Highlights

Evolution of Code Technical Debt in Microservices Architectures

Kevin Maggi, Roberto Verdecchia, Leonardo Scommegna, Enrico Vicario

- Technical debt in microservices grows much faster during initial development phases.
- Technical debt growth is strictly related to the number of microservices.
- Technical debt growth rate does not to vary when microservices are added or deleted.
- Heterogeneous activities can introduce technical debt to different extents.
- Technical debt removal relies on specific actions aimed to improve software quality.

Evolution of Code Technical Debt in Microservices Architectures

Kevin Maggi[a,*], Roberto Verdecchia[a], Leonardo Scommegna[a], Enrico Vicario[a]

[a]*Department of Information Engineering, University of Florence, Via di S. Marta 3, 50139, Florence, Italy*

Abstract

Context: Microservices are gaining significant traction in academic research and industry due to their advantages, and technical debt has long been a heavily researched metric in software quality context. However, to date, no study has attempted to understand how code technical debt evolves in such architectures.
Aim: This research aims to understand how technical debt evolves over time in microservice architectures by investigating its trends, patterns, and potential relations with microservices number.
Method: We analyze the technical debt evolution of 13 open-source projects. We collect data from systems through automated source code analysis, statistically analyze results to identify technical debt trends and correlations with microservices number, and conduct a subsequent manual commit inspection.
Results: Technical debt increases over time, with periods of stability. The growth is related to microservices number, but its rate is not. The analysis revealed trend differences during initial development phases and later stages. Different activities can introduce technical debt, while its removal relies mainly on refactoring.
Conclusions: Microservices independence is fundamental to maintain the technical debt under control, keeping it compartmentalized. The findings underscore the importance of technical debt management strategies to support the long-term success of microservices.

Keywords: Microservice, Technical Debt, Software Evolution

1. Introduction

In the past decade, microservices architectures (MSAs) [28] have gained popularity in academic research and have been increasingly adopted in industry practice. In addition to high scalability and flexibility, microservices independence throughout the entire software lifecycle is perhaps the main benefit. The maintenance phase is highly beneficial for code quality as it heavily relies on it. Studies [17] show that the resulting software quality in MSAs is perceived positively, making the maintenance phase easier to handle.

Despite many benefits, MSAs also present challenges, such as consistency management [76] and the additional effort required for integration and system testing [42]. Additionally, adopting the microservice architectural style can lead to a potential loss of the bigger architectural design and context [62, 75]. To face the increased complexity, developers resort to sub-optimal implementation expedients, namely the Technical Debt (TD) [10]. While providing temporary benefits, these solutions cause a considerable drop in software quality over time, making future development interventions harder or even impossible. If left unmanaged, TD leads to slowed development, more bugs, or the need to completely rewrite a system from scratch [70].

TD, and especially the Code TD type [4] on which this study focuses, has been extensively addressed in literature for years [3, 40]. Similarly, albeit being popularized around a decade ago, the microservice

*Corresponding author
Email addresses: kevin.maggi@unifi.it (Kevin Maggi, roberto.verdecchia@unifi.it (Roberto Verdecchia), leonardo.scommegna@unifi.it (Leonardo Scommegna), enrico.vicario@unifi.it (Enrico Vicario)
ORCID(s): 0009-0001-0651-805X (Kevin Maggi), 0000-0001-9206-6637 (Roberto Verdecchia), 0000-0002-7293-0210 (Leonardo Scommegna), 0000-0002-4983-4386 (Enrico Vicario)

architectural style is still gaining considerable academic interest [25, 65]. It is thus surprising that, despite both TD and MSAs being popular topics in literature, only a handful of studies have to date focused on the relationship between the two subjects. Even more amazing that, to the best of our knowledge, no one has ever quantitatively investigated in-depth how TD evolves in MSAs.

Given the novelty in this area, *i.e.,* the evolution of TD in MSAs, this work aims to be a stepping stone towards a new topic investigation. As a first step, we consider the Code TD, one of the most widespread TD types in literature [56, 74] and the most monitored by practitioners [31] due to the prevalence of tools such as SonarQube [11]. However, we plan to extend this exploration to Architectural TD and smells.

