyzed and how, with the results, it is possible to answer to the RQs.

3.3.1 Dataset Definition

As dataset to be analyzed I want a substantial number of repositories of microservice-based software-intensive systems characterized by a real industrial-like development process, so real systems or industrial demos (no toy examples or exercise-in-style demos).

During a preliminary discovery phase a lack of this type of OSS projects turned out, clearly because most companies do not make their products open-source; this is also the reason why I had to accept also industrial demos, surely more widespread.

In literature many authors use the dataset provided by Rahman *et al.* [20] and extended by Taibi [48]. Anyway most of the projects collected do not meet the aforementioned requirements.

In the end it has been decided to define a custom dataset by querying GitHub[1] and filtering the results to select the "most interesting" from the point of view of the evolution, like Figure 3.1 shows.

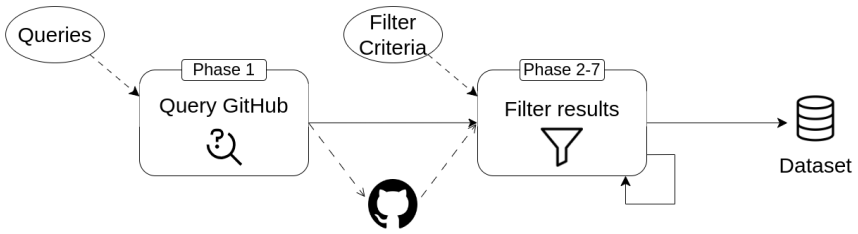

Figure 3.1: Dataset creation process overview

Queries

In 2022, according to the "The State of Developer Ecosystem" annual survey conducted by JetBrains [49], the most frequent programming languages in microservices development come out to be: Java, that is by far the most used, Python, Go, C#, TypeScript and JavaScript. Therefore it has been decided to focus the research on these 6 languages.

The repositories retrieved[2] by queries, pre-filtered by language, have been selected by:

- topic: those with "microservice(s)-architecture" or "microservice(s)" as topic (with at least 10 stars to avoid the most insignificant ones) or
- keyword: those containing the "microservice" keyword (with at least 100 stars to avoid the most insignificant ones).

The resulting composite query can be formulated as follows:

[1] www.github.com
[2] At the date of 11/09/2023

$$language \in \{Java,\ Python,\ C\#,\ Go,\ TypeScript,\ JavaScript\} \land \Big(\big(topic \in \{\text{microservices},\ \text{microservice},\ \text{microservices-architecture},\ \text{microservice-architecture}\} \land stars \geq 10 \big) \lor \big(keyword \in \{\text{microservice}\} \land stars \geq 100 \big) \Big)$$

Filter Criteria

The results need to be filtered[3] to obtain repositories that meet the typology requirements and to discard the ones surely not interesting as case study. Before to discern manually real industrial cases and industrial demo cases, reducing the number of results with some automated filter is essential; this is done by the first 4 filter steps with the following Filter Criteria:

FC_1: *the repo must have a docker-compose file (with a standard name).*
It is essential to be able to detect and count microservices, as I will illustrate in Chapter 4.
The *docker-compose* file should be present in all commits, anyway at this step, in order not to check commit-by-commit all the repositories resulting from querying, I make a first rough selection looking at only the last one;

FC_2: *the repo must have at least 250 commits.*
This criterion simply discards repositories not representative for long-lived software applications;

FC_3: *the repo must have at least 2 contributors.*
This criterion discards repositories not characterized by an industrial-like development (trivially practitioners' exercise-in-style).
It has been decided to count bots as real contributors, since some of them are able to modify code, potentially introducing TD;

FC_4: *the repo must have a docker-compose file for at least ⅔ of the commits and in any case at least 250 commits.*

[3] at the date of 11/09/2023, including commits until 08/09/2023 (included)

This aspect has already been discussed with FC_1 and FC_2, but now the remaining results should be few enough to do a commit-wise check with a reasonable effort.

At this point the results should be few enough to make a manual check on the nature of repositories to discard all the false positives in terms of architecture (systems not MSAs) and typology (toy examples, complex exercise-in-style demos, etc.):

FC_5: *the repo must present a MSA and must be a real industrial system or an industrial demo.*

The check has to be done manually by examining[4] the description, the README file and, in absence of clear clues, the artifacts like *docker-compose*, structure of the code, etc.

Complete industrial MSA, starter kits, templates and big-industry related projects are also considered acceptable, while single microservices, MSA components, examples from books/sites, libraries, development frameworks and platforms are not.

Last filtering step is the most fine-grained because it considers the interestingness of the evolution in terms of number of microservices.

FC_6: *the repo must have an interesting evolution in the number of microservices.*

This criterion has been formally formulated as follows:

$FC_{6.1}$: *the repo must have at least 5 microservices at some point in its history.*

This criterion is essential to discard repositories not representative for complex industrial MSA systems;

[4]All the repositories have been inspected in the state they were as of 15/09/2023

$FC_{6.2}$: *the repo must not have a flat evolution period longer than ½ of its history and in any case more than 750 commits.*
This criterion serves to discard repositories that do not evolve at all or at least that have already reached their maturity for too long;

$FC_{6.3}$: *the repo must not have no microservices for more than ³⁄₁₀ of its history.*
Also this criterion serves to discard repositories with poor evolution or that have been migrated to microservices from another architecture (i.e. monolithic) during their development;

$FC_{6.4}$: *the repo must not return to no microservices more than one time every 175 commits.*
This criterion seems quite strange, but it is essential to avoid repositories with an exceptionally stormy development: indeed few repositories, above all in their initial phase, tend to be refactored so frequently (with some commits with no detected microservices between the starting form and the final one) that it is impossible to appreciate the evolution.

All the parameters' values set in this last group of Filter Criteria have been found empirically, tuning them in order to find the best filtering level that allows to obtain an appropriate number of resultant repositories (approximately between 10 and 20).

3.3.2 Dataset Analysis

Regarding the analysis of the dataset, it is described in Figure 3.2 and it consists, for each repository, of an iteration on all its commits and, for each commit, in counting microservices and measuring the TD.

All the analysis data is collected before to proceed to their analysis for answering to the RQs.

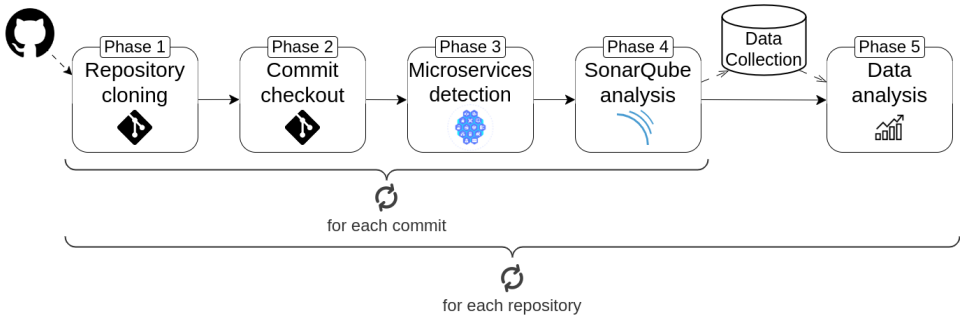

Figure 3.2: Dataset analysis process overview

Phase 1: Repository cloning

Of course the first step simply consists in cloning the repository from GitHub.

Phase 2: Commit checkout

The second step is to checkout the commit to be analyzed. The commits taken into account are those of the main branch (and of secondary branches already merged into the main, as highlighted by Kovalenko *et al.* [21]) linearized in temporal order. During this phase some additional version control related data is collected to enable future deeper investigation of the dataset with no need of executing again the collection phase. Among these data the main ones are: committer, author, date, commit hash and above all parent(s) hash (so the graph topology information missed due to linearization are not completely lost).

Phase 3: Microservices detection

Before to analyze the TD, the microservices composing the system are detected and counted. This process is based on an analysis of the Docker

configuration files, *docker-compose* and *Dockerfile*. In Chapter 4 this new method, designed on purpose for this study, is presented in details.

Phase 4: SonarQube™ analysis

Last data to collect are those related to the code quality analysis by Sonar™ tool suite[5], one of the most known tool for the purpose [19]. SonarScanner examines files and SonarQube analyzes raw data to compute metrics. The main metric of interest is obviously the SQALE index, that measures the TD in minutes needed to pay it [22], but also other metrics are collected in order to open to future further investigations on the same dataset (project size in terms of files, lines, lines of code and cognitive complexity; issues, bugs, code smells and vulnerabilities).

The measurements consider the whole repositories, analyzing all the languages supported by SonarQube, and not only the main one. For some languages, namely Java and C#, SonarScanner requires compiled code. For this, the projects containing these languages needed special attention: in the Java case, at first the projects are built using the build automation tool used by the author (Maven™[6] or Gradle™[7] or their wrappers if available), and then they are analyzed by SonarScanner for Maven and SonarScanner for Gradle; in the C# case, the projects are built and analyzed using the standard procedure for SonarScanner for .NET. In these two cases, based on how the specific SonarScanner versions work, the analyzed code (of any of the languages) is just the one included by the building configuration files.

Data Analysis

As final step of the research process, the data collected through dataset analysis is analyzed to answer the RQs.

[5] www.sonarsource.com

[6] www.maven.apache.org

[7] www.gradle.org

To answer RQ_1 various tests and investigations are conducted on TD evolution, trend and seasonality:

- first of all an overall trend presence is tested through the Mann-Kendall test [23], a non-parametric test based on Kendall's rank correlation coefficient (Kendall's τ) that looks for the increasing or decreasing trend of a time series, whose implementation is available as open-source R library[8];
- being that commits are irregularly-spaced, while next steps require a regularly-spaced time series, this gap is filled transforming the series by (i) keeping only the last commit of each day and (ii) introducing fictitious findings on missing days with linear interpolation;
- then the trend is obtained, for graphical means analysis, by applying LOESS regression [24], that basically smooths the TD evolution;
- to gain further insights into the "hotspots" of TD time series (i.e. commits characterized by the most outlier values in TD measurement), the corresponding commits are manually scrutinized: commits are ordered by absolute value of introduced (or paid back) TD and top 10 commits' content is analyzed;
- in the end the seasonality is analyzed. Since both the following steps require that at least 2 periods (in this case years) of data are collected, systems with a shorter life are not considered for this analysis. The investigation comprehends:
  - a seasonality test, using the combined method presented by Ollech and Webel [25] (basically a QS test and a Kruskall-Wallis test) and made available as open-source R library[9];

[8] www.rdocumentation.org/packages/Kendall/versions/2.2/topics/MannKendall
[9] www.rdocumentation.org/packages/seastests/versions/0.15.4/topics/combined_test

  - if a seasonality is found, a decomposition of TD evolution into its trend, seasonal and irregular components. For this I adopt the STL algorithm [26] since it does not assume a time series distribution and it was successfully used in previous software engineering studies [27, 28]; moreover an open-source implementation is available as an R library[10]. The resulting trend and seasonal components are then inspected qualitatively via graphical means. The interpolation tends to slightly smooth the seasonality trend, so further considerations about this choice are discussed in Chapter 7.

Before to move on, data is cleaned by removing those isolated commits that, due to a bad formatted *docker-compose*, count incorrectly no microservices; in this way following analysis are not influenced by them.

Indeed, to answer RQ_2, potential correlation and causation between number of microservices and TD evolution are examined:

- the first aspect to clarify is if a correlation exists between the two time series and this is done with Cross-Correlation Function (CCF) [29–31], that tests correlation between the two time series at different lags (i.e. offset by some commits forwards and backwards);
- if correlation is present at negative lags (i.e. TD time series correlates with delayed microservices time series), then, to determine if microservices number changes cause an increment/decrement of TD, the Granger Causality test [32] is conducted. The optimal lag order is calculated by leveraging the Akaike Information Criterion [33];
- in the end, potential correlation between the derivative of TD time series and the microservices number is analyzed to understand if the

[10] www.rdocumentation.org/packages/stats/versions/3.6.2/topics/stl

growth speed of TD depends on the number of microservices. Also in this case CCF is exploited.

Since Granger Causality test requires stationary data and CCF can give spurious results with non-stationary series [34], before all the steps, this assumption is tested with the Augmented Dickey-Fuller test [35] and in case of non-stationarity, I make data stationary by differencing them.

The choice of studying correlation (and causation consequently) at different lags comes due to the fact that a microservice is detected at its insertion as Docker container, but this moment not necessarily coincides with the start of its development. This could have introduced a delay between the TD time series and the number of microservices one (or even the opposite, i.e. the insertion as Docker container is done after the development of the microservice).

Chapter 4

Microservice detection

What I have described so far implies the ability of getting the number of microservices present in a MSA in an automatic mode and starting from the artifact in the repository, i.e. the configuration files and every other machine readable resource included in the repository, but surely not the documentation of the *README* file, which often, but not always, comprehends a list of the services.

4.1 State of Art

This assumption is not so trivial, because that of recovering the list of microservices in a MSA is still an open problem and it is not trivial itself.

To the best of my knowledge, no one has ever tried to set up a method only for the detection of microservices; usually the aim has been that of recover the architecture in its (almost-)entirety (*Software Architecture Recovery*), i.e. also dependencies and/or other relations between microservices and perhaps also infrastructural components; so for this reason almost all methods found in literature are dynamic or hybrid static-dynamic. Obviously getting the number of microservices from the entire architecture is possible, but the effort of a dynamic method, understood both as time and computational effort, is excessive just for enumerating them.

Baresi *et al.* [36], facing a study similar to this one in *Empirical Software Engineering*, have set up a static black-box method that exploits the information in the *docker-compose* file to get the microservices. They discern the

microservices from the infrastructural components by looking at the container's image name and looking for the presence of common infrastructural components' names (i.e. databases, monitors, gateways, service discoveries, buses and servers). The blacklists cannot never be exhaustive, so false positives are quite frequents; moreover in some particular cases, the repository can contain one of the blacklisted words in its name and so the microservices' images, causing the methods to fail with no detected microservices. Also the choice of *docker-compose* is rudimentary, because it chooses the first file named `docker-compose.yml`, but: (i) it is not for sure that it is the one which lists the microservices and moreover the *docker-compose* can have custom names (trivially it could have the other version of extension), so basically in some repositories this method is useless; (ii) in case of multiple *docker-compose*s choosing the first one that is found by the research brings in some cases to different selection in consecutive commits, causing the undesired effect that some commits are evaluated on a *docker-compose* while other commits on another (although the first is still present).

Soldani *et al.* [38] [37] have developed a prototype tool named μMINER that implements a hybrid static-dynamic method for deriving the topology graph model of the architecture of a microservice-based applications, starting only from their deployment, specifically the *manifest* file of *Kubernetes*. After a first step that mines information from the aforementioned file, the application is run in order to collect interactions between components by sniffing the network. As final step the graph is refined analyzing the exchanged packets.

Alshuqayran *et al.* [39] in their hybrid static-dynamic method have adopted a white-box approach; indeed in the static phase they gather information about software artifacts extracting them not only from the *docker-compose*s and *Dockerfile*s but also from *Maven POM* files and even from YAML configurations, Java source code and documentation. After

having reverse engineered the code to obtain the UML class diagram, other data is collected by running the application; then all the information are processed by various steps that aim to determine the architectural concepts and the microservices are identified, and at the end some mapping rules associate them to implementation artifacts.

Rademacher *et al.* [40] also have adopted a white-box method, although static, where they gather information from *docker-compose*, *Dockerfile*s and source code, trying to identifying microservices from Java classes that employ annotations for web-based data-binding and from *Dockerfile* base images. All the data is then analyzed with also configuration files and interactions listed in the *docker-compose* to reconstruct the application's model.

Granchelli *et al.* [41] have developed a tool named MicroART that implements an hybrid static-dynamic semi-automatic method. A GitHub analyzer collects information from *docker-compose* and *Dockerfile*s, then a Docker analyzer collects runtime data in order to know the interactions between microservices. Before the final step, that aims to refine the reconstructed model, an architect is required to identify and mark the service discovery service.

4.2 Overview on Problem and Solution

4.2.1 Problem Formulation

Summarizing, the problem is the following:

> *Given a MSA project determine of how many and of which microservices is made.*

The solution should be a method that is:

- **static**, i.e. does not run the code, in order to be extremely lightweight, and

- **black-box**, i.e. does not use the source code, in order to be language independent.

4.2.2 Proposed Solution

The solution I propose exploits *Docker*[1] configurations, since it is the most common containerization platform. So the artifacts I exploit are the configuration files of *Docker*, the *Compose file*s (from here referred as *docker-compose*s, as they are better known) and the *Dockerfile*s.

More on... *docker-compose* Infobox 4.1

The *docker-compose*, as said in the official documentation, "is used to configure your Docker application's services, networks, volumes, and more". It defines:

- services (i.e. containers) with their names, the images they run and/or the build information, the running configurations, the environment variables, the networks they are attached to, the ports they expose, the volumes they can have access to and many other technical data;
- networks that allow services to communicate with each other;
- volumes that store persistent data and that can be reused by multiple containers;
- other technical configurations.

More on... *Dockerfile* Infobox 4.2

The *Dockerfile*, again as said in the official documentation, "contains all the commands a user could call on the command line to assemble an image". It is basically the recipe for building an image starting from a base image specifying:

continue...

[1] www.docker.com

...continued

- which files to copy in the image's filesystem;
- which ports to expose;
- which commands to execute when building the image;
- which instruction set to execute when running the image.

The *docker-compose* can be useful to collect information about the containers that form the application, both microservices and infrastructural components. The *docker-compose* (as explained in Infobox 4.1) contains a lot of information about the containers, however in order to only get a list of microservices only a subset of them could be useful, as described in following sections.

The *Dockerfile* can be used to understand how a container was built and, as consequence, if it is a microservice or an infrastructural component. The *Dockerfile* (as explained in Infobox 4.2) contains a lot of information, but also in this case only some of these information can help to detect microservices and they will be described in next sections.

Given that these are the only processed artifacts and that they are scanned for information statically, this method is black-box and static.

4.3 Concept

My solution is based on 3 macro-steps as illustrated in Figure 4.1:

1. first of all **locating the *docker-compose*** is required: this means not only to find where it is in the filesystem (that would be a relatively easy task), but also which *docker-compose* to select, because there can be more than one *docker-compose* in a project;

2. once it has been selected, the ***docker-compose* should be mined**, in order to extract a list of containers that it runs and

3. in the end **discerning microservices from infrastructural components** is necessary, and this could be done thanks to the information extracted in the previous step and the *Dockerfiles*.

Now it will be shown each phase with a description of how it has been divided into subtasks, which are the problematic aspects and how they have been conceptually addressed; a more technical overview, with pseudocode, will be presented in the next section.