In a previous work [72], we conducted a preliminary case study on the TD evolution of a single MSA project constituted by 13 microservices and counting 977 commits and 38k lines of code. From the study, we observed that Code TD can constantly increase over time or present long periods where TD remains constant while development activities carry on. In addition, we observed that the TD increase or decrease occurs consistently, regardless of the number of microservices currently present in the system.

This study aims to generalize the research by considering 12 additional systems to corroborate or disprove the case study findings through the collection and statistical analysis of data along the history of the repositories. Understanding how Code TD evolves in microservice-based systems is necessary for researchers and practitioners to know how it can be effectively managed in such architectures, with the end goal of better supporting the long-term success of software-intensive systems based on such architectural style.

We can generalize the main conclusions from the preliminary study: the growth of TD is correlated with the number of microservices, but the growth rate is not. Considering more cases, we can extend our results by identifying additional aspects of TD evolution in MSAs, *e.g.,* differences between the initial and subsequent development phases and more nuanced findings regarding the activities that led to the introduction of TD.

The main contributions of this research are:

(i) An in-depth examination of Code TD through the evolution of 13 MSAs spanning over 10k commits;
(ii) An exhaustive statistical analysis of the TD evolution and its relation to number of microservices; and
(iii) A comprehensive replication package[1] containing the entirety of the raw, intermediate and final data, analysis traces, and scripts used to collect and analyze the data on which our findings are based.

Paper Structure This paper is structured as follows. Section 2 provides the background information required, while Section 3 discusses related work. Sections 4 and 5 describe, respectively, the design and practical aspects of the study, including a detailed description of the research process and the dataset. Section 6 presents the results, and Section 7 offers an in-depth discussion of the findings. Section 8 addresses threats to validity, and finally, in Section 9, conclusions are drawn, and potential future works are outlined.

2. Background

In this section, we describe the two main topics of this work, namely: (i) TD, what it is, what causes it, and why developers do not manage it (or at least do not fully pay it off); and (ii) MSAs, with a closer look at the aspects that can either help or impede TD management in systems architected in such style.

2.1. Technical Debt

According to Avgeriou *et al.* [10], the software engineering community converges on defining TD as

> *"a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible; impacting internal system qualities, primarily maintainability and evolvability."*

[1] `https://doi.org/10.6084/m9.figshare.25922506`

Various types of TD have been introduced in literature [4], but Architectural TD, Code TD, and Design TD are the most studied [56]. Code TD (CTD) is one of the most considered TD types and refers to elements in the source code that negatively affect its complexity, legibility, and correctness, making source code harder to maintain; in other words, it regards issues related to bad coding practices [41]. Indices examples are code metrics (*e.g.,* cognitive or cyclomatic complexity), code duplication, slow algorithms, and code outside standards [4]. Architectural TD (ATD) takes into account ill-suited architectural decisions with indicators like issues in software architecture and structural dependencies [4] that might lead to slow deterioration of the architecture, causing performance degradation or even runtime failures [73]. Instances of ATD are non-uniformity of architectural patterns and policies, dependencies violations (including modules and external dependencies), non-modularity, compliance violations, and inconsistency in managing integration with subsystems [14]. Design TD (DTD) concerns issues at the design level and involves instances of code smells, god classes, dispersed or intensive coupling, and schizophrenic classes [4]. Some authors also consider DTD items a few types of ATD or CTD instances, such as design issues, duplicated code, and code metrics [77], leading to a partial overlap [41]. The literature defines other types of TD, *e.g.,* Requirements TD, Test TD, Build TD, Documentation TD, Defect TD, Community TD, and Versioning TD, but their appearance is much less frequent [41].

Ramač *et al.* [54] aggregate results from an international family of surveys conducted by researchers from the TDresearchteam[2] over six countries. The goal was to provide a comprehensive overview of common causes and effects of TD in the industry. From the results, the main reason for introducing TD is the deadline rush. Aspects relating to project management and technical decisions also appear frequently: inappropriate planning, ineffective project management, lack of well-defined processes, focusing solely on productivity at the expense of quality, and non-adoption of good practices. In addition to technical and managerial factors, human factors like pressure and lack of experience are also involved. Concerning the effects of high TD, the most commonly observed effects are delivery delay and low maintainability. These effects result from bad code, which could lead to increased costs, low performance, more effort, and the need for refactoring and rework. Lastly, although not so cited by interviewees, financial loss is potentially one of the most impacting effects of TD.

Introducing TD might be inevitable, particularly in the early phases of software development. However, according to Freire *et al.* [30], only a minority of practitioners pay off the TD, both for technical and managerial reasons. A large portion of interviewees report as decision factors a focus on short-term goals, but others also cite a lack of organizational interest, time and resources, and high cost and effort. Bomfim *et al.* [43] briefly outline the underlying causes of these reasons: high effort might be due to a lack of test coverage and insufficient knowledge from the developers of all parts of the system, whereas the organizational interest is influenced probably by the belief that paying off TD has a low impact for the business.

2.2. Microservices Architecture Characteristics and Implications on Technical Debt

An MSA is a distributed application in which all its modules are microservices, defined as cohesive, independent processes that interact *via* a messaging system [28]. Each microservice can be developed independently, fulfilling a specific business functionality, and executed independently, enabling easy deployment in cloud environments through containerization [69]. This architectural style refers to various principles, widely and rigorously discussed by Velepucha and Flores [69], guaranteeing systems flexibility, modularity, and evolution. These characteristics ensure that an application can keep up with the changing business guided by the market, thanks to the ease of maintenance and implementation of new features [28].

Componentization involves creating small, independently deployable microservices exposed in a way that hides internal details. *Organization around business capabilities* (substantially a strict application of Domain-Driven Design [29] and single-responsibility principle [59]) refers to dividing the business to implement each microservice by focusing on single functionalities rather than technical layers. The *evolutionary design* is a natural consequence of these principles: each microservice can evolve autonomously, meaning that can be replaced with a new implementation or even discontinued without affecting the system.

[2] `www.tdresearchteam.com/home` (accessed on 20th April 2024)

From TD management perspective, one of the aspects of MSA that has the greatest impact is the *decentralized governance*, which allows the organization of developers in small independent teams (also known as two-pizza teams [55]). On the one hand, such teams are responsible for managing only a microservice within the overall system, which helps to contrast the lack of knowledge of developers; on the other hand, managing resources (both human resources and time) might be challenging if not handled at the managerial level.

According to Bogner *et al.* [15], in the industry, TD is usually not handled differently in service- and microservice-based architecture. Despite this, variations from other system architectures sometimes exist and are mainly related to applied processes, such as different testing processes (*e.g.,* integration testing) or diverse development methodologies (*e.g.,* the Agile approach).

2.2.1. Architecture Stability

The *Stability* of a system is the extent to which the software product can resist unexpected effects from modifications of the software [2]. Martin proposed an *Instability* metric [46] as a function of *Coupling* measures, which quantifies the instability of packages, *i.e.,* the degree of difficulty in changing a package without affecting the correct functioning of other packages. Hence, *Stability* is strictly related to a system's evolution.

Instability has a direct impact on TD, in particular, the Architectural TD, since it relates to design and architecture levels. Instability issues can lead to Architectural Smells (ASs), known as Instability ASs, such as *Unstable Dependencies*, where a component relies on other, less stable subsystems [7]. As *Unstable Dependencies* increase, the risk of necessary change to a central component inevitably grows, and the effort required for future maintenance actions, *i.e.,* the TD, raises. In this context, an Architectural Debt Index (ADI) that detects these ASs by measuring Martin metrics [46] has been proposed [8, 57, 60]. Moreover, this index keeps track of ASs history, making it oriented to the healthy evolution of systems.

However, according to Capilla *et al.* [19], in the microservices context, *Unstable Dependencies* have a limited impact on the overall ATD; the authors hypothesize that this aspect could be a consequence of MSA principles, such as componentization and evolutionary design.