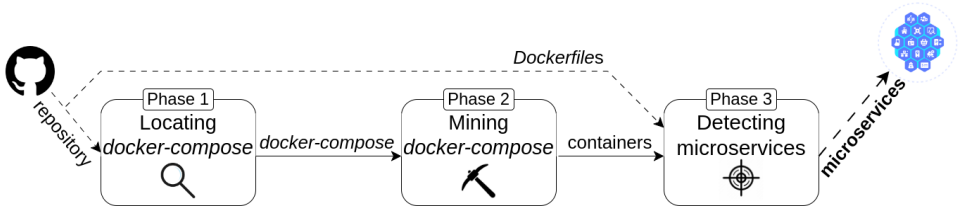

Figure 4.1: Microservices detection workflow

4.3.1 Locating *docker-compose*

The research and selection phase of the *docker-compose* is only one of the difficulties of the whole method, but it is the first that shows up. Both the sub-phases are not easy to perform.

The root cause of all the difficulties is the non-uniqueness of the *docker-compose* (and so the capability of having non-fixed names to differentiate them).

In a project there can be multiple *docker-compose*s for various reasons:

- some *docker-compose*s are "duplicated" because they have different aims, for example there can be `docker-compose.dev`, `docker-compose.test` and `docker-compose.prod` that respectively refer to the one to be used during the development, the one for the testing phase and the one for the production;

- some *docker-compose*s have complementary purposes, that is they list different groups of containers that all together form the application, for example a `docker-compose.microservices` and a `docker-compose.infrastructure` that deal respectively the microservices and the infrastructural components;
- some *docker-compose*s are partials, in the meaning that they are used only in particular cases to override some containers' configurations defined in other *docker-compose*s (often these *docker-compose*s do not even specify image or build information).

And all these cases are orthogonal, so it is also possible to deal with a combination of them.

Then the *docker-compose*s can be in any directory, because there is not a standard one documented and neither a de-facto standard one used in practice. Many put it in the base directory of the project or in the code directory, as many put it in other directories different one from another.

A controversial corner-case that I met during the preliminary manual inspection of a certain number of repositories, with the aim of collect enough experience on various practices, is that of defining a *docker-compose* per microservice. This practice, although not very widespread in absolute terms, is more widespread than one might expect, since it is a totally useless practice: indeed the *docker-compose* serves to run and configure multiple containers correctly configured to work all together. In case of single container, it can be configured directly from the run command. Probably the goal of who follows this practice is that of having one single command (`docker compose up`) that works indiscriminately for all the microservices, thus avoiding individual documentation of commands.

The only not-so-bad news regards the filename of *docker-compose*s. Indeed, although it is completely free, two standard names exist: `docker-compose` and `compose` (to put in Cartesian product with the two versions of

YAML[2] extension: `.yaml` and `.yml`). In case of multiple *docker-compose*s the documentation suggests to add a suffix to the standard names, and in fact this is the most common practice, but it is quite widespread also the use of a prefix. Basing on my experience, the cases of completely custom filenames are very few.

Following each step of this phase (shown shortly in Figure 4.2) is treated individually, overviewing how it is useful to address the just mentioned difficulties. A more detailed discussion on how each solution can be technically performed will be at Section 4.4.

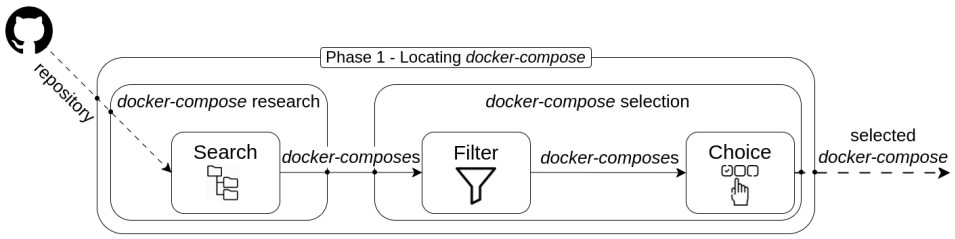

Figure 4.2: Phase 1 workflow: *docker-compose* location

*docker-compose*s research

The research phase is that with the easiest solution: searching the *docker-compose*s by extension is not feasible because the YAML format is a common configuration format used by many services and neither by filename because not fixed; however, as previously explained, the almost-standard names (i.e. the two standard names with the two versions of extension) can be searched admitting also prefixes and suffixes.

docker-compose selection

The selection phase is by far the most complex and difficult. The aim of this phase is to select the *right docker-compose* among all those present,

[2] www.yaml.org

where "right" means *the one that lists all the microservices.*

This aim represents also the renounce to those cases with a *docker-compose* per microservice: indeed this cases can mislead a selection algorithm because neither just one of these *docker-compose*s nor all together cannot give any information about the *overview* on architecture (indeed it is not obvious that for the correct run of the application all the present *docker-compose*s should be invoked, e.g. some of them can refer not to applications components but to developing or testing utilities, although they are indistinguishable from the real microservices).

In conclusion the idea for addressing this phase is that of filtering the collected *docker-compose*s in order to discard those considered unacceptable, according to some rules, and then being able to select from the remain results the most suitable one (or better *the most likely right one*). This can be done by defining some empirical rules to assign a priority to each of them based on path and filename; after that just by selecting that with the highest priority.

4.3.2 Mining *docker-compose*

Also the mining of *docker-compose* is not trivial, but luckily it is not necessary to resort to any heuristic, since it is sufficient to follow the official documentation of Docker Compose[3].

The information that can help in microservices detection is:

- the **list of Docker services** that the *docker-compose* runs on invocation and
- for each of them:
  - the **image** of the service, i.e. the image run by the container that backs the service;
  - when present, the **build** of the image (formed by a *Dockerfile* and a *context*), and

[3]https://docs.docker.com/compose/

– the **name** of the container that backs the service.

Other information is more related to the technical aspect of configuration.

However it is not obvious that all these data are coded in plain. Indeed looking at the containers that form the composite one at runtime it is possible to have some different or additional data, that will be explained soon. Anyway getting the information as they are at runtime is perfectly possible thanks to the fact that Docker documentation fully illustrates how Docker Compose works. These are the mechanisms that can resolve data at runtime:

- **environment variables**: values in the *docker-compose* can be set using variables that will be interpolated at runtime basing on the value of environment variables set in the `.env` file (or a custom *env* file specified as argument at invoking time or as attribute in case of *docker-compose* inclusion) or from the shell;
- **inclusions**: a *docker-compose* can declare dependency on another *docker-compose* by "including" it. Inclusions apply recursively and at runtime, once the included *docker-compose* is loaded, its content is copied in the *docker-compose* that includes;
- **extensions**: a service listed by a *docker-compose* can extend another service (from the same *docker-compose*, an included one or any external one); in this case the service that extends inherits all the configurations (and it can then override them) recursively up to the base service.

Resolving these cases as Docker Compose does at runtime, and as it is officially documented, allows to get all the information[4]just without the overhead of running the composite container (i.e. the entire application).

In Figure 4.3 an overview of this phase and following a brief description of each step. A more detailed overview will be discussed at Section 4.4.

[4]To be clear: "all the information" means all the information recoverable from coded settings, e.g. if an environment variable has to be set from command line when invoking Docker Compose, this data cannot be recovered for obvious reasons.

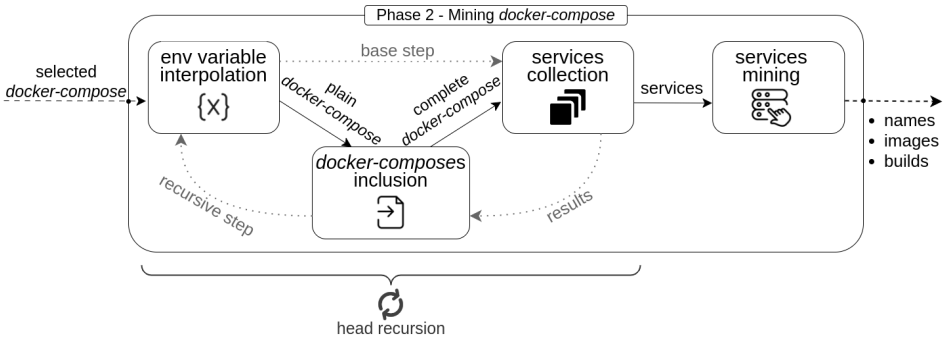

Figure 4.3: Phase 2 workflow: *docker-compose* mining

Environment variables interpolation

The first step is obviously that of resolving the variables, following the documented syntax, in order to have all the values in plain and this has to be done for the selected *docker-compose* as well for the recursively included ones. The environment variables values are taken from the default `.env` file (when it exists) for the selected *docker-compose*, while for the included ones from the specified *env* file (or the default `.env` if not specified).

*docker-compose*s inclusion

This is basically the recursive step: if there is some included *docker-compose*, it should be recursively mined to obtain its data. Since, as already said, these data can be used in the including *docker-compose*, a head recursion is needed. All the arguments defining inclusion should be taken into account to correctly interpret the inclusion, as explained in the official documentation.

Services collection

This step represents the base step of recursion: when there is no included *docker-compose* or when they are all already processed, then collecting the services listed in the current *docker-compose* is possible. In this step the

services extensions, if present, have to be unfolded recursively up to the base container (processing services either from an included *docker-compose*, so extended service has already been collected, or from an external one, looking for it in the referenced *docker-compose*).

The information inherited and the current ones specified should be merged, again as described in the official documentation, in order to perfectly replicate the behavior of Docker Compose.

At this point the information is made explicit.

Services mining

Now that all the services are collected they can be mined to take desired information from each of them, that as already said are: the image's name, the container's name (if specified, otherwise in its place the service's name, as officially documented) and, whenever it is present, the build (i.e. a *Dockerfile* and if present a *context*).

These are all the details needed for detecting microservices in the last phase.

4.3.3 Detecting microservices

This phase is the real core of this method. Starting from the data just collected, that just to remember are the image, the container name and the build, the aim is to discern real microservices from infrastructural components.

Notice that there are two main cases with a huge difference:

- in the first case the Docker service specifies the build: this is the easy case, indeed the *Dockerfile* that generates the image gives much more information than the image alone;
- in the other case, when there is not a specified build (i.e. developers have a separate workflow that builds the image and deploys it in a

image registry and the *docker-compose* pulls it from there), the only hope to have additional information for a precise detection is that of trying to match the service to one of the *Dockerfile*s present in the repository. And this task has to be accomplished basing only on image's and container's names.

Doing what just illustrated without any additional precaution leads to, more or less, the same results as a blacklisting approach based on images' names. Indeed it is absolutely not true that a *Dockerfile* corresponds to a microservice.

There are a lot of cases where developers have another not-so-virtuous habit: often *Dockerfile* is used to initialize or configure an infrastructural component's container (almost always a database). In other words instead of using the base image of the third-party component and configuring it through the *docker-compose*, they define and build a new image from the base one already configured and they use it in the *docker-compose* without configuration; from this the need of discarding this type of *Dockerfile*s (in order to prevent a match with a service).

Following each step of this phase (illustrated in Figure 4.4) is treated individually, showing how they, all together, can address the described difficulties. For more details, the Section 4.4 discusses how they are technically performed.

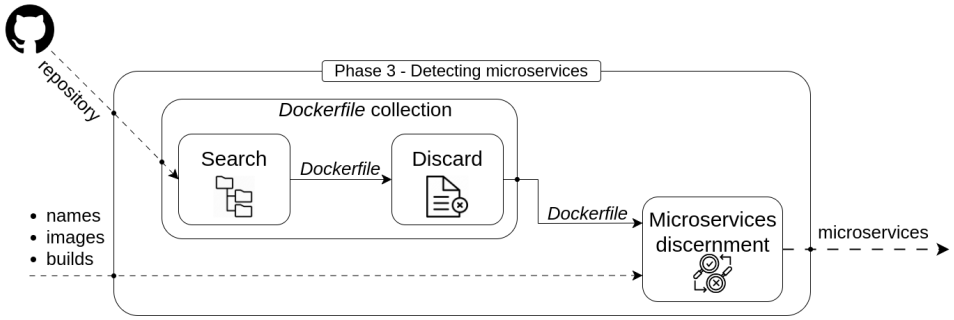

Figure 4.4: Phase 3 workflow: detecting microservices

*Dockerfile*s collection

Also the *Dockerfile*, as *docker-compose* does, can have a custom filename, but in its case practically everyone uses the standard name (possibly preceded or succeeded by an affix). And being that the *Dockerfile* does not have an extension, there is not any hope to recover those really few with a totally custom name.

So it is sufficient to search for files that contain "Dockerfile" in the name and to discard those false positives with an extension (often script files with hard-coded parameters to be run to invoke the build of the image).

As last step the *Dockerfile*s that are related to third-party vendor or those for example and demo purposes should be discarded to avoid easily avoidable false positives.

Microservices discernment

At this point almost everything is ready for the final step. The only aspect that remains to define is how, in case a service does not specify a build, matching it to a *Dockerfile* basing on its image's and container's names is possible.

Observing a large number of repositories it is possible to realize that very often these names, as is normal, contain the names of microservices they refer to. And the microservices' names also appear, obviously, sooner or later in the path of the directories containing source code and/or *Dockerfile* of each of them.

In conclusion the idea behind the match method is that of recover the name of the microservice from the data available and look for a match in the set of collected *Dockerfile*s: if a match exists and it is unique, then is (almost) possible to conclude that a microservice has been detected.

The very last thing to do, both in case of a referenced *Dockerfile* or a matched one, is that of checking if the *Dockerfile* really define a microservice

and it is not only for configuration/initialization. This can be done by looking at what type of files it copies in the image's filesystem: in case of configuration or script files only it should belong to an infrastructural component, otherwise if it copies some user code it should belong to a microservice.

4.4 Design

Now that the overview of the three phases has been presented, more technical details on how they work will be treated in the following subsections. For each step of the phases which aspects can introduce false positives and false negatives will be highlighted, in order to have a clear view of how the entire method's results can be influenced.

All the rules and parameters that will show up soon are tuned empirically basing on the observation of a large number of microservices architectures. The design phase is the result of multiple rounds of generalization: the rules have been made gradually more generic in order to apply to a greater number of repositories (they have never been made more specific in order to apply to corner cases, indeed some of them still remain uncovered by this method).

4.4.1 Locating *docker-compose*

The process followed by this phase is shown at high level in Algorithm 4.1. Following all the tasks of which is composed are illustrated in details.

Algorithm 4.1 *docker-compose* location

Input: repository
Output: selected *docker-compose*

1: Look for *docker-compose*s
2: Discard *docker-compose*s with "non-neutral" folder in path
3: Group *docker-compose*s by path
4: Order groups by path preference
5: **for each** group of sibling *docker-compose*s

continue...

...continued

```
if a docker-compose without affixes exists
  return it
else
  Discard docker-composes with "non-neutral" affixes
  Discard docker-composes with undesired affixes
  Order docker-composes by affix priority
  if some docker-composes still exist
    if only one docker-compose has the highest priority
      return it
    else
      return that with the shortest filename
```

Looking for *docker-compose*s

This task is completed simply by searching all files named

`*docker-compose*.yaml`, `*docker-compose*.yml`,
`*compose*.yaml`, `*compose*.yml`.

This task can introduce false negatives, i.e. those *docker-compose*s that have a totally custom name, anyway these cases are very few.

Discard *docker-compose*s with "non-neutral" folder in path

This task aims to discard those *docker-compose*s relative to single microservices. It discards all the *docker-compose*s whose path contains at least a folder considered "non-neutral", where the neutral ones are listed below and are referred mainly to Docker, code directory, software lifecycle phases and generically to microservices:

`docker`, `compose`, `swarm`,
`src`, `services`,
`dev`, `test`, `staging`, `deploy`, `integration`, `release`, `prod`,
`iac`, `saas`, `devops`, `setup`, `script`, `complete`, `etc`.

Folders should *contain* one of these words, they do not have to be exactly equal, so also `docker-compose`, `developing`, `testing`, `production` and `scripts` are good.

When a folder does not contain any of these keywords it should be referred to the name of a single microservice, this is why it is discarded.

These keywords turned out to cover almost all cases, but both false positives and negatives are possible (e.g. *`abbreviated-repo-name`*`/compose.yml` is probably a false negative).

Order groups by path preference

*docker-compose*s preference order cannot rely on their depth in the filesystem because some directories are clearly preferable to other regardless the depth of the *docker-compose*s (e.g. `development/docker/compose.yml` is preferable to `test/compose.yml`). To give more importance to the directories, a preference order among the aforementioned keywords is defined and then the groups are ordered lexicographically with respect to the path. The preference order is that in which the keywords appear above (Docker related keywords, then code directory, software lifecycle phase and in the end miscellaneous ones).

Discard *docker-compose*s with "non-neutral" affix

To avoid *docker-compose*s that refer to particular contexts (and that probably do not list microservices), neutral affixes have been identified and all those *docker-compose*s that present different affixes are discarded. The neutral affixes, referring mainly to software lifecycle, are listed below:

`services`, `base`, `dev`, `build`, `stack`, `prod`, `stable`, `deploy`, `test`.

Also in this case the keyword is sufficient to be contained (not exactly), so also `development`, `production`, `deloyment` and `testing` are included.

Just to give an example of a *docker-compose* that can be discarded at this step: `compose.mysql.yml` most likely contains only configuration for the database.

Also this step can possibly introduce false positives and negatives, but rarely, because that of identifying the *docker-compose* with the phase of lifecycle is a quite common convention.

Discard *docker-compose*s with undesired affixes

In case some *docker-compose*s have more affixes, those with undesired ones are discarded. The undesired ones are `infra` and `override` because in almost the totality of cases where they are present they refer to *docker-compose*s that, respectively, treat infrastructure components and override configurations.

Order *docker-compose*s by affix priority

To select the most likely right *docker-compose*, the remains *docker-compose*s are ordered by the affix priority, where the priority is the above order of appearance: initial software lifecycle phases are preferred in order to appreciate the changes without delays.

Returning *docker-compose* with the shortest filename

In case (quite rare) of more *docker-compose*s with the same priority, in order to make the choice deterministic (so as not to make different choice in different commits), that with the shortest path is chosen. But it is not a heuristic, so it is not necessarily the best choice.

4.4.2 Mining *docker-compose*

Regarding the process followed in the second phase only the last sub-phase, i.e. the services mining, is explained since for the previous ones, where the

docker-compose is reconstructed following its syntax, following the official documentation is sufficient.

In Algorithm 4.2 an high level overview, that for its simplicity is the faithful copy of the implementation. It is interesting to comment two aspects: (i) as name has been chosen the container's name when present, because the service's name is a symbol more for internal use purpose and (ii) when extracting the build information is extremely important to preserve not only the *Dockerfile* information, but also the *context*.

Algorithm 4.2 Services mining

Input: list of Docker services from the selected *docker-compose*
Output: list of service tuples (image, container name, build)

```
1:  for each service
2:
3:      if image field exists
4:          Take that as image
5:
6:      if container's name field exists
7:          Take that as name
8:      else
9:          Take service's name as name
10:
11:     if build field exists
12:         Take that as build          ▷ preserving context information
13:
14:     Append (image, name, build) to a list
15: return tuples list
```

4.4.3 Detecting microservices

The process followed in the last phase is by far the most complicated to explain, but it will be treated step by step. In Algorithm 4.3 an overview at high level, then all the steps will be revealed in details.