3. Related Work

In a previous work [72], we examine a big open-source software (OSS) project based on MSA. We look at (i) the characteristics of the TD trend from a qualitative point of view, noting that despite an overall growing trend, there are several periods with no significant TD increase; and its seasonality, finding that is absent; and (ii) the correlation between the number of microservices and the amount of TD, which seems to exist and be meaningful. This study further explores these aspects to determine if findings can be generalized to a broader set of MSAs, by considering 12 additional MSA systems implemented in different languages. This study also complements the findings with other facets, including a quantitative evaluation of the trend, identification of possible common trend patterns, a focus on the activities performed in the commits associated with the most outlier values in TD variations, and an in-depth analysis of correlations considering various time lag displacements.

A thorough study on the evolution of microservices is carried out by Assunção *et al.* [9] involving 11 OSS projects. They aim to understand how microservices evolve from 3 different perspectives: technological (the technologies enabling microservices development), services (the business logic of the system), and miscellaneous (system-specific aspects). The findings reveal that rarely the evolution is service-based, with the vast majority of commits related to technical changes. Since both technical and services evolutions can impact the TD to different extents, this study complements that one, by utilizing the more narrow and focused lens of TD in MSAs.

By considering the literature that focuses on Code TD evolution in MSAs, we can identify, at our best, only few studies. The work of Lenarduzzi *et al.* [38] may be the most similar to this one. They investigate the effect of migrating from a monolithic architecture to a microservices one on Code TD in a case study. Results show the TD growing slower after the migration. In fact, although an initial rapid growth, the TD tends to stabilize and to grow slower with microservices than in the monolithic system. This study differs

from ours for at least two reasons, namely (i) we do not focus on the migration of code from a single, solid monolith to multiple, independent microservices, and (ii) we consider a large number of study cases, some of which are even bigger than the small case (5 microservices) considered by Lenarduzzi *et al.* [38].

Bogner *et al.* [16] conduct a study to examine how the sustainable evolution of 14 microservice-based systems is ensured through 17 semi-structured interviews. The main focus is not on TD, but it emerges as one of the main issues that undermine sustainable evolution. Additionally, although the study mentions some tool-based DevOps processes, it is based on a qualitative research method. In a different work, Bogner *et al.* [15] survey 60 software professionals on how TD can be limited through maintainability assurance. The results indicate that using systematic techniques benefits software maintainability. However, these tools are underutilized in practice. Both studies adopt a qualitative approach and do not explicitly account for the TD and its evolution, unlike this one.

By examining other related literature, TD is primarily analyzed from an architectural perspective in the context of microservices. Such studies are facilitated by tools that detect and measure ATD. Arcan [6] is one of the most known and used; it implements TD indexes that consider ASs like *Cyclic*, *Unstable*, and *Hub-Like Dependencies* [57, 60]. It also identifies microservice-specific smells like *Shared Persistence* and *Hard-Coded Endpoints* [51]. ATDx [50] provides a TD index [71] incorporating rules from third-party tools, including Arcan. Sen4Smell [26] acts as a wrapper for Arcan, prioritizing ASs based on their evolutionary history. Lastly, Aroma [12] evaluates ASs like *No API Gateway* through dynamic analysis of microservices.

Capilla *et al.* [19] study the ATD presence on MSAs and, similarly to our work, try to analyze the relation with microservices. Interstingly, their results indicate no relation between ASs and system size (in terms of LOC and microservices). However, they conjecture that ASs might be related to the complexity, as a high number of dependencies remarkably raises the chances of having cyclical or unstable dependencies. In contrast to our research, their work does not focus on the analysis of evolution over the systems' lifespan.

Sas *et al.* [61] explore how instability-related ASs evolve by tracking their history. They find that while the total number of ASs does not significantly increase and often rises only temporarily, the number of components involved grows, leading to tighter, almost inseparable connections. Unlike this work, our approach does not trace the evolution of individual smells but only the general progress of TD.

de Toledo *et al.* [68] conduct a multiple case study to investigate the ATD in microservices from a qualitative point of view. The study, involving 25 interviews, identifies ATD issues, their negative impact, and common solutions to repay them. In a similar work, de Toledo *et al.* [66] focus on ATD in the communication layer through a qualitative analysis of documents and interviews related to a large case study. These studies differ from ours in that the research method is primarily qualitative instead of quantitative, and the analysis does not address the evolution in a time window.

Likewise, Zhong *et al.* [78] analyze the impacts, causes, and solutions of ASs as possible root causes of ATD accumulation in MSAs. According to their findings, ASs can arise either inadvertently or intentionally, often due to business aspects and responsibility management. Frequently, the only solution is to reconstruct microservices and improve the organization of responsibility and communication among teams. This study investigates specifically six types of ASs while our aim is not to address specific smells, whether related to code or architecture, but an overall TD overview. In line with the microservices reconstruction theme, Brogi *et al.* [18] propose an approach integrated into the μTOSCA toolchain by Soldani *et al.* [64], which enables ASs automatic identification and selection of suitable refactorings to resolve them. de Toledo *et al.* [67] investigate the benefits of repaying ATD on MSAs. Their results evidence the importance of this activity on future incidents. Specifically, their total number is notably reduced, with a remarkable reduction of critical and high-priority incidents. Conversely, our focus, as of now, is not on the benefits of repaying the TD but rather its evolution during the software lifecycle.

A more systematic scrutiny of the literature on TD in microservices is conducted by Villa *et al.* [74]. Based on the analysis of the 12 primary studies they select, they somehow confirm the intuition on which this research is based, namely the absence of studies focusing on the TD evolution in microservice-based systems. Moreover, they find that Architectural, Code, and Design TD are the most common types of TD reported in microservices contexts. This result aligns with the general trend observed in developer discussions on Stack Overflow, as highlighted by Kozanidis *et al.* [37], which analyzes TD-related questions. These findings firmly support the focus of this work, namely the evolution of Code TD over the lifecycle of MSAs.

Summary To summarize the current state of the literature and highlight the differences with our work: the key novelty of our study lies in its focus on the quantitative analysis of Code TD evolution in MSAs, over time and across multiple case studies. Unlike previous research, which often focus on migration scenarios, qualitative methods, or specific architectural aspects and smells (even tracking them over time), our study provides a broader view of Code TD progression throughout the lifecycle of MSAs and its overall relationship with architecture evolution.

4. Study Design

In this section, we outline the research design of the study in terms of research goal (Section 4.1), research questions (Section 4.2), and research process (Section 4.3).

4.1. 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.* [13], 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 TD*
> ***From the viewpoint of*** *software engineering researchers*
> ***In the context of*** *microservice-based software-intensive systems.*

This study aims to be a stepping stone towards extensively investigating how MSAs evolve from the TD point of view. A comprehensive understanding of specific TD characteristics and trends could help propose improvements for current MSA development practices and support tools. In fact, by knowing that, developers could roughly predict the impact of incoming interventions on the total TD and take appropriate action, such as scheduling extensive special maintenance to lower the TD and bring maintainability back to acceptable levels before a major change in architectural or business levels (*e.g.,* the introduction of new microservices or features).

The focus of this first step into the evolution of TD in MSAs is the influence of such architectural style adoption on code maintainability. Therefore, our focus is on Code TD rather than Architectural TD, whose main causes (*e.g.,* architecture smells) are already at an higher level of abstraction. The choice of considering Code TD, moreover, is influenced by other factors too. Firstly, Code TD is one of the most frequent types of TD appearing in microservice-based systems [74]; secondly, differently from other types of TD, Code TD is supported by a wide range of consolidated tools that are extensively used both in academic research and industrial practice [11], and consequently, it is by far the most considered TD type by practitioners [31]; lastly, for these reasons, it enables the study to consider a heterogeneous set of experimental objects.

4.2. Research Question

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

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

Rationale This RQ addresses the overall goal of the study, which is to investigate the evolution of Code TD in MSA. Examining TD evolution permits us to catch both the trend over time and distinguishing characteristics, such as its relation with architectural events (*i.e.,* variations in the number of microservices), sudden changes in conjunction with specific code activities, or different growth rates based on the current life stage. Such knowledge can lead to a better understanding of TD and its enhanced management.

For a more systematic approach, the RQ can be broken down into sub-RQs, each focusing on one of the different aspects of TD evolution that this research aims to explore. Specifically, we decompose it into the following two RQs, each associated with one or more hypothesis testing.