Algorithm 4.3 Microservices detection

Input: list of service tuple (image, container name, build)
Output: list of microservices

```
1:  Order services by descending length of image and container name
2:  for each service
3:    if service specifies build
4:      if Dockerfile is local
5:        if Dockerfile exists
6:          → Verified Microservice ←        ▷ if Dockerfile copies some code
7:        else
8:          → Unverified Microservice ←
9:      else
10:       Not a Microservice                 ▷ infrastructural component
11:     elseif service specifies image
12:       Look for all Dockerfiles
13:       Look for a match with image
14:       if it exists
15:         → Matched Microservice ←         ▷ if Dockerfile copies some code
16:       else
17:         Look for a match with name
18:         if it exists
19:           → Matched Microservice ←       ▷ if Dockerfile copies some code
20:         else
21:           Not a Microservice             ▷ infrastructural component
22:   else
23:     Not a Microservice                   ▷ abstract base service
24:
25:   if it is a microservice
26:     Validate Dockerfile
```

Order services by descending length of image and container

Here the aim is, as already mentioned, to privilege the more accurate match leaving last the more "risky" ones, that is to say those that will lead more likely to a false positive.

Look for *Dockerfile*s

This step is apparently easy, but it hides some difficulties, indeed in Algorithm 4.4 it is possible to see that some precautions have been adopted.

In order to get all *Dockerfile*s it is sufficient to search for files whose name contains `*Dockerfile*`, but in this way also some false positives are collected. They are mainly script files used to hard code some build parameters, so that the build can be run just by running these files instead of having to write the command with all the arguments, or text files for writing down the command.

These false positives can be removed in a "simple" way: given that the *Dockerfile* has not an extension, removing all the results with any extension is sufficient. The problem is that since filenames can contain dots (and often they do, using them as separator with the affixes) recognizing the extension is not possible, but rather detecting those with recurrent extensions is possible. The recurrent extensions identified in these cases are:

`.sh`, `.ps1`, `.nanowin`, and `.txt`.

At this point also *Dockerfile*s relating to third-party services and for demo purposes have to be filtered and this is performed by discarding those that contain in the path folder like

`vendor`, `external`, `example`, and `demo`.

But a last step is needed to avoid problems during match attempts: some directories can have more than one *Dockerfile* and this will invalidate the match because a match is only a match when there is no doubt on the *Dockerfile* to associate to a service. If more *Dockerfile*s have the same path, automatically there is a multiple pairing and this prevents the match. To avoid double matches all *Dockerfile*s with same path are discarded except one: that with the shortest filename (again, this is not a heuristic, but only a way to make the choice deterministic).

Algorithm 4.4 *Dockerfile* research

Input: repository
Output: list of *Dockerfile*s good for matches

1: Research of all *Dockerfile*s files
2: Discarding of false positives
3: Discarding of *Dockerfile*s for external services or demo purpose
4: Grouping by path

Look for a match with image/name

To look for a match between a service and a *Dockerfile* using image's or container's name is quite simple (as described in Algorithm 4.5). Just need to list the possible names of the potential microservice identified by the Docker service, order them by descending length (again to privilege more accurate match) and selecting those *Dockerfile*s that contain the name in the path.

Algorithm 4.5 Match attempt

Input: service, *Dockerfile*s list
Output: matched *Dockerfile* (if possible)

1: List possible names of eventual microservice
2: Order name by descending length of name
3: **for each** name
4: Select *Dockerfile*s that contain it in the path
5: **if** there is one and only one
6: ***That is a match***

The only aspect to reveal is: how the potential microservice's name can be obtained from image's or container's name.

In case of image's name, it will be something like this:

```
({registry, dockerHubUser}/)?(user[-_/])?(repo[-_/])?microservice
```

and, discarded the first part that is useless, up to four possible names can be detected, as indicated above.

In case of container's name, instead, up to two possible names can be detected as indicated below (the first two are alternatives).

```
({srv, (micro)?service}[-_/])?microservice([-_/]{srv, (micro)?service})?
```

Validate *Dockerfile*

As already mentioned in the previous section, both in case of verified or matched microservice, it is required to check if the *Dockerfile* really refers to a microservice by verifying which types of files it copies in the container's filesystem; indeed if it only copies script and configuration files, indicated by the following extension

`.sh`, `.xml`, `.txt`, `.yaml`, `.yml`, `.sql`, `.conf`, `.config`, `.cnf`, `.cfg`, `.cf`, `.crt`, `.key`,

it will be discarded as infrastructural element.

This step can introduce two types of false positives. The first is quite similar to all the others already mentioned and it happens when script or configuration files have an extension different from those identified empirically. The second case is quite different: a *Dockerfile* can copy an entire directory in the container's filesystem and it is quite frequent that all the configuration files are stored in a folder to simply copy them all together. For a really accurate check all the files contained in the copied directories should be verified (but when it deals with code directories the number of files they contain can be really huge).

4.5 Implementation

This method has been implemented as a small Python library that makes available the functions for: (i) selecting *docker-compose*, (ii) extracting the list of Docker services and (iii) determining the microservices from them.

Also the minor steps are available, but the configurations can be easily tuned directly acting on some macros.

The code is publicly available on GitHub[5].

4.6 Preliminary Effectiveness Evaluation

Being this a totally new approach for detecting microservices before even evaluating quantitatively its accuracy and efficiency, a preliminary qualitative evaluation on its effectiveness is required.

This evaluation had the only intent of verifying that the method just presented has solid foundations. For this reason I did not rely on rigorous criteria for defining false positives and false negatives, but rather I used criteria based on instinct and experience. Clearly the next step is necessarily to conduct a real and rigorous quantitative evaluation.

The targets have been both the *docker-compose* selection and the microservices detection (not the intermediate phase because it is purely deterministic). The tests were about to check if:

- given the repositories, and so the set of *docker-compose*s, the method has chosen the right one and
- given the set of Docker services and the set of *Dockerfile*s, the method has detected correctly the microservices.

As already said the verification was instinct-based, without rigorous criteria, so basically the positive or negative result was given basing only on the names of Docker services and the paths and filenames of *docker-compose*s and *Dockerfile*s; however in case of unclear situation, the inputs were analyzed in its content too.

The results are definitely positive and give rise to hope for the quantitative evaluation. The *docker-compose* selection was right in almost the totality of

[5] github.com/KevinMaggi/detection-and-identification-of-microservices

cases, except just a few corner cases. The microservices detection was a little more imprecise, but still at more than satisfactory levels. The false positives were extremely limited and perhaps even almost all fixable implementing the already mentioned in-depth mechanisms for validating the *Dockerfile*s; the false negatives were definitely more, but concentrated in those cases where the repository was not very schematic (e.g. when microservices' name were extended somewhere and contracted somewhere else, effectively preventing a match).

4.7 Conclusion

Waiting for a more reliable quantitative evaluation of accuracy, I can conclude for now that this method has some strengths:

- the *docker-compose* selection has a high case coverage and efficiency;
- the *docker-compose* mining is complete, in the sense that following all the mechanisms documented by Docker is possible to recover all the available information;
- the microservice detection has on average a fairly high accuracy, but very high in those repositories that follow some convention/good practice;
- its lightweightness is clearly a big advantage for those studies in Empirical Software Engineering that are interested only in the number of microservices, indeed the reduced time needed allows also to execute the analysis commit-wisely in a reasonable time.

Clearly is not perfect, so it has also some weaknesses:

- the *docker-compose* selection in some cases, although very limited, goes wrong;
- the microservices detection presents some false negatives, above all in those repositories outside the box;

- possible infrastructural components developed as part of the repository (and whose code and *Dockerfile* are in the repository) are recognized as microservices, because at the current state they are extremely hard to discern from real microservices.

4.8 Future Work

Refining the rules for both the *docker-compose* selection and the microservices detection is obviously possible by observing a bigger dataset of repositories and by introducing an in-depth mechanism for validating the *Dockerfile*s; however the next mandatory step to do is the quantitative evaluation of accuracy. It can be done following two approaches: (i) applying the method to a dataset of repositories with known architecture and structure (so the ground truth) or (ii) defining precise criteria to identify which false positives and false negatives are.

Chapter 5

Experiment Execution

Returning to the experiment, in this chapter I discuss just some aspects that turn out during its execution, while its results will be presented and discussed in the next one.

5.1 Dataset Definition

The queries retrieved a total of 2491 single repositories (so collapsing repositories resulting from multiple queries into a single result).

FC_1 removed the repositories without a *docker-compose* reducing the set of results to 686. During this filtering step one repository has been verified by hand, being that it causes the script to crash (actually it causes a crash of `git clone`, likely due to its huge size).

FC_2, FC_3, FC_4 removed repositories with too short history, non industrial-like development and too few commits with an acceptable *docker-compose*, reducing the set to 198, 186 and 121 results.

FC_5 was the manual criterion (the last one before final selection): from the 121 repositories, only 46 revealed to be a real or industrial demo MSA.

At the end the parameters of FC_6 have been tuned to find a configuration that would have given between 10 and 20 results: such a configuration has been found with 15 final selected repositories, that are shown in Table 5.1.

The defined dataset is enough heterogeneous under different point of view, first of all the main languages of repositories that are somehow representative of the languages diffusion showed in JetBrain's survey [49]. For this reason it

can be considered satisfactory for the present study.

ID	Repository	Main language(s)	Type
$S01$	1-Platform/one-platform	TS	Real system
$S02$	OpenCodeFoundation/eSchool	C#	Real system
$S03$	ThoreauZZ/spring-cloud-example	Java	Industry demo
$S04$	asc-lab/micronaut-microservices-poc	Java	Industry demo
$S05$	bee-travels/bee-travels-node	JS	Industry demo
$S06$	dotnet-architecture/eShopOnContainers	C#	Industry demo
$S07$	geoserver/geoserver-cloud	Java	Real system
$S08$	go-saas/kit	Go	Real system
$S09$	jvalue/ods	TS, JS	Real system
$S10$	learningOrchestra/mlToolKits	Python	Real system
$S11$	microrealestate/microrealestate	JS	Real system
$S12$	minos-framework/ecommerce-example	Python	Industry demo
$S13$	nashtech-garage/yas	Java, TS	Industry demo
$S14$	netcorebcn/quiz	C#	Industry demo
$S15$	open-telemetry/opentelemetry-demo	TS	Industry demo

Table 5.1: Selected repositories (more details in Appendix A)

5.2 Dataset Analysis

5.2.1 Data Mining

During the execution of analysis some commits get lost due to the building process in the systems that require it. Indeed during the building with the automation tool some commits had not built successfully. In case of long chunks of consecutive commits failing in build and in any case whenever possible, some fixes have been made to the Maven POM, Gradle build script and .NET `global.json` file, to resolve the error allowing the build to success. Interventions mainly concerned about dependency version updates, formatting corrections and platform/build automation tool version update.

Although the build has been "forced" when possible (i.e. allowing to the

build not to fail on errors when they were non-blocking and apporting the just mentioned fixing), in some cases of build failure the error was not clear or an objective solution did not exist, so rather than using a subjective heuristic to fix the issue, I opted to discard the affected commits.

These cases are mainly related to: (i) dependencies (or repositories of dependencies) that do not exist anymore, (ii) dependencies that make the build to fail, (iii) absence of build file (in the very initial commits) and (iv) errors in the build file that are not objectively fixable.

The occurrence of these cases in systems is showed in Table 5.2. Given the relatively low number of skipped commits (the total incidence on the whole dataset is less than 1%) I do not believe that this factor could have noticeably influenced the results. Further considerations are reported in the threats to validity Chapter 7.

ID	Commits	Omitted commits	
$S01$	1502	0	0%
$S02$	275	17	6,18%
$S03$	279	5	1,79%
$S04$	384	6	1,56%
$S05$	375	0	0%
~~$S06$~~			
$S07$	1027	11	1,07%
$S08$	629	0	0%
$S09$	1428	26	1,82%
$S10$	1214	0	0%
$S11$	412	0	0%
$S12$	1069	0	0%
$S13$	535	0	0%
$S14$	658	27	4,10%
$S15$	572	0	0%
Total	10359	92	0,89%

Table 5.2: Analyzed commits per system

The system $S06$ during the analysis phase has been discarded because of

severe building issues: apparently (a high number of) older commits are not anymore buildable with actual versions of .NET (even modifying requested version on configuration files), probably because, being a very long-living system, the newer versions break backward compatibility with the used ones. Actually only a minority of commits was buildable, so I decided to exclude it.

5.2.2 Data Analysis

After the analysis data collection, a suspicious TD constantly equal to 0 for a long time has been discovered in $S11$; a manual inspection has revealed that in its initial phase (and for a considerable portion of its history) it was a multi-repositories system. Due to this, it has been excluded from data analysis phase.

Chapter 6

Results

After executing the experiment, the results can be put together and used to finally answer the RQs. This chapter intends only answer the RQs, while a more in-depth discussion on the meaning of findings is conducted in Chapter 8. In this chapter only a summary of results is showed in order to facilitate their exposure, but a complete report of outcomes is in Appendix B.

6.1 RQ_1: TD evolution trend in MSA

Mann-Kendall trend test gives immediately a clear overview on the overall trend of TD in a MSA; in fact all the 13 systems revealed the presence of a trend ($p–value \leq 2,2e{-}16$). Moreover, as possible to see in Table 6.1, almost all the systems have a strong or very strong trend (i.e. a Kendall's τ close to 1 in absolute value); only $S15$ has a quite weak trend with a Kendall's τ equal to 0,23. For all the systems it is about a growing trend, as could be expected, except for $S14$ that has a quite strong negative trend, representing an exception since usually the TD tends to accumulate, and for this it deserves to be explored further.

ID	τ
$S01$, $S02$, $S03$, $S04$, $S07$, $S08$, $S10$, $S13$	$\tau \geq 0,79$
$S05$, $S09$, $S12$	$0,49 \geq \tau \geq 0,59$
$S14$	$\tau = -0,58$
$S15$	$\tau = 0,23$

Table 6.1: Kendall's τ on TD trend

Going to see the trend obtained from LOESS regression is possible to see that almost all the systems follow the same trend characterized by a strong growth in the initial phase of development, followed by a sudden decline in the growth rate (although TD continues to grow, as showed in the example in Figure 6.1). I can conjecture that the elbow point corresponds to the passage from the initial intensive development to maintenance phase, since younger systems either showcase a smoother change (like system $S08$) or do not present at all a variation in the trend (it is the case of systems $S07$ and $S13$). This conjecture can find confirmation in the fact that after the elbow point the commit frequency decreases, sometimes also in a significant way, and going to see the original TD evolution it is possible to see notable plateaus.

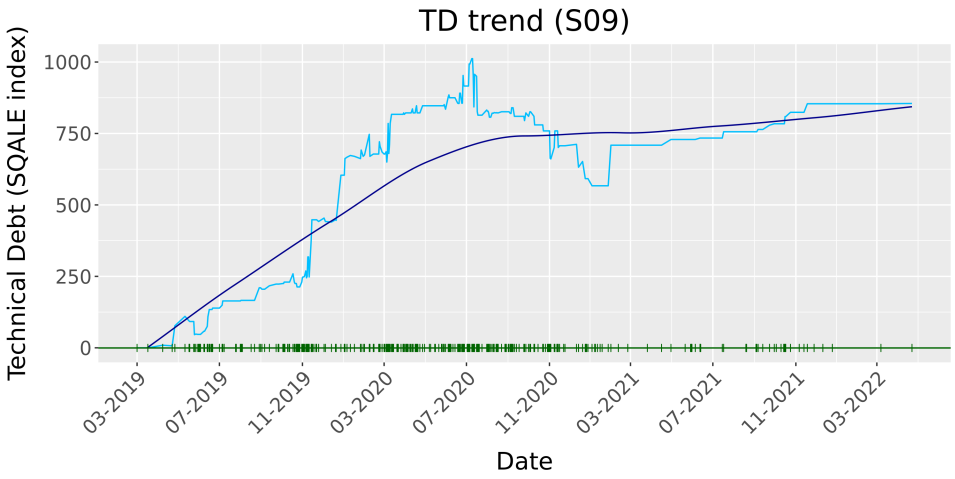

Figure 6.1: Trend of TD in system $S09$

Regarding systems $S14$ and $S15$, those with a weaker or negative trend, they show a similar trend characterized by a high initial TD followed by a constant decreasing until a minimum and then a slow increasing. $S14$ in the end stabilizes at a value lower than the initial (reason why the Kendall's τ is negative), while $S15$ keeps growing, as showed in Figure 6.2.

The TD evolution suggests an insight on the initial phase, that reveals a common factor: both the systems start with a considerable quantity of already

developed code (so with a certain TD), and the following phase consists on various refactoring that inevitably lower the TD accumulated in early commits (or, in practice, even before the first commit).

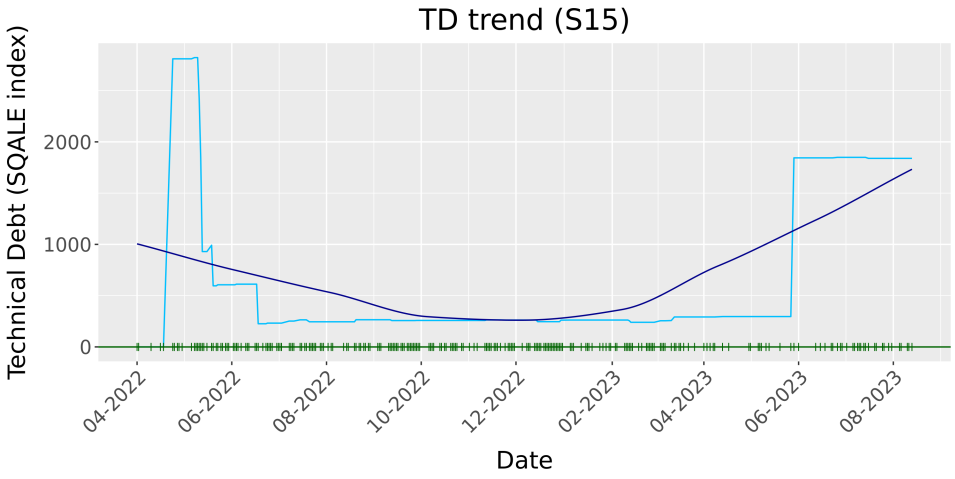

Figure 6.2: Trend of TD in system $S15$

Basing on these results I can already reject the null hypothesis $H_0^{1.1}$ with statistical confidence, concluding that in MSA systems TD generally tends to grow over time

As further investigation, the top 10 hotspots in TD have been examined qualitatively. This investigation reveals that adding a component (be it a microservice, an infrastructural component or a fronted UI) is the activity that absolutely introduces more TD, immediately followed by an evolution of the business logic. Anyway these are not the only activities that can introduce TD, since in other cases this has been increased also by an upgrade of a library/package or even by a refactoring. However, as one might expect, refactoring, with fixing and microservice removal to a lesser extent, is the main activity when a significant part of TD is paid back. This double face of refactoring is probably related to its intent, either to clean up the code (and so paying back TD) or simply to slightly modify it in order to getting ready for further development.