> **RQ_1**: *What is the evolution trend of Code TD in a microservice-based software-intensive system?*
>
> **$H_0^{1.1}$**: *TD evolution does not change in time*
>
> **$H_a^{1.1}$**: *TD evolution changes in time*
>
> **$H_0^{1.2}$**: *TD evolution does not present periodic trend*
>
> **$H_a^{1.2}$**: *TD evolution presents periodic trend*

With RQ_1, the aim is to understand the overall evolution trend of TD in microservice-based systems, *e.g.,* determining if it is constant through the evolution of the software system, if it shows a growing trend, or if a periodic trend can be observed. We consider natural to conjecture that there is an increasing trend (as resulted in [38]) with an observable periodic trend (*i.e.,* seasonal periods during which developers are more or less prone to incurring TD, such as before or after holidays, or approaching a periodic deadline).

Tight deadlines, like other business-related dynamics, are an important cause of TD introduction [22]. Industries often follow an established development roadmap, possibly including periodic activities. Detecting such a periodicity can help predict software evolution from a TD viewpoint [33].

Understanding how TD tends to increase/decrease over time might help in better managing financial and human resources. This means allocating them more effectively during periods (of the project's timespan or the year) when TD is more likely to occur. It also allows for identifying the relationship between TD fluctuations and specific code-related activities, which can help in focusing efforts on the most critical tasks.

> **RQ_2**: *Is there a relation between Code TD evolution and the number of microservices?*
>
> **H_0^2**: *TD evolution does not depend on the number of microservices*
>
> **H_a^2**: *TD evolution depends on the number of microservices*

With RQ_2, we aim to understand if there is a relation between the evolution of TD and the number of microservices that compose the system. We could hypothesize that, due to sub-optimal implementation choices, as the number of microservices increases, the overall complexity of the system increases as well and consequently the Code TD grows at a higher rate, possibly showing a superlinear or even exponential relation with the number of microservices.

Revealing an underlying connection between the two metrics can provide insights into how architectural changes, like adding or removing a microservice, affect the total TD. With this knowledge, developers can anticipate future Code TD trends based on planned architectural changes. This foresight allows for proactive, early measures to reduce TD before it becomes too burdensome.

4.3. Research Process

Following, we describe the research process that allowed us to answer the RQs: criteria used to define the dataset to analyze, how selected repositories have been inspected to gather data, and how this data can help to answer the RQs through a rigorous statistical analysis.

4.3.1. Repository Identification

As a dataset to be analyzed, we want a substantial amount of repositories of microservice-based software-intensive systems that have undergone an actual industrial-like development process, so we are looking for actual systems or industrial demos that are not toy examples or exercise-in-style demos. However, during our preliminary discovery phase, we found a lack of this type of OSS project, mainly because most companies do not make their products open-source. As a result, we also have to accept industrial demos, which are more widespread and, hopefully, are characterized by a similar development process.

Although many authors in the literature use the dataset provided by Rahman *et al.* [53] and its extension[3], most of the projects collected do not meet the just seen requirements. Therefore, we decided to define a custom dataset by querying GitHub and filtering the results to select the most interesting projects from an evolution standpoint, as shown in Figure 1. Querying and filtering criteria are summarized in Table 1.

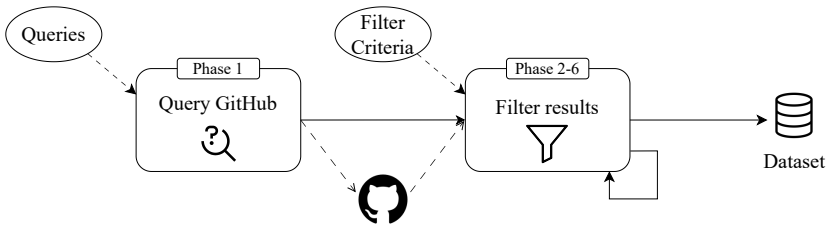

Figure 1: Repository selection process workflow

Phase 1: Query. In 2022, according to the "The State of Developer Ecosystem" annual survey conducted by JetBrains[4], the most commonly used programming language in microservices development is Java, followed by Python, Go, C#, TypeScript, and JavaScript. As a result, our research will focus on these six languages.