The last inspected aspect regards the "size" of commits (i.e. changed files

and added/deleted lines): although it is more likely that a big commit impacts more TD than a smaller one, in many cases small commits introduced a notable TD suggesting that the relation between these quantities is not so strong.

Moving on the seasonality, the Ollech & Webell combined test returns a negative response for all the considered systems (i.e. a *p–value* greater than 0,01 and 0,002 in the individual tests), confirming what it was also evident visually, namely that there is no seasonality. Contrary to what one might expect, therefore, TD is not more likely to be introduced/paid back in certain period of the year (namely before/after seasonal holidays). Clearly for none of the systems the STL decomposition has been performed.

This gives enough statistical evidence to reject null hypothesis $H_0^{1.2}$ on the absence of a seasonality.

RQ_1 answer (*TD evolution trend in MSA*)

TD displays an overall increasing trend in time, above all in the initial development phase, albeit the rate decreases in advanced phases characterized by several (long) periods with limited TD introduced.

TD variations can happen both adding microservices or evolving the business logic, but a variety of other activities can also be the cause. Refactoring is the main tool to pay back TD, but not always and not the only tool.

TD does not present seasonality, meaning that it could be introduced throughout the year in the same way.

6.2 RQ_2: relation between TD and microservices

In order to study the potential correlation between TD and number of microservices, I start by graphically inspecting the time series of the two metrics. As it can be seen in the case reported in Figure 6.3, the two evolutions seem to display an overall similar growth; however the correlation does not appear to be always present in all commits. As example I can consider the $S13$ system

reported in the figure, when the removal of a microservice in mid-April 2023 does not correspond to any variation in TD; or at the end of February 2023 when a sudden fall of TD does not match with a variation in microservices number.

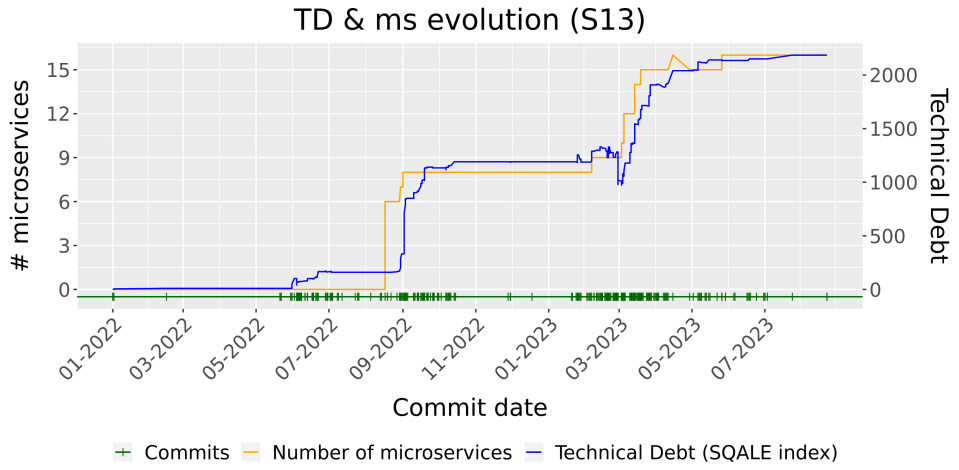

Figure 6.3: Evolution of TD and microservices number in system $S13$

Anyway some other cases show an even weaker correlation, as showed in Figure 6.4 ($S08$), where the TD has an almost constant growing trend while microservices number has a peak in the middle of development and then decreases.

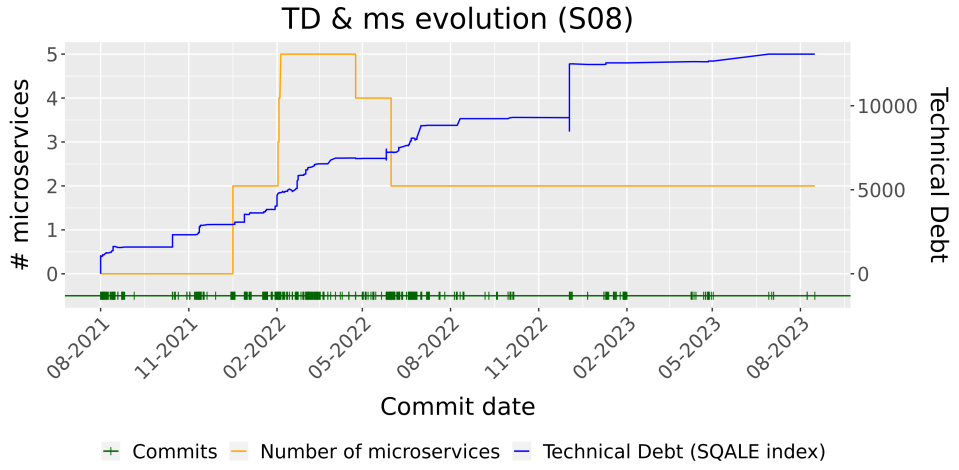

Figure 6.4: Evolution of TD and microservices number in system $S08$

In order to clarify this aspect with statistical meaning, the Cross-Correlation gives formal results considering also possible lags. As Table 6.2 summarize, 9 systems out of 13 show a strong or very strong correlation at some lag, and only 4 do not. All the systems characterized by a correlation, show the strongest correlation at negative or no lags (meaning that the TD never precedes microservices number), except for $S15$ that has the strongest correlation at a positive lag, albeit some negative ones are present also at negative lags (showed also in Figure 6.5).

ID	Cross-Correlation (at some lag)
$S01$, $S02$, $S09$, $S10$, $S15$	very strong ($>>$ confidence level)
$S07$, $S12$, $S13$, $S14$	strong ($>$ confidence level)
$S03$, $S04$, $S05$, $S08$	absent or very weak ($<$ *or* $\approx$ confidence level)

Table 6.2: Cross-Correlation between TD and microservices

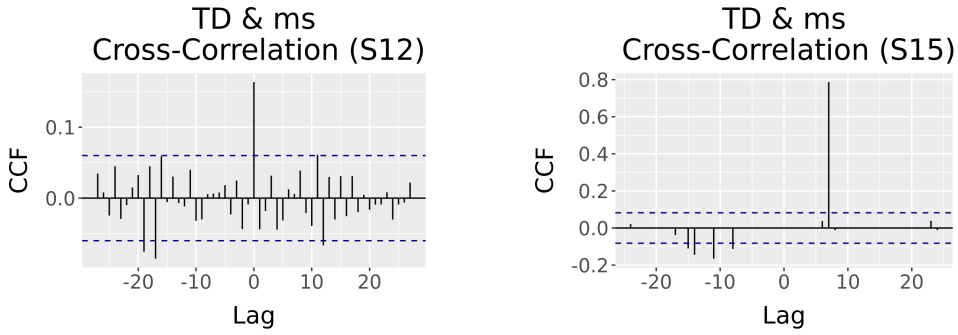

Figure 6.5: $S12$ and $S15$ TD and microservices CCF

The 9 systems where the two time series correlate (in some measure) have been subjected to the Granger causality test to determine if they only correlate or if the microservices time series can be useful to forecast future TD trend being a relation of causality between them. Of these, only 4 (listed in Table 6.3) revealed a causality, with a $p\text{–}value <$ or $<< 0.01$. Among them also $S15$ appears, although, as seen before, the strongest correlation was not one of those at negative lags; evidently in this case, albeit a causality relation exists,

number of microservices can hardly be used to forecast future trend of TD.

ID	Granger causality
$S01$, $S07$, $S10$, $S15$	Yes
$S02$, $S03$, $S09$, $S12$, $S13$	No

Table 6.3: Granger causality

In light of this analysis, enough statistical evidence has been gathered to already reject the null hypothesis H_0^2, concluding that a relation between the two metrics generally exists.

However, in the end a possible correlation between TD growth rate and number of microservices has also been investigated, since intuitively I could expect that as the number of microservices increases (and the system becomes more complex), the TD grows at higher rate. This conjecture could be discarded in light of the result of Cross-Correlation: indeed, although some lags with a strong positive correlation are present (Table 6.4), they are always matched with a nearby lag characterized by an equally strong negative correlation (Figure 6.6), suggesting that the overall correlation is definitely not enough significant to support the hypothesis.

Therefore adding or removing a microservice has a similar impact in the TD growth rate independently from the microservices already present. As a subjective interpretation I could conjecture that this effect is a consequence of an appropriate adherence to the microservice architecture principles, through which microservices are developed independently by following a loosely coupled and highly cohesive architecture.

ID	Cross-Correlation (at some lag)
$S07$, $S09$, $S10$	very strong ($>>$ confidence level)
$S01$, $S02$, $S12$	strong ($>$ confidence level)
$S03$, $S04$, $S05$, $S08$, $S13$, $S14$, $S15$	absent or very weak ($<$ *or* $\approx$ confidence level)

Table 6.4: Cross-Correlation between TD growth rate and microservices

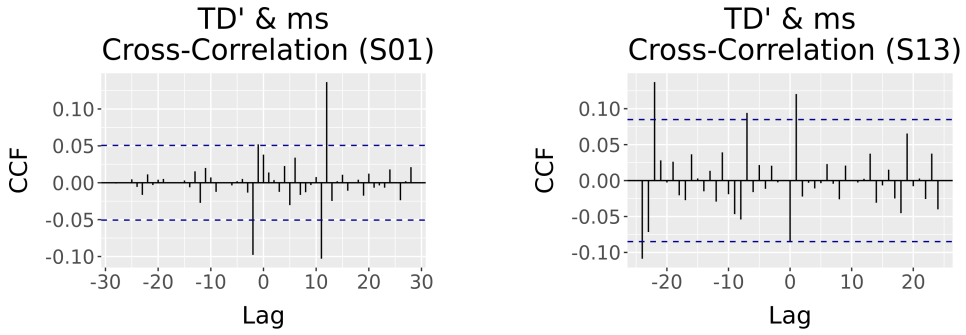

Figure 6.6: $S01$ and $S13$ TD growth rate and microservices CCF

RQ_2 answer (*Relation between TD and microservices*)

TD and microservices number are generally strongly correlated, although in some cases this relation is not present at all, with a phase shift in advance of the microservices number over the TD.

In some cases also a causality relation exists, but they seem to be quite isolated cases and not a general tendency.

Regarding the growth rate of TD, this is definitely not related (at least not significantly) to the number of microservices; so being the relation between TD and microservices number quite linear, the addition or removal of a microservice does not impact the growing rate of TD.

Chapter 7

Threats to Validity

Despite my best effort, the results of this study have to be interpreted in light of threats to validity which might have influenced them. It is about aspects that can have affected execution of experiments, collection of results or interpretation of them, and in this chapter I discuss them following the categorization of Runenson *et al.* [42].

7.1 Construct Validity

This category concerns whether the measurements adopted are appropriate to answer the RQs. For this aim I relied on two measures: the TD and the number of microservices.

Regarding the first one, it has been measured with SonarQube, that uses the SQALE index. It is a tool (and index) widely used in literature and in practice [19], but I am aware that using a different tool could have led to different measurements; anyway the difference should not have been substantial [45].

I already said that transforming an irregular-spaced time series to regularly-spaced one can change the seasonal component result of STL analysis; anyway as highlighted by Eckner [44], the only effect seems to be a smoothing. Since I am not interested on the exact seasonality, but only on its coarse shape, this aspect definitely does not influence the results (even more so than none of the system have showed seasonality).

With regard to the number of microservices, a more detailed discussion on possible defects of the detection method has been covered in Section 4.7.

Anyway the heuristic on which is based, i.e. that a microservice corresponds to a Docker container, should not have affected the results, being that a possible delay or advance of microservices time series has also been taken into account by studying the correlation between the two time series at different lags with CCF.

About CCF, being the two time series non-stationary, a Detrended Cross Correlation Analysis (DCCA) [46] could have been conducted, but consequently the data should have been differentiated to meet the stationarity requirement of Granger test for causality, so I decided to consistently use the same approach also for correlation.

7.2 Internal Validity

Here I treat possible confounding factors that could have produced the recorded results. Just to avoid potential "noise" in the results introduced by TD measurement, I: (i) discarded all commits that (requiring a compilation) failed to build and for which it was not possible to find an objective fix that would have not altered the compilation with respect to the author's intentions; (ii) manually scrutinized those commits characterized by an anomalous TD value and (iii) conducted a rigorous statistical analysis on the collected data.

7.3 External Validity

Concerning the ability to generalize the findings, I can reasonably assert that comparable results might be observed in other microservices-based software-intensive systems. It is true that a lack of MSA systems is observed, due to the fact that most of them are commercial products, so finding an open source one is uncommon, but the dataset as already showed is enough heterogeneous in terms of: language, typology, size and scope.

A smaller threat is about the choice of considering only single-repository

systems, i.e. those systems contained entirely in one repository unlike multi-repositories ones that can spread components over multiple repositories. This choice has been made not to miss intermediate evolution of the minor repositories between one commit and the next of the main aggregator repository. Anyway this aspect should be related only to the size and organization of the system and should not impact its development mode and other aspects that could influence TD.

7.4 Reliability

The results of this study, given their almost purely quantitative nature, are extremely likely replicable by other researchers, with the exception of the manual scrutiny conducted to analyze commits with anomalous TD values (that anyway are a minimal fraction). The rest of the study is completely based on mining and data analysis scripts made available in a replication package reported in Appendix C with the environment characteristic, settings and all other information.

Chapter 8

Discussion

The results just seen allow not only to answer the RQs, as I did in the previous chapter, but also to elaborate a wider overview on the TD in MSA, also thanks to the additional qualitative inspections conducted to complement statistical results. From this study, various (good) properties of MSA show up and several lessons can be learnt.

I illustrated how TD evolution shows some periods where it does not grow significantly, but I also pointed out, that they happen almost exclusively in an advanced phase of the development, when the system is likely to have a slower evolution (if not an only maintenance phase). During these phases sometimes long periods of decreasing can also be noticed, very likely induced by actions aimed to pay back the TD. While this can be seen as a virtuous practice, the real problem is localized at the early phases of development. In this phase, the development is very intense, as commit frequency proves, and probably this is the reason why developers more likely resort to sub-optimal expedients.

It is above all during this phase that developers should be aware that a variety of activities, although to different extent, can introduce TD (even refactoring if it does not aim specifically at paying back TD): evolving business logic, introducing new microservices, adding new dependencies or frameworks and also writing tests. Adopting a code quality analysis phase in the development workflow at the initial stage certainly reduces the development, but it could allow to early isolate the main root causes of TD that characterize the whole system lifecycle.

About the relation between TD and microservices number, that has showed

up in many systems but not all, it is difficult to find a pattern that identifies the characteristic of systems where this property has been found. The only findable connection regards the nature of the systems: in fact all the 3 systems where the correlation is significant and also a causation is present (namely $S01$, $S07$ and $S10$) are real systems, and not industrial demos. Anyway, in the other analyzed real systems a causality has not been found ($S02$ and $S09$) and in a case not even a correlation ($S08$). A real industrial workflow, hence, might be enough rigorous to determine a strict connection between the two metrics.

Another interesting aspect regards the non correlation between TD growth rate and the number of microservices. Indeed this proves that the addition or removal of a microservices has similar impact regardless the number of already present microservices; in other words TD evolution is in nearly linear relation with the number of microservices. This means that although the system becomes bigger and more complex the growth of TD does not rear up. Clearly such a property is very desirable (also with respect to other architectures, like the monolithic one), and the possible cause could be found in the MSA principle of independent development.

Chapter 9

Conclusion

In this study I present a multiple case study to investigate the evolution of Technical Debt in microservices architectures. The investigation consider 13 microservice-based systems, heterogeneously selected by different aspects, above all diversified by for language, size, number of microservices and nature (real cases or industrial demos).

The study is of primarily quantitative nature, based on Mining Software Repository, source code quality analysis and statistical analysis of data, although some results have been complemented with manual qualitative inspection of commits.

The results show that TD evolution is characterized by an overall growing trend, particularly accentuated in early development phases; the evolution anyway is not characterized by any periodic trends. TD variations can be caused by a variety of activities, but the more impacting ones are clearly the addition/removal of a microservice, the evolution of business logic and the refactoring. Moreover the extent of TD variations is independent of the number of microservices already present, although the overall TD evolution is strongly correlated to it.

As concluding remarks, I note that adhering to microservices architecture principles might keep TD compartmentalized within microservices, allowing it to not increase superlinearly, and so making it more manageable with respect to other types of architectures (e.g. monolithic ones). It is crucial for developers to remain aware of the potential TD they may incur in, regardless the activities they undertake, because even a trivial change, like the upgrading of

a dependencies, could have a significant impact on the system's TD. So if it is feasible to maintain a consistent level of TD during the evolution of the system, an increase of TD may be inevitable as the system grows in size and complexity.

9.1 Future Work

This study opens up the possibilities for a series of in-depth analysis, in fact there are several facets which could provide us more information on the phenomenon under investigation that have not yet been considered. As future work, this study can be complemented by considering other factors:

- the individual contribution of each microservice to the TD at system level can be measured, by extending the microservice detection method to also find out the paths;
- a more in-depth and systematic analysis of TD hotspots can be conducted to understand which types of activities expose more developers to an increase of the TD;
- the individual contribution of each developer can be analyzed, by measuring the TD they introduce or pay back during the implementation of microservices, to understand if the TD is homogeneously distributed or not;
- other types of TD other than Code TD, can be measured, e.g. the ATD with the ATDx tool [47];
- interviews with the developers of considered systems can be conducted to gain further insights on trends and TD hotspots (e.g. which are the reasons that have led to introduce TD).

Appendix A

Dataset

This appendix showcases the 15 systems composing the dataset analyzed in the study and already listed in Table 5.1. In Table A.1 a description of the systems, while in Table A.2 some metadata that give an approximate overview on their size and diffusion.

ID	Description
$S01$	Integrated hosting platforms for Single Pages Applications
$S02$	School administration software
$S03$	Spring Boot demo application (by Alibaba employees)
$S04$	Demo insurance sales system (by Altkom Software)
$S05$	Travel agency demo web application (by IBM)
$S06$	Demo e-shop (by Microsoft)
$S07$	Open source server for sharing spatial data
$S08$	Starter kit for Software-as-a-Service systems
$S09$	Application for collecting data from multiple sources
$S10$	Distributed Machine Learning integration tool
$S11$	Open Source Real estate management system
$S12$	Demo e-shop (by Minos Framework)
$S13$	Demo e-shop (by NashTech)
$S14$	Demo real time quiz application (by Barcelona .NET Group)
$S15$	Demo e-shop (by OpenTelemetry)

Table A.1: Brief description of the systems composing the dataset

ID	Stars	Forks	Commits	Contributors	SLOC	AVG # ms	From hash	To hash	From date	To date
$S01$	44	29	1502	21	417k	$\approx 6,5$	`a1c9466`	`aa761bf`	01/04/2020	09/08/2023
$S02$	79	36	275	9	5k	≈ 3	`e556352`	`46b19b5`	03/30/2019	04/25/2023
$S03$	109	80	279	4	7k	$\approx 8,5$	`1ba3e35`	`654cf9b`	01/11/2017	08/14/2020
$S04$	481	171	384	9	39k	≈ 7	`5169e13`	`9871a2e`	07/25/2018	04/19/2021
$S05$	30	12	375	7	255k	≈ 4	`7cde267`	`1228252`	09/04/2019	12/09/2021
$S06$	24572	10514	4336	168	137k	$\approx 11,5$	`3cbaf41`	`b9aae67`	09/06/2016	08/29/2023
$S07$	193	61	1027	11	67k	≈ 8	`9f6414d`	`f8f3d90`	07/09/2020	09/06/2023
$S08$	103	18	629	2	116k	$\approx 2,5$	`e5a1488`	`9194ce4`	08/23/2021	09/07/2023
$S09$	35	24	1428	16	219k	≈ 6	`d4a4459`	`843943f`	03/26/2019	05/17/2022
$S10$	75	22	1214	14	7k	$\approx 5,5$	`b11a11a`	`084699b`	03/19/2020	05/11/2022
$S11$	385	147	412	3	32k	$\approx 4,5$	`47efe58`	`c5cc2ab`	11/05/2017	06/17/2023
$S12$	11	1	1069	4	44k	$\approx 8,5$	`f4307ad`	`9e3cdcc`	06/02/2021	02/01/2022
$S13$	257	103	535	28	82k	≈ 8	`c88ec52`	`4177b13`	01/18/2022	09/08/2023
$S14$	120	41	658	3	4k	$\approx 3,5$	`b70c960`	`82fb546`	02/22/2017	11/20/2018
$S15$	1132	580	572	88	55k	≈ 15	`75bf84f`	`54aefe5`	04/26/2022	09/07/2023

The commits taken into account are those from the very first to the last of 08/09/2023 (at the date of 11/09/2023).
The preliminary SLOC metric has been measured with cloc[a]

Table A.2: Metadata of the systems composing the considered dataset

[a] github.com/AlDanial/cloc

Appendix B

Outcomes

Here all the results of tests and investigations conducted are fully reported for completeness.

B.1 Mann-Kendall trend test

Results of the Mann-Kendall test, with resulting τ and $p–value$. A τ value near to 1 indicates a strongly increasing trend, while near to -1 a strongly decreasing trend.

ID	τ	$p–value$
$S01$	0,83	$\leq 2,22e{-}16$
$S02$	0,82	$\leq 2,22e{-}16$
$S03$	0,82	$\leq 2,22e{-}16$
$S04$	0,84	$\leq 2,22e{-}16$
$S05$	0,49	$\leq 2,22e{-}16$
$S07$	0,90	$\leq 2,22e{-}16$
$S08$	0,97	$\leq 2,22e{-}16$
$S09$	0,59	$\leq 2,22e{-}16$
$S10$	0,79	$\leq 2,22e{-}16$
$S12$	0,59	$\leq 2,22e{-}16$
$S13$	0,84	$\leq 2,22e{-}16$
$S14$	-0,58	$\leq 2,22e{-}16$
$S15$	0,23	$\leq 2,22e{-}16$

Table B.1: Mann-Kendall trend test results

B.2 TD trend

Plots of TD trend obtained with LOESS regression (in lighter the real measured evolution).

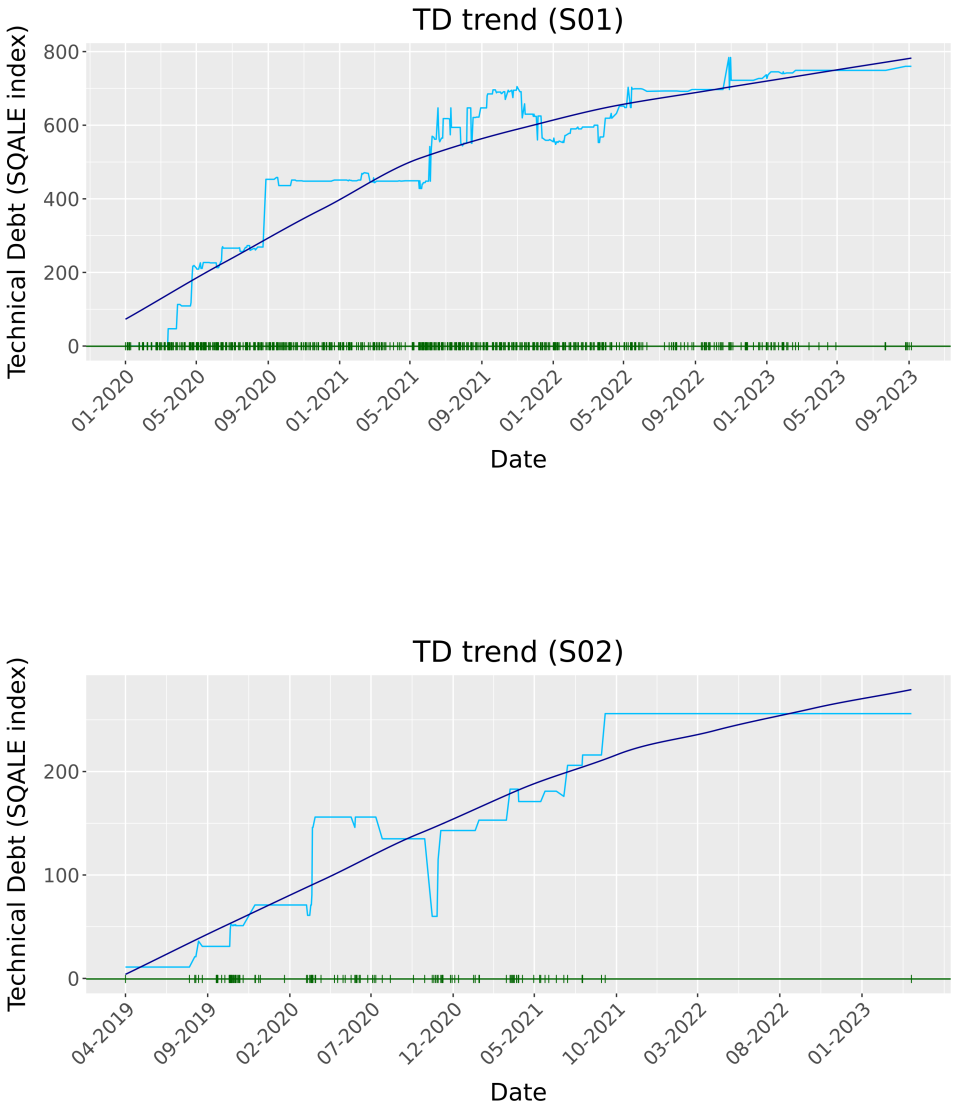

Figure B.1: $S01$ and $S02$ trend plot

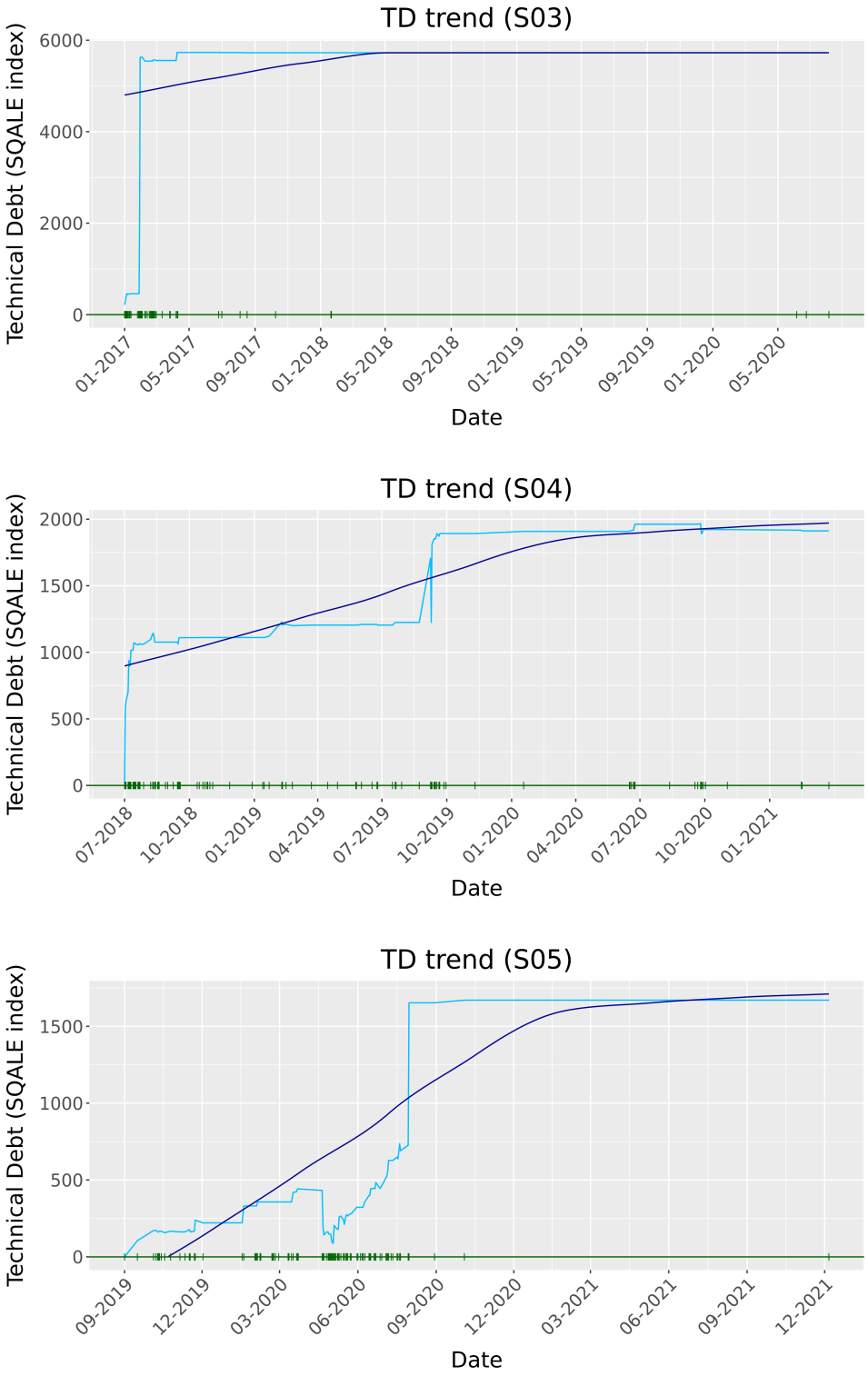

Figure B.2: *S*03, *S*04 and *S*05 trend plot

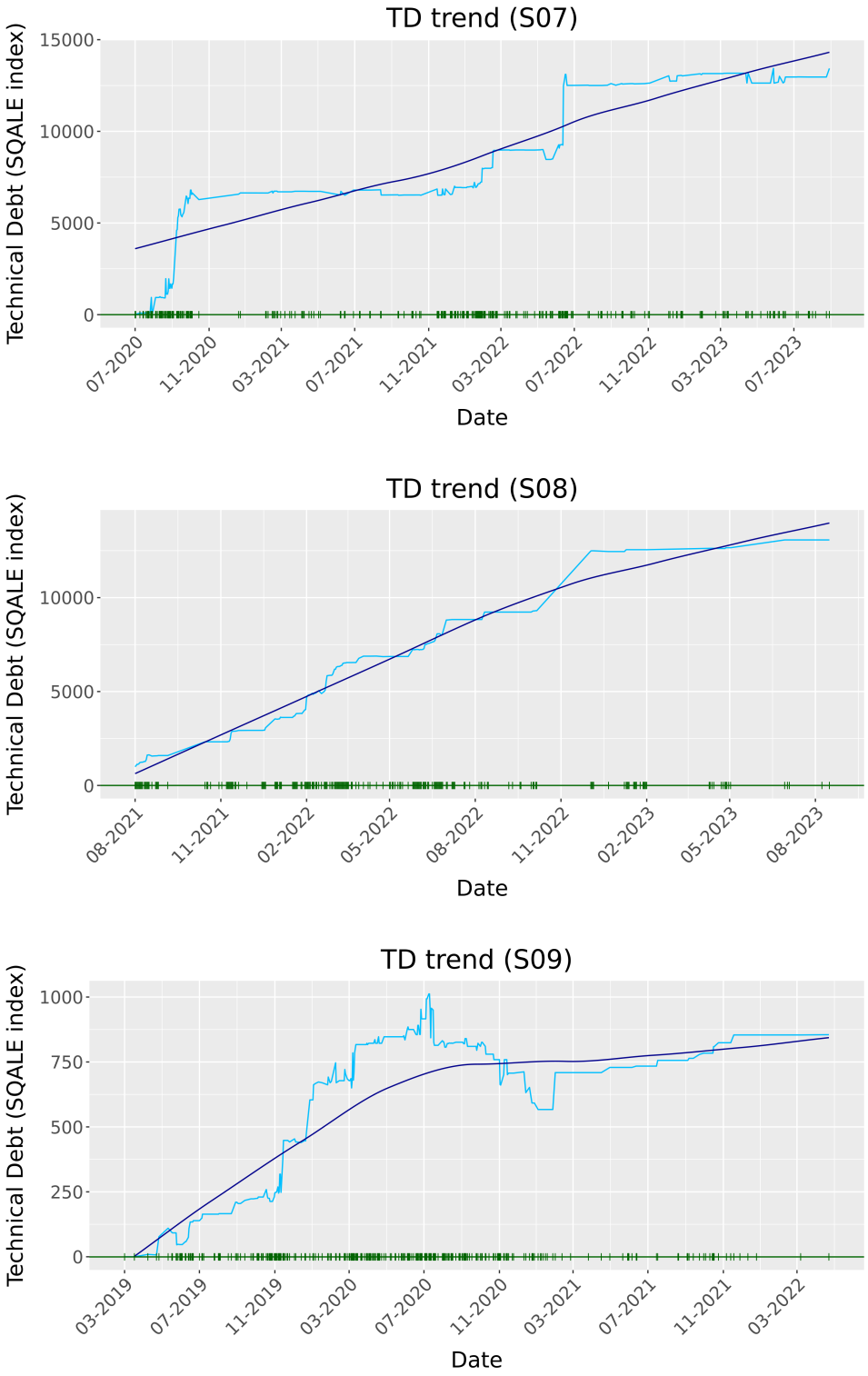

Figure B.3: *S*07, *S*08 and *S*09 trend plot

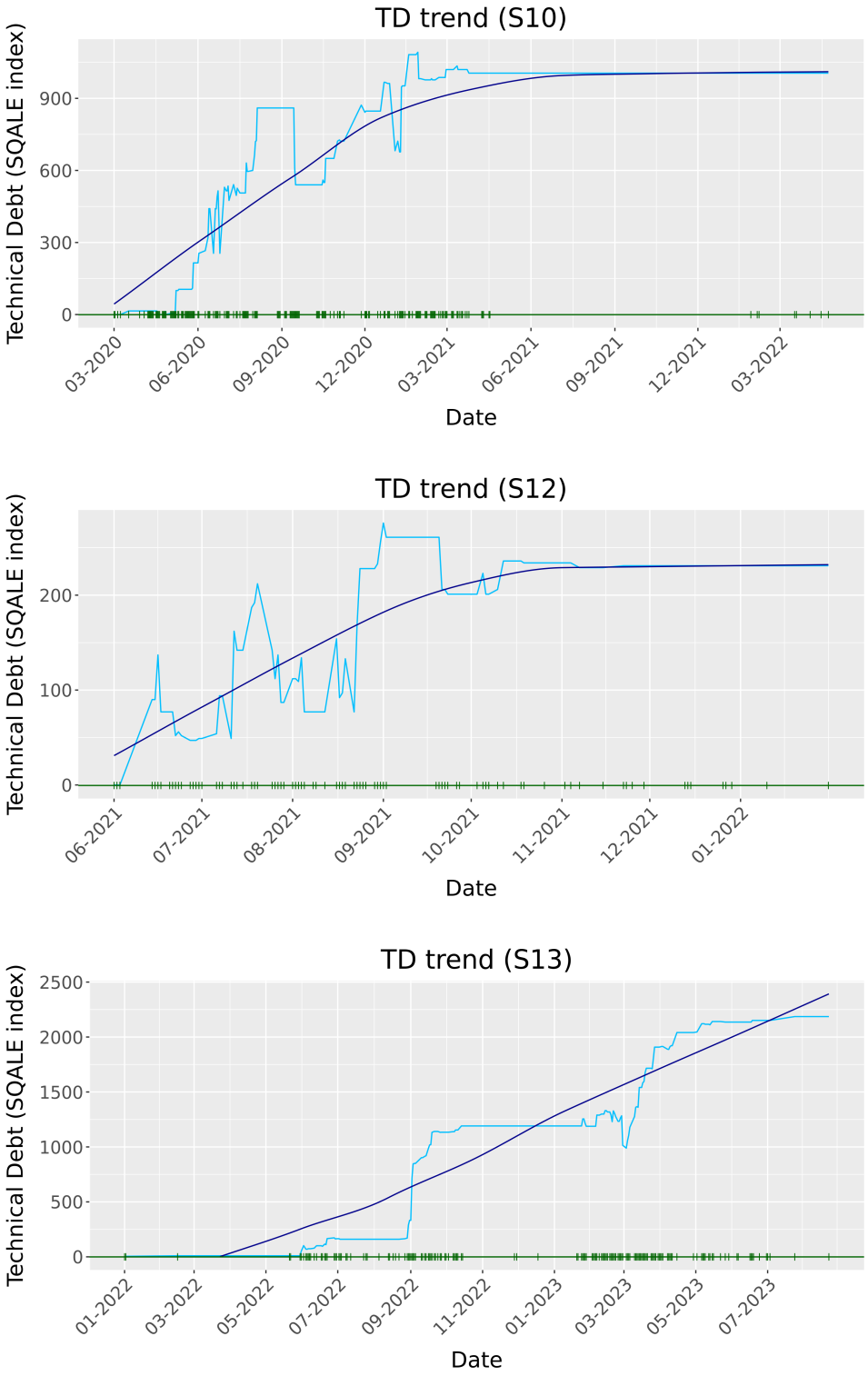

Figure B.4: $S10$, $S12$ and $S13$ trend plot

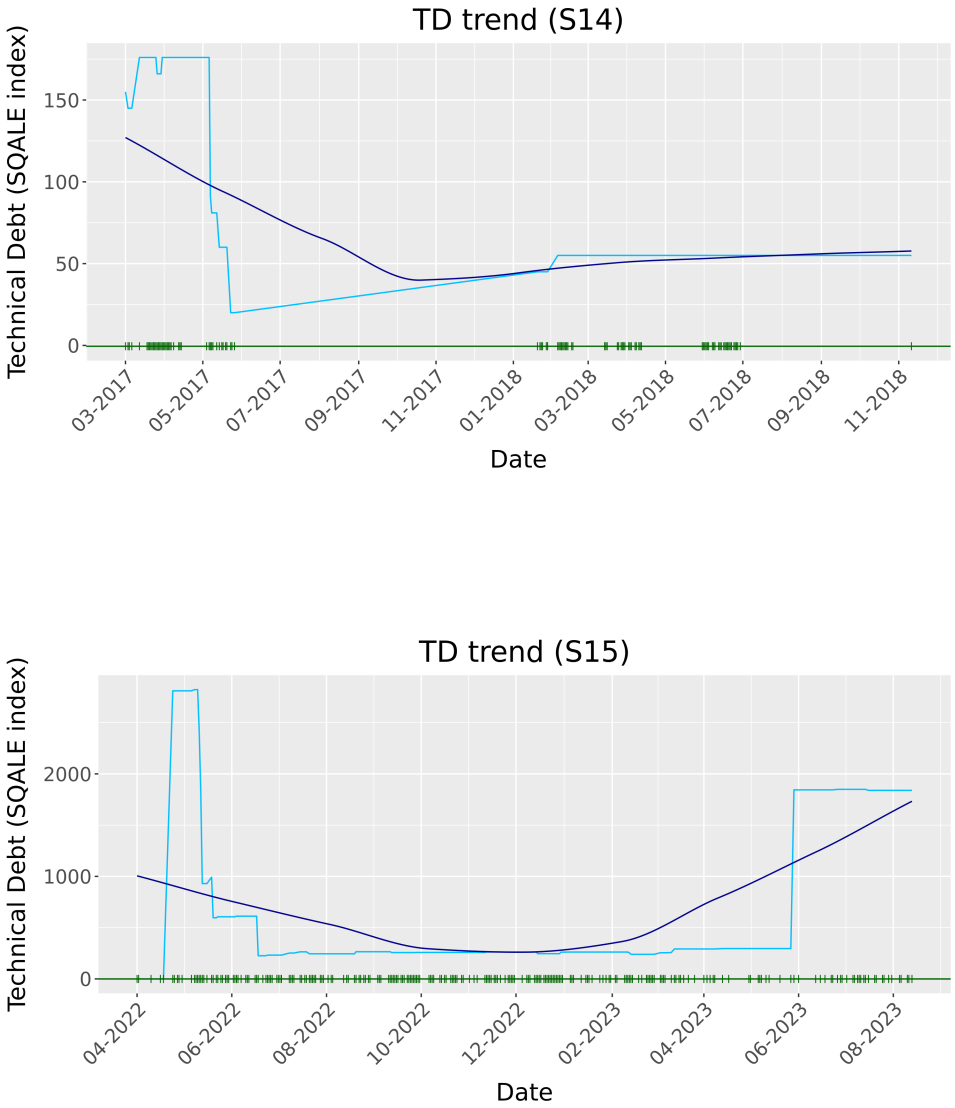

Figure B.5: $S14$ and $S15$ trend plot

B.3 Hotspot inspection

Results from manual inspection of top 10 TD hotspots, reporting commit statistics and main activity performed.

Commit	ΔTD	Files	Adds	Dels	Activity
`477618d`	+139	23	16124	1	added SPA
`4241f46`	+71	17	1752	1089	evolution
`bc3d70e`	+70	42	15834	0	added package
`3e98824`	+67	59	18016	0	added SPA
`d15e29a`	+62	27	11855	958	evolution
`eca89a9`	−62	10	135	272	fixing
`fd683c5`	−60	21	2830	9304	test
`fe3ec08`	+55	53	15546	0	added SPA
`5ea9a61`	−55	109	17853	23134	changed SPA framework
`c7677c9`	+51	60	1702	435	added SPA

Table B.2: *S*01 hotspots insight

Commit	ΔTD	Files	Adds	Dels	Activity
`d5835ca`	−75	36	0	1049	removed microservice
`41ff3f6`	+65	30	898	0	added microservice
`dc0e813`	+55	12	262	4	added gateway
`ffa43de`	+30	41	401	291	evolution
`1ca5244`	+30	60	809	1257	evolution
`9740132`	+26	31	534	8	evolution UI
`0343337`	−21	36	36	39245	substituted local libraries with package manager
`a10ad82`	+20	9	50	45	evolution microservice
`8b8db0e`	+20	4	68	6	refactoring
`a907392`	+20	6	169	0	added Docker to microservices
`3446715`	+20	3	25	13	error fixing

Table B.3: *S*02 hotspots insight

Commit	ΔTD	Files	Adds	Dels	Activity
`8d72c71`	+3510	13	257	0	evolution
`f596edc`	+3510	13	257	0	evolution
`24a4ac9`	+1698	1	1	0	added module
`1ba3e35`	+216	46	1258	0	initial commit
`89db00d`	+180	51	751	295	refactoring
`1d26165`	−102	84	300	349	refactoring
`b366099`	+44	8	89	9	added microservice
`6496d0a`	−40	6	2	97	refactoring
`c2bf313`	+40	4	95	0	refactoring
`b206af8`	+39	28	125	109	miscellaneous

Table B.4: *S*03 hotspots insight

Commit	ΔTD	Files	Adds	Dels	Activity
`0ab3b02`	+547	80	2489	10	added microservice
`d895297`	+482	44	2199	2	added microservice
`b4e0c2e`	+120	27	384	335	added microservice
`d2c5226`	−105	14	13	138	evolution microservice
`e7e3ba8`	+92	20	1050	0	added microservice
`349b737`	+85	17	381	0	evolution microservice
`9cd7c1d`	+75	39	423	22	initial UI in Angular
`2d4b650`	−75	8	29	24	deprecations fixing
`c60e021`	+72	52	2890	751	initial UI in Vue.js
`c5f1e18`	+70	13	230	7	evolution microservice

Table B.5: *S*04 hotspots insight

Commit	ΔTD	Files	Adds	Dels	Activity
`9a4a93f`	+992	36	309	261	momentarily Jaeger disabling
`699ee32`	−357	22	249	220	error fixing + prettification
`b6e54a1`	+116	102	153543	109	version 2
`918e970`	+115	57	2530	383	UI
`d72f706`	+110	15	81917	3	added microservice
`72b380b`	+107	132	1502	623	miscellaneous
`c9efad1`	+105	9	1372	0	initial commit
`ed61356`	+104	26	631	119	added microservice
`8660e53`	+77	20	8859	0	added microservice
`0b9c379`	+70	2	66	26	evolution microservice

Table B.6: *S*05 hotspots insight

Commit	ΔTD	Files	Adds	Dels	Activity
a0004ec	+3231	93	1670	1165	evolution
19db639	+2557	35	3364	1	evolution microservice
2bcfef9	+885	1	1	1	update Java from 11 to 17
f7ee79f	+723	16	348	5	evolution UI
86fd33f	+702	46	829	537	evolution microservice
174b87d	+682	156	3867	4203	refactoring dependencies
5741d71	−680	2	19	250	test
285b40a	−538	4	37	10	Maven profile
b047f7c	−537	615	317	228	project structure refactoring
e2d5b76	+500	1	2096	0	evolution microservice

Table B.7: *S*07 hotspots insight

Commit	ΔTD	Files	Adds	Dels	Activity
ba2d690	+3988	29	15627	162	added microservice + evolution
669adfe	+1025	101	17848	0	initial code
6c4f742	−797	20	343	616	deleted microservice
704257c	+731	15	2983	148	evolution
f6e431d	+643	64	2765	667	evolution
82cf969	+629	20	7151	28	added microservices
9fb9120	+549	36	5198	1	added microservices
2aec0c2	+467	12	5093	90	added microservice + evolution
c286326	+414	89	4598	171	added package + evolution
d9b3122	+397	78	6169	63	added microservice + evolution

Table B.8: *S*08 hotspots insight

Commit	ΔTD	Files	Adds	Dels	Activity
00cee1c	+148	1	68	15	test
c2aa19d	+142	41	689	368	refactoring microservice
ad48c09	+105	6	14	5	refactoring microservice
b109171	−96	7	71	86	refactoring microservice + test
cc583c5	−87	9	160	152	error fixing
e1438c0	−80	17	44	23	warning fixing
9b810a8	+78	2	58	4	test
17cb8ec	+72	4	40	18	test
4384431	−72	2	0	12	refactoring
4a8edbd	+70	24	626	0	initial structure microservice

Table B.9: *S*09 hotspots insight

Commit	ΔTD	Files	Adds	Dels	Activity
`50d70c8`	−250	7	52	256	refactoring
`2be757c`	+150	8	386	7	initial configuration, code, files
`37a21af`	+130	18	849	115	added microservice + evolution
`a4f3b14`	+125	6	267	0	added microservice
`d5cf670`	+120	1	1	1	refactoring
`47b3cbc`	+120	6	406	1	added microservice
`7f8bbea`	+120	14	794	73	added microservice
`09c49db`	−110	21	695	771	refactoring microservices
`71addd6`	+100	12	127	27	evolution
`5d9e0d1`	+80	2	26	24	refactoring

Table B.10: $S10$ hotspots insight

Commit	ΔTD	Files	Adds	Dels	Activity
`080ba4d`	−75	13	78	167	configuration
`9f429e6`	−75	13	78	167	configuration
`c572806`	+60	13	693	0	initial gateway
`f51681f`	−60	6	89	23	test + dependencies
`54998de`	+60	13	171	31	refactoring
`5786052`	+60	14	170	14	splitted microservice
`592984a`	+50	12	59	135	docker configuration
`fab4fbe`	+47	11	437	32	refactoring + test
`2b32e29`	+40	5	87	11	added microservice
`bbeba43`	−40	124	1682	645	refactoring structure

Table B.11: $S12$ hotspots insight

Commit	ΔTD	Files	Adds	Dels	Activity
`b40a71c`	−304	39	718	1113	refactoring
`fba9452`	+149	11	507	23	evolution
`400822b`	+120	46	1432	45	added microservice + fixing
`9e29850`	+110	41	1575	277	added microservice
`2f14180`	+106	3	174	1	test
`3866e7b`	+98	18	182	27	evolution
`974e58a`	+98	20	613	13	evolution microservice
`be130c7`	+94	5	324	0	added microservice + test
`0bf2eb2`	+92	38	1341	19	added microservice
`55b748a`	+88	6	357	2	evolution microservice + test

Table B.12: $S13$ hotspots insight

Commit	ΔTD	Files	Adds	Dels	Activity
`a0f2499`	−75	29	49	568	substituted component with library
`c4f4bed`	−40	14	21	74	upgrade library
`8645526`	+31	11	363	0	added microservice
`9a39ba5`	−21	9	6	194	updated package
`56861be`	−10	11	118	95	refactoring
`20f5743`	−10	15	39	115	refactoring
`76355dd`	+10	9	64	27	refactoring
`2fddec3`	−10	1	1	2	updated package
`89b0c84`	−10	36	87	189	refactoring
`540d41a`	−10	4	88	27	evolution + test
`57b5da8`	+10	76	445	145	refactoring

Table B.13: *S*14 hotspots insight

Commit	ΔTD	Files	Adds	Dels	Activity
`b28cb38`	+2810	201	3654	7	initial code
`c623a8f`	+1548	27	15601	1249	evolution + dependencies
`bc0cd67`	−585	21	197	3041	migrated microservice to other language
`02cd941`	−515	10	67	2085	refactoring microservice
`c8f1829`	−396	10	191	3690	refactoring microservice
`a6b6abb`	−396	13	54	3663	refactoring microservice
`a1d1d73`	−396	9	35	3653	refactoring microservice
`e554195`	−386	23	1894	4259	migrated microservice to other language
`7c5cee8`	+72	14	156	55	evolution
`deaf1f6`	+35	236	502	2727	license comments + fixing

Table B.14: *S*15 hotspots insight

B.4 Ollech&Webel seasonality test

All the tested systems got a negative response in the seasonality test (only system with a > 24 months history subjected to test).

ID	Response
$S01$, $S02$, $S03$, $S04$, $S05$, $S07$, $S08$, $S09$, $S10$	No seasonality

Table B.15: Ollech&Webel seasonality test

B.5 TD and microservices evolution

Plots with evolution of TD, microservices number and commit frequency.

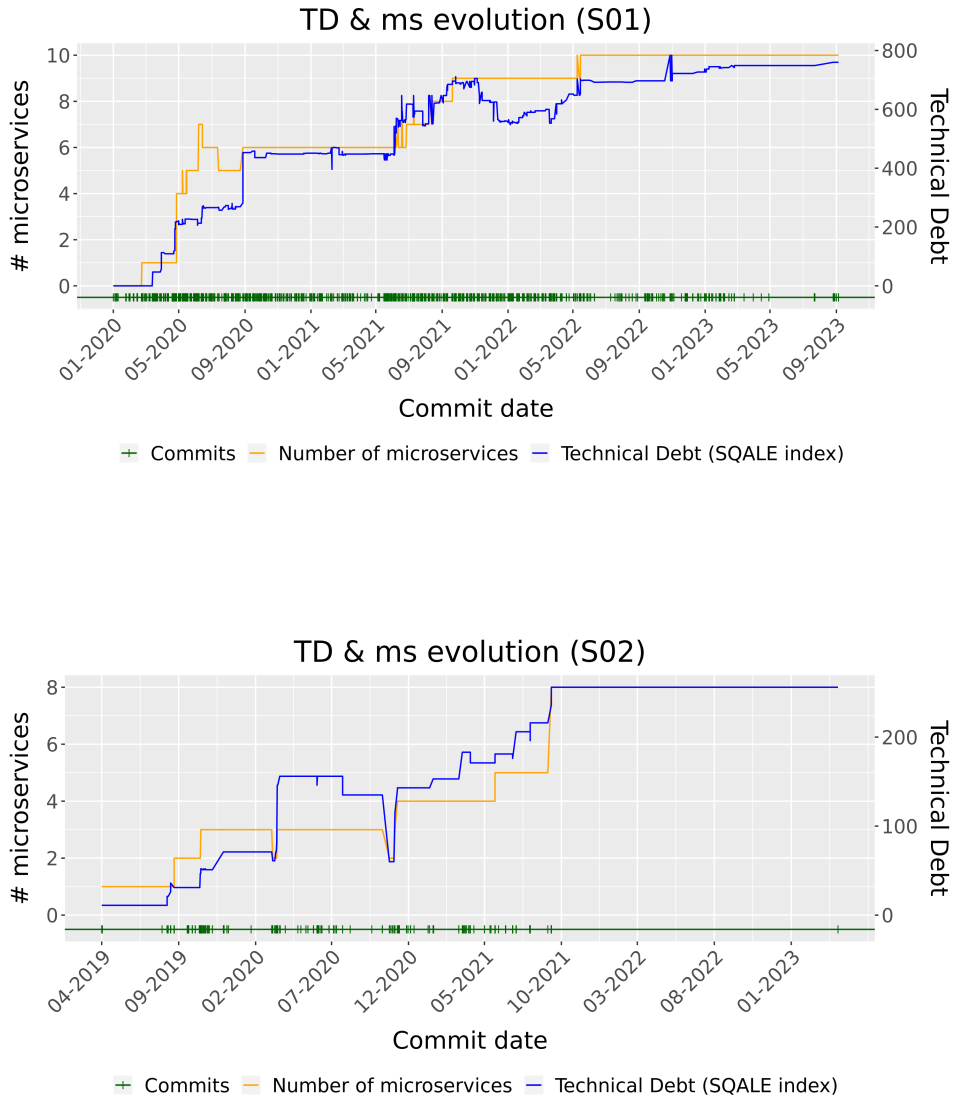

Figure B.6: $S01$ and $S02$ evolution plot

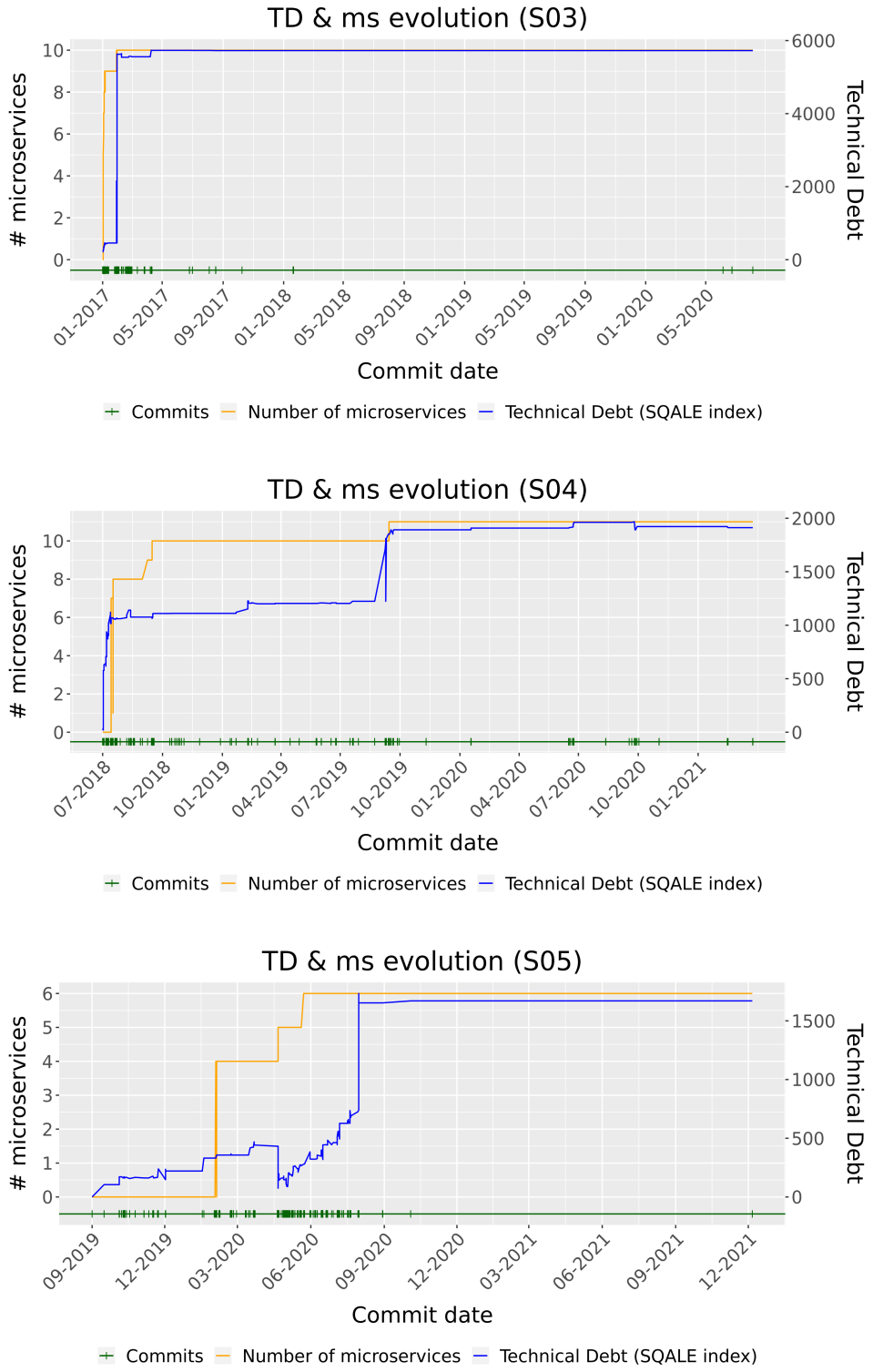

Figure B.7: *S*03, *S*04 and *S*05 evolution plot

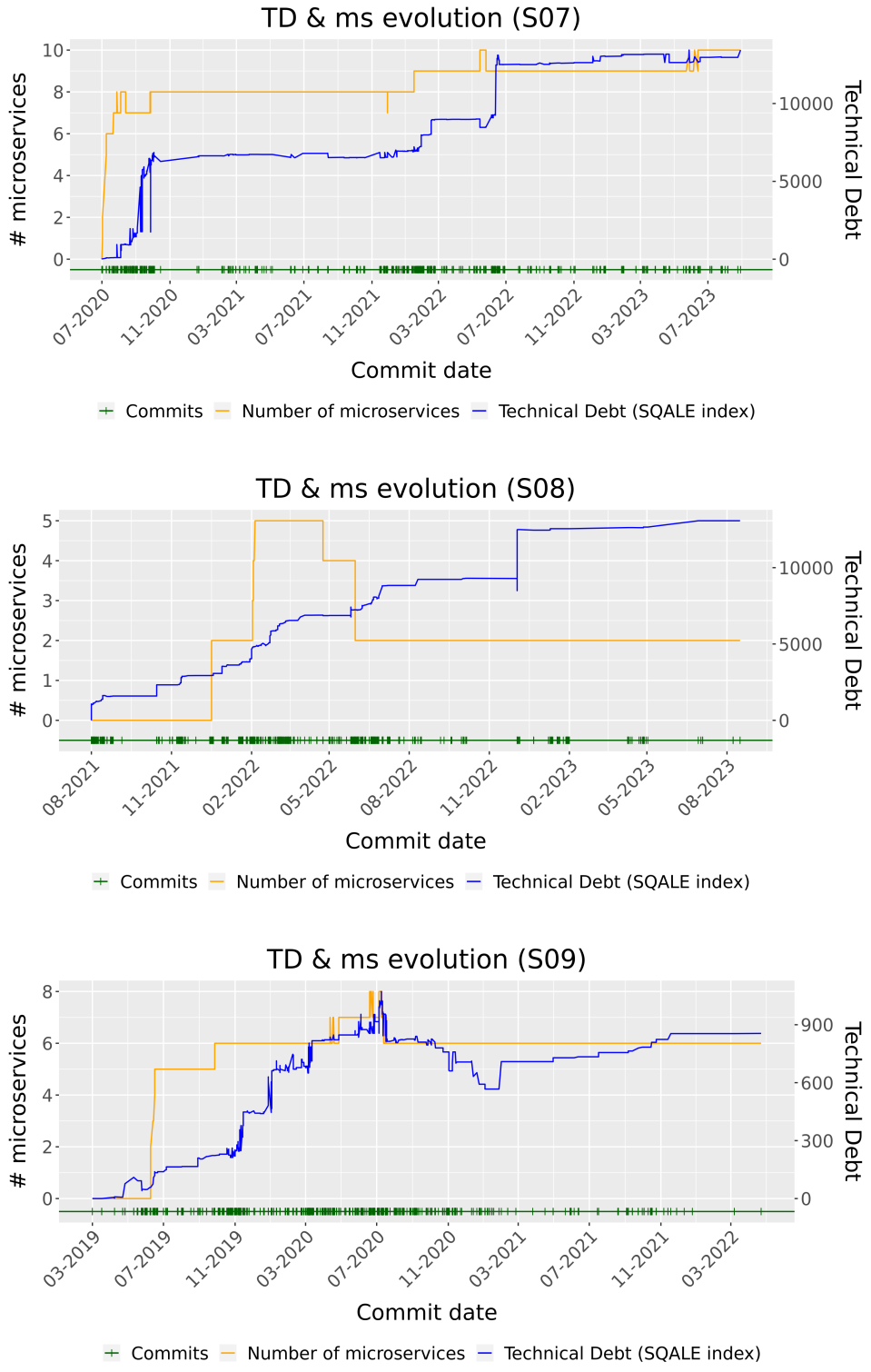

Figure B.8: *S*07, *S*08 and *S*09 evolution plot

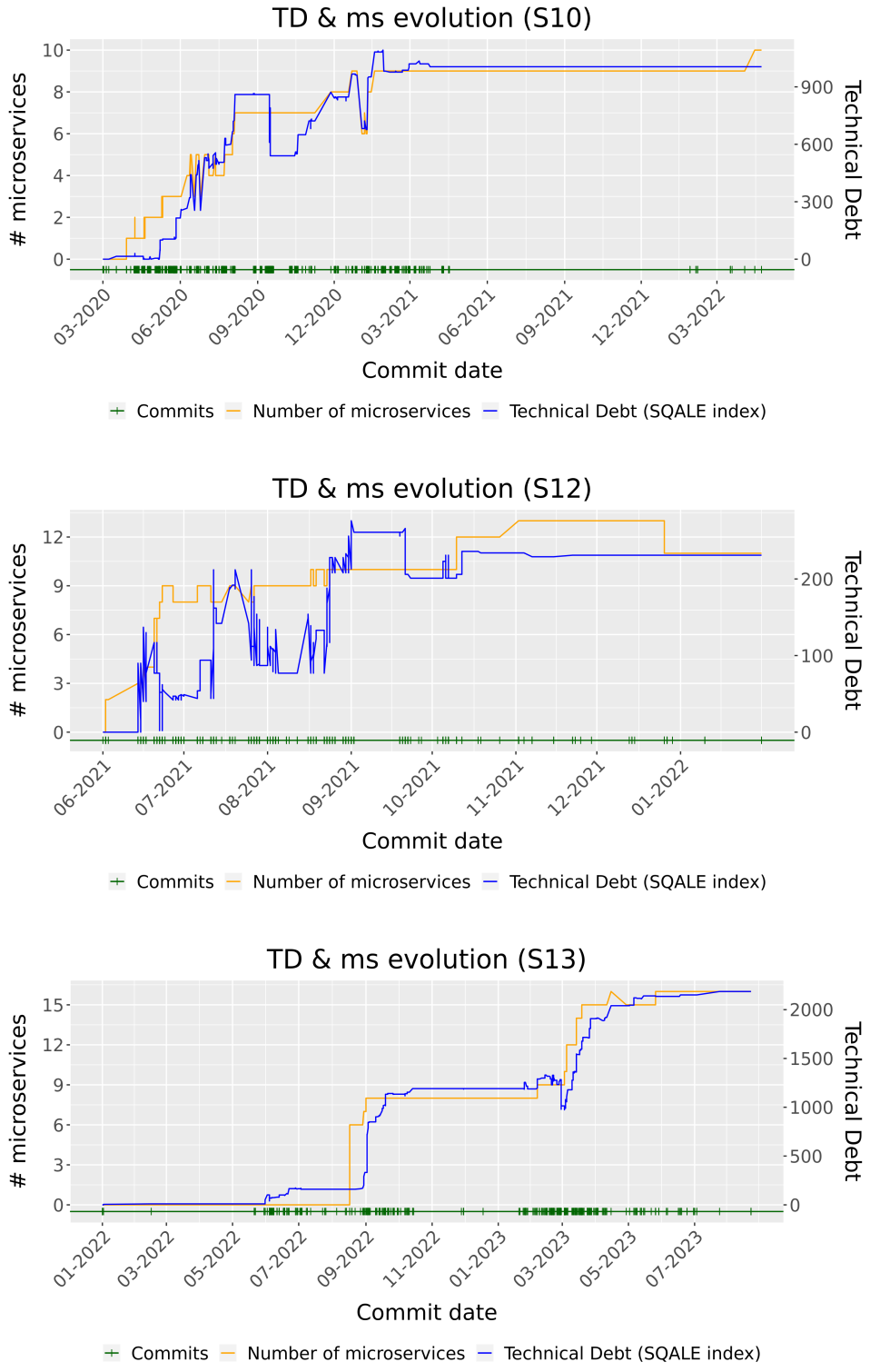

Figure B.9: $S10$, $S12$ and $S13$ evolution plot

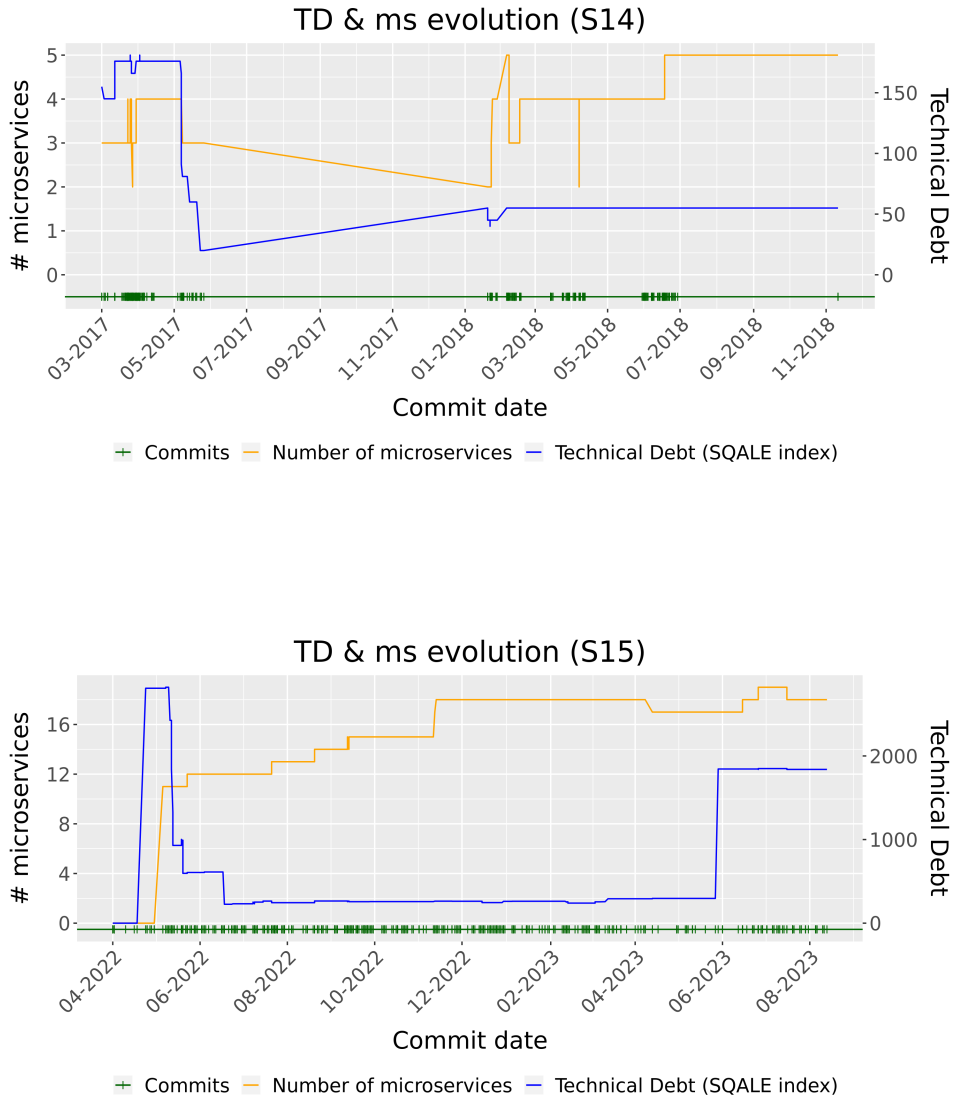

Figure B.10: $S14$ and $S15$ evolution plot

B.6 TD and microservices correlation

Cross-Correlation Function results between TD evolution and number of microservices evolution.

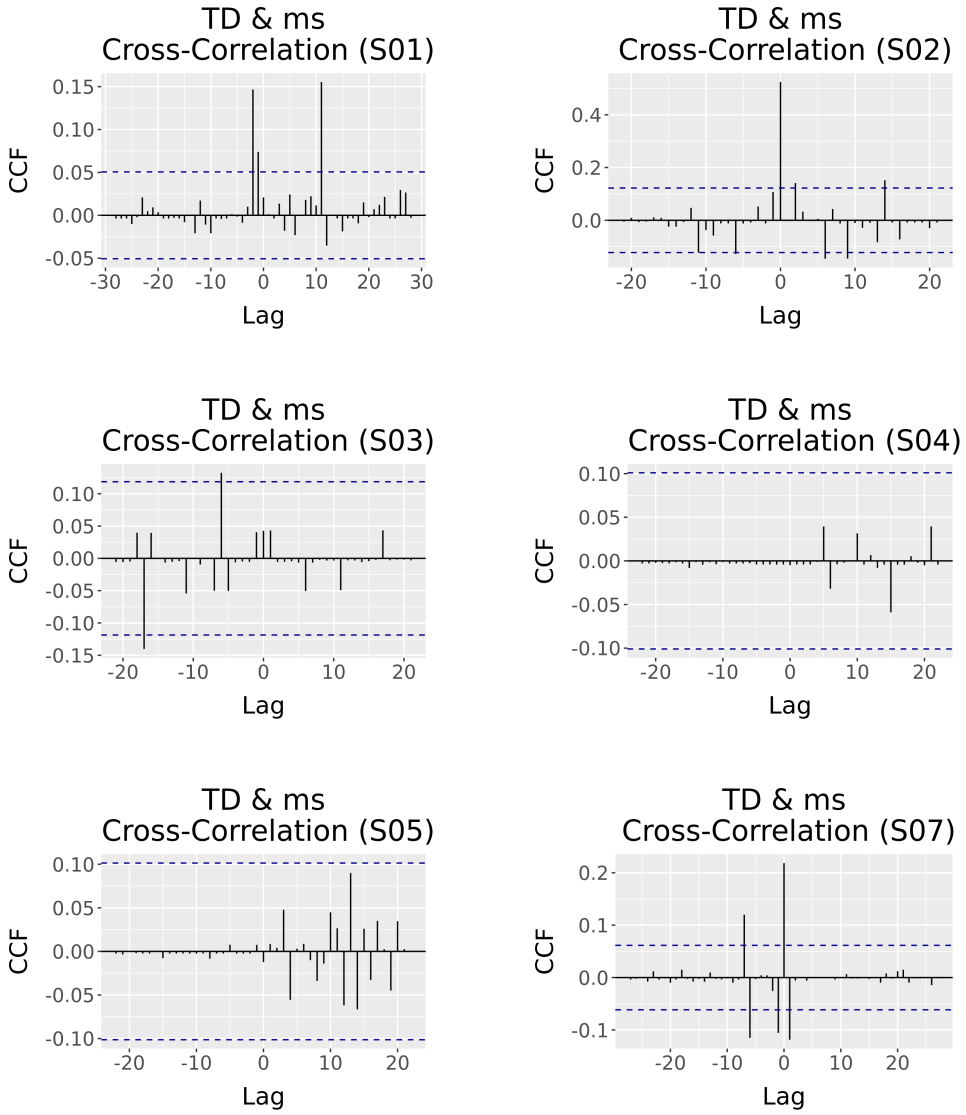

Figure B.11: *S*01–*S*07 CCF between TD and number of ms

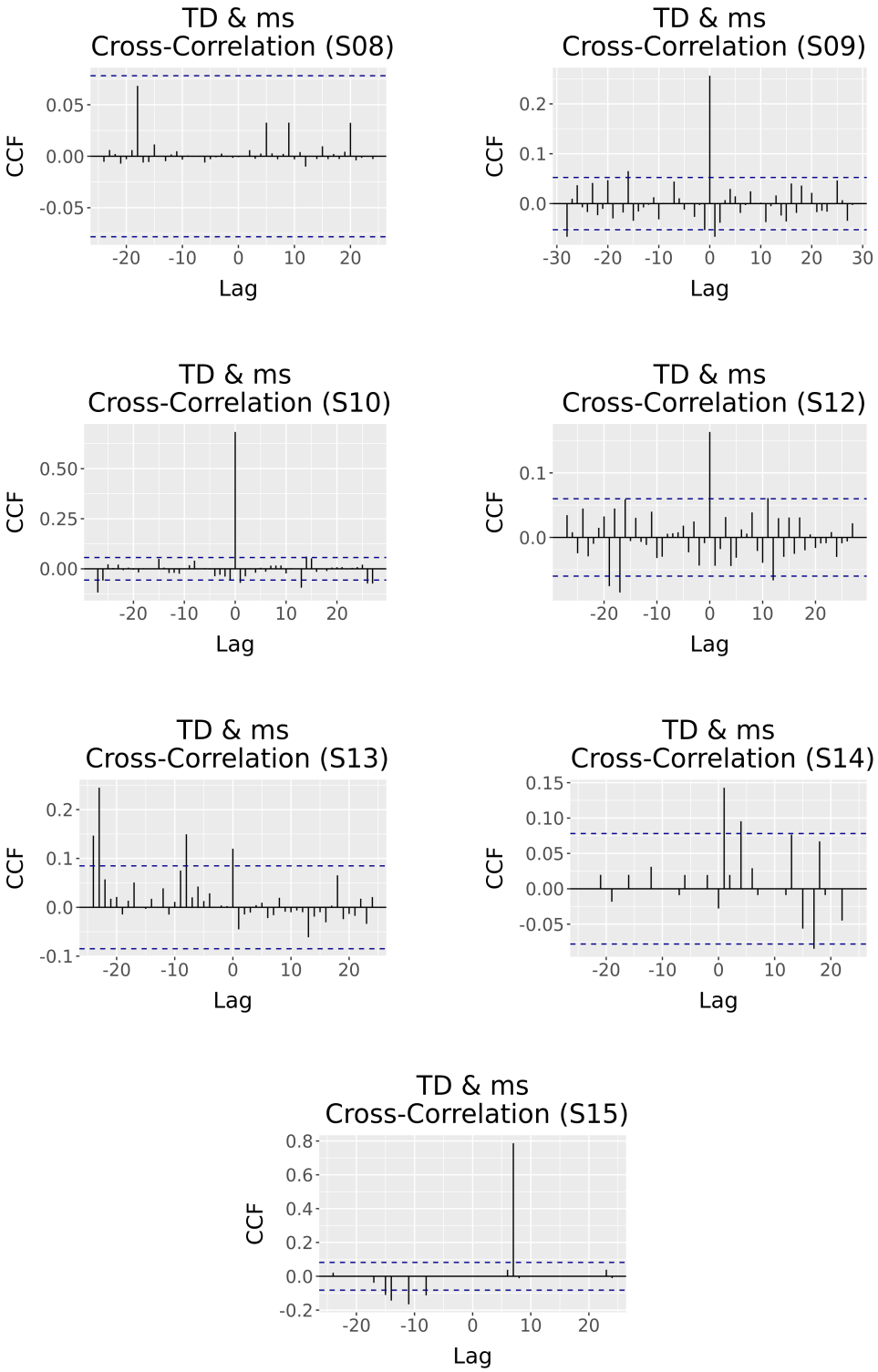

Figure B.12: *S*08–*S*15 CCF between TD and number of ms

B.7 Granger causality test

Results of Granger causality test in systems that present a correlation at negative lags between TD and microservices number.

ID	Causality	$p–value$
$S01$	Yes	$1,55e{-}9$
$S02$	No	0,22
$S03$	No	0,55
$S07$	Yes	$8,00e{-}05$
$S09$	No	0,88
$S10$	Yes	0,01
$S12$	No	0,40
$S13$	No	0,91
$S15$	Yes	$1,08e{-}17$

Table B.16: Granger causality test results

B.8 TD growth rate and microservices correlation

Cross-Correlation Function results between TD growth rate and number of microservices evolution.

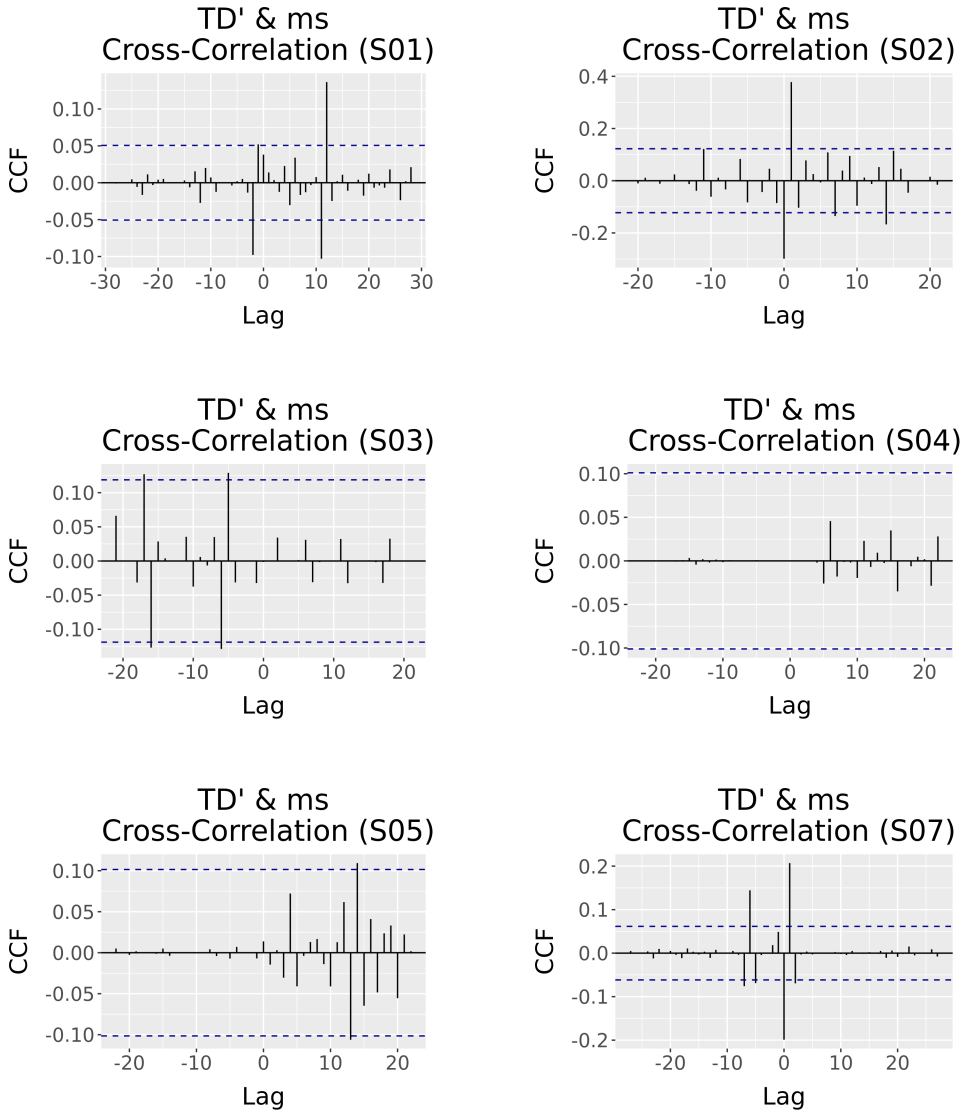

Figure B.13: $S01$–$S07$ CCF between TD growth rate and number of ms

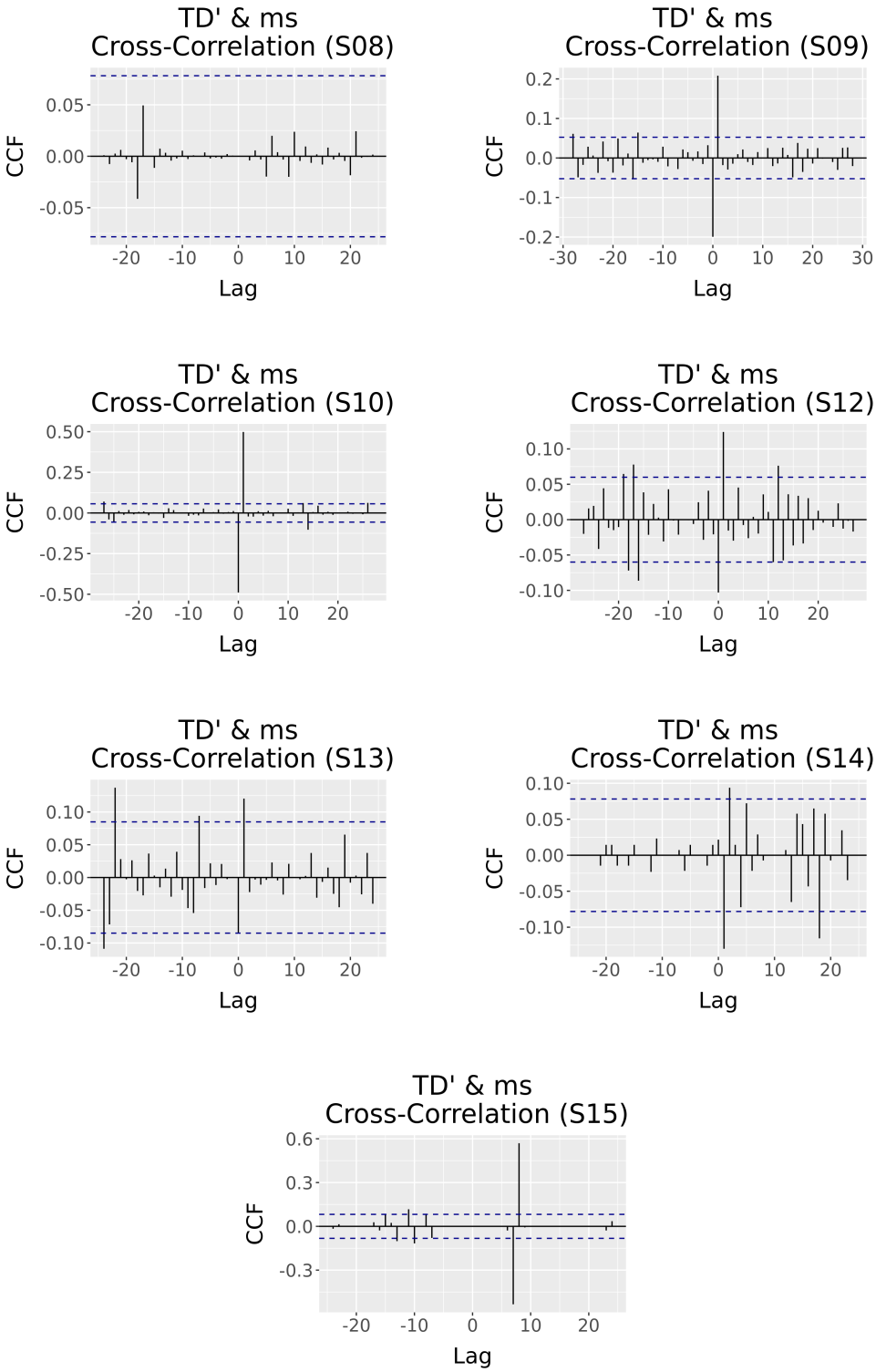

Figure B.14: *S*08–*S*15 CCF between TD growth rate and number of ms

Appendix C

Replicability

A replication package with scripts, configuration and data (both raw data and final results) is publicly available at:

`github.com/KevinMaggi/evolution-of-code-td-in-msa_rep-pkg`.

Brief instructions about how to replicate the steps taken in the study are described in the `README`.

Following the technical specification under which the experiments have been executed:

- Python v3.10.6
- SonarScanner CLI v4.8.0.2856
- SonarScanner for Maven v3.9.1
- SonarScanner for .NET v5.13.1
- Node.js v18.16.1
- Maven v3.6.3
- Gradle v8.3
- .NET v7.0.113
- OpenJDK v17.0.8.1
- SonarQube™ v9.9 LTS
- R v4.3.1
- Docker v24.0.6
- Docker Compose v2.21.0

List of dependencies (with version) of Python scripts instead is available in the `src/requirements.txt` file.

Bibliography

References

[1] R. Verdecchia, K. Maggi, L. Scommegna, and E. Vicario, "Technical Debt Evolution in Microservice Architectures: A Preliminary Case Study," in *Proceedings of the 1st International Workshop on Quality in Software Architecture (QUALIFIER)*, 2023.

[2] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)," *Dagstuhl Reports*, vol. 6, 2016.

[3] N. S. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola, "Towards an Ontology of Terms on Technical Debt," in *2014 Sixth International Workshop on Managing Technical Debt*, 2014.

[4] N. Rios, T. Mendes, M. Mendonça, R. Spínola, F. Shull, and C. Seaman, "Identification and Management of Technical Debt: A Systematic Mapping Study," *Information and Software Technology*, vol. 70, 2015.

[5] Z. Li, P. Avgeriou, and P. Liang, "A Systematic Mapping Study on Technical Debt and Its Management," *Journal of Systems and Software*, vol. 101, 2015.

[6] P. Francesco, P. Lago, and I. Malavolta, "Architecting with Microservices: a Systematic Mapping Study," *Journal of Systems and Software*, vol. 150, 2019.

[7] J. Soldani, D. Tamburri, and W.-J. Heuvel, "The Pains and Gains of Microservices: A Systematic Grey Literature Review," *Journal of Systems and Software*, vol. 146, 2018.

[8] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, "Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality," in *Proceedings of the 2019 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2019.

[9] W. Assunção, J. Krüger, S. Mosser, and S. Selaoui, "How do microservices evolve? An empirical analysis of changes in open-source microservice repositories," *Journal of Systems and Software*, vol. 204, 2023.

[10] V. Lenarduzzi, F. Lomio, N. Saarimäki, and D. Taibi, "Does Migrating a Monolithic System to Microservices Decrease the Technical Debt?," *Journal of Systems and Software*, vol. 169, 2020.

[11] V. Lenarduzzi and D. Taibi, "Microservices, Continuous Architecture, and Technical Debt Interest: An Empirical Study," *arXiv e-prints*, 2018.

[12] S. Toledo, A. Martini, and D. Sjøberg, "Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study," *Journal of Systems and Software*, vol. 177, 2021.

[13] S. Toledo, A. Martini, A. Przybyszewska, and D. Sjøberg, "Architectural Technical Debt in Microservices: A Case Study in a Large Company," in *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 2019.

[14] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, "Assuring the Evolvability of Microservices: Insights into Industry Practices and Challenges," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019.

[15] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, "Limiting Technical Debt with Maintainability Assurance - An Industry Survey on Used Techniques and Differences with Service- and Microservice-Based Systems," in *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 2018.

[16] A. Villa, J. O. Ocharán-Hernández, J. C. Pérez-Arriaga, and X. Limón, "A Systematic Mapping Study on Technical Debt in Microservices," in *2022 10th International Conference in Software Engineering Research and Innovation (CONISOFT)*, 2022.

[17] N. Kozanidis, R. Verdecchia, and E. Guzman, "Asking about Technical Debt: Characteristics and Automatic Identification of Technical Debt Questions on Stack Overflow," in *Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, Association for Computing Machinery, 2022.

[18] V. R. Basili, G. Caldiera, and D. Rombach, "The Goal Question Metric Approach," in *Encyclopedia of Software Engineering*, Wiley, 1994.

[19] P. C. Avgeriou, D. Taibi, A. Ampatzoglou, F. Arcelli Fontana, T. Besker, A. Chatzigeorgiou, V. Lenarduzzi, A. Martini, A. Moschou, I. Pigazzini, N. Saarimaki, D. D. Sas, S. S. de Toledo, and A. A. Tsintzira, "An Overview and Comparison of Technical Debt Measurement Tools," *IEEE Software*, vol. 38, no. 3, 2021.

[20] M. I. Rahman, S. Panichella, and D. Taibi, "A curated Dataset of Microservices-Based Systems," in *Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution*, CEUR-WS, 2019.

[21] V. Kovalenko, F. Palomba, and A. Bacchelli, "Mining file histories: Should we consider branches?," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[22] J.-L. Letouzey, "The SQALE method for evaluating Technical Debt," in *2012 Third International Workshop on Managing Technical Debt (MTD)*, 2012.

[23] K. W. Hipel and A. I. McLeod, *Time series modelling of water resources and environmental systems*. Elsevier, 1994.

[24] W. S. Cleveland and S. J. Devlin, "Locally weighted regression: an approach to regression analysis by local fitting," *Journal of the American statistical association*, vol. 83, no. 403, 1988.

[25] D. Ollech and K. Webel, "A random forest-based approach to identifying the most informative seasonality tests," Discussion Papers 55, Deutsche Bundesbank, 2020.

[26] R. B. Cleveland, W. S. Cleveland, J. E. McRae, and I. Terpenning, "STL: A seasonal-trend decomposition," *J. Off. Stat*, vol. 6, no. 1, 1990.

[27] A. Atchison, C. Berardi, N. Best, E. Stevens, and E. Linstead, "A Time Series Analysis of TravisTorrent Builds: To Everything There Is a Season," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017.

[28] I. Malavolta, R. Verdecchia, B. Filipovic, M. Bruntink, and P. Lago, "How Maintainability Issues of Android Apps Evolve," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018.

[29] W. N. Venables and B. D. Ripley, *Modern Applied Statistics with S*. Springer, 2002.

[30] T. R. Derrick and J. M. Thomas, *Time Series Analysis: The Cross-Correlation Function*, ch. 7, pp. 189–205. Human Kinetics Publishers, 2004.

[31] P. J. Brockwell and R. A. Davis, *Time series: theory and methods*. Springer science & business media, 2009.

[32] C. W. Granger, "Investigating causal relations by econometric models and cross-spectral methods," *Econometrica: journal of the Econometric Society*, 1969.

[33] H. Akaike, "Information theory and an extension of the maximum likelihood principle," in *Selected papers of hirotugu akaike*, pp. 199–213, Springer, 1998.

[34] R. Dean and W. Dunsmuir, "Dangers and uses of cross-correlation in analyzing time series in perception, performance, movement, and neuroscience: The importance of constructing transfer function autoregressive models," *Behavior research methods*, vol. 48, 2015.

[35] D. A. Dickey and W. A. Fuller, "Distribution of the estimators for autoregressive time series with a unit root," *Journal of the American statistical association*, vol. 74, no. 366a, pp. 427–431, 1979.

[36] L. Baresi, G. Quattrocchi, and D. A. Tamburri, "Microservice Architecture Practices and Experience: a Focused Look on Docker Configuration Files." Preprint, 2022.

[37] G. Muntoni, J. Soldani, and A. Brogi, "Mining the Architecture of Microservice-Based Applications from their Kubernetes Deployment," in *Proceedings of the 16th International Workshop on Engineering Service-Oriented Applications and Cloud Services (WESOACS)* (Springer, ed.), 2021.

[38] J. Soldani, G. Muntoni, D. Neri, and A. Brogi, "The μTOSCA toolchain: Mining, analyzing, and refactoring microservice-based architectures," *Software: Practice and Experience*, vol. 51, 2021.

[39] N. Alshuqayran, N. Ali, and R. Evans, "Towards Micro Service Architecture Recovery: An Empirical Study," in *2018 IEEE International Conference on Software Architecture (ICSA)*, IEEE, 2018.

[40] F. Rademacher, S. Sachweh, and A. Zündorf, "A Modeling Method for Systematic Architecture Reconstruction of Microservice-Based Software Systems," in *25th International Conference on Exploring Modeling Methods for Systems Analysis and Development (EMMSAD 2020)*, 2020.

[41] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, "MicroART: A Software Architecture Recovery Tool for Maintaining Microservice-Based Systems," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017.

[42] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, vol. 14, 2009.

[43] C. Wohlin, M. Höst, and K. Henningsson, "Empirical research methods in software engineering," *Empirical methods and studies in software engineering: Experiences from ESERNET*, 2003.

[44] A. Eckner, "A note on trend and seasonality estimation for unevenly-spaced time series," *eckner.com*, 2012.

[45] T. Amanatidis, A. Moschou, N. Mittas, A. Chatzigeorgiou, A. Ampatzoglou, and L. Angelis, "Evaluating the Agreement among Technical Debt Measurement Tools: Building an Empirical Benchmark of Technical Debt Liabilities," *Empirical Software Engineering*, vol. 25, 2020.

[46] B. Podobnik and H. E. Stanley, "Detrended cross-correlation analysis: a new method for analyzing two nonstationary time series," *Physical review letters*, vol. 100, no. 8, 2008.

[47] S. Ospina, R. Verdecchia, I. Malavolta, and P. Lago, "ATDx: A tool for Providing a Data-driven Overview of Architectural Technical Debt in Software-intensive Systems," in *European Conference on Software Architecture (2021)*, 2021.

Online Resources

[48] D. Taibi, "A curated list of open source projects developed with a microservices architectural style." `https://github.com/davidetaibi/Microservices_Project_List`. *last accessed 01/09/2023.*

[49] Jetbrains, "The state of developer ecosystem 2022." `https://www.jetbrains.com/lp/devecosystem-2022/`, 2022. *last accessed 11/09/2023.*

List of Figures

List of Tables