We used a composite query to select the most relevant repositories for MSAs from GitHub. Specifically, the query targets projects where the main programming language is one of those mentioned above and that either feature microservices-related GitHub topics (*i.e.,* "microservice", "microservices", "microservice-architecture", and "microservices-architecture") or include the keyword "microservice". Given the vast number of repositories on GitHub related to microservices (tens of thousands selected with topics and hundreds of thousands with keywords), we establish a star criterion to make an initial screening of the highest quality projects suitable for case studies, discarding repositories with only a few GitHub stars. In particular, we require that repositories collect at least 10 GitHub stars if selected through the topics or at least 100 GitHub stars if retrieved through keyword. We set these numbers empirically to avoid picking the most trivial repositories and to ensure a manageable number of results. The resulting query can be expressed as follows:

$$
\begin{aligned}
&\mathit{language} \in \{\text{Java},\ \text{Python},\ \text{C\#},\ \text{Go},\ \text{TypeScript},\ \text{JavaScript}\} \wedge \\
&\Big(\big(\mathit{GH\text{-}topic} \in \{\text{microservice(s)},\ \text{microservice(s)-architecture}\}\ \wedge\ \mathit{GH\text{-}stars} \geq 10\big) \vee \\
&\ \big(\mathit{keyword} \in \{\text{microservice}\}\ \wedge\ \mathit{GH\text{-}stars} \geq 100\big)\Big)
\end{aligned}
$$

Phases 2-6: Filters. To obtain repositories relevant for our research, the results need to be filtered to keep only those that meet the wanted prerequisites and discard the ones that are not embodying a relevant case study from an evolution viewpoint, *e.g.,* have no commit history or have a flat evolution. The Filter Criteria (FC) employed are:

[3] `www.github.com/davidetaibi/Microservices_Project_List` (accessed on 20th April 2024)
[4] `www.jetbrains.com/lp/devecosystem-2022/` (accessed on 20th April 2024)

FC_1: *The repository must have at least 250 commits.*
This criterion discards repositories unrepresentative of long-lived software applications that have already reached a certain maturity. An empirical threshold of 250 commits has been established, balancing the repositories' lifespan and the quantity of filtered results. The commits considered are those of the main branch and any secondary branches already merged into the main [36];

FC_2: *The repository must have at least 2 contributors.*
This criterion detects repositories that are the result of a single developer's effort (*e.g.,* trivially exercise-in-style and personal projects, which, from our observations, are generally developed by single practitioners and aspirants);

FC_3: *The repository must have a Docker compose file for at least ⅔ of the history and at least 250 commits.*
This criterion is essential since the method we exploit for detecting microservices, Claim [44], described more in detail in Section 4.3.2, has the use of Docker as a prerequisite;

FC_4: *The repository must present an MSA and be an actual industrial system or an industrial demo.*
This filtering step is the only one done manually by examining the description, the README file, and, in case of no clear clues, other artifacts, *e.g.,* Docker *compose file*s or the structure of the code. Starter kits, templates, and big-industry-related projects are also assumed acceptable, as from manual scrutiny result to be complex MSA systems developed in an industry-like setting. MSAs presenting a single microservice, only MSA components, examples from books/sites, libraries, development frameworks, and platforms are instead discarded with this criterion;

FC_5: *The repository must have a noteworthy evolution in the number of microservices.*
This last step is essential to select the final fine-grained set of the most engaging case studies among the filtered results. Our objective is to analyze case studies that already reached a certain level of maturity but are still evolving since evolution is our primary focus. We formally formulated the criterion based on: (i) the maximum number of microservices ever had; (ii) the longest period with flat evolution; and (iii) the number of commits with no microservices present.
To discard early stage prototypes and sluggish, crystallized systems, we reject repositories that have never counted at least 5 microservices and have spent more than half their life without any microservices number changes or more than a third of their life with no microservices at all. All the threshold values of this criterion have been found empirically *via* preliminary experimentation by tuning them to obtain an appropriate number of resultant repositories. The aim is to select the repositories most representative of a complex industrial system, near its maturity and that has not experienced a migration from other architectures (*e.g.,* the monolithic one) during its development.

Criteria	Purpose	Metrics
Query	Retrieve relevant cases for the MSA topic	Language, Github stars, GitHub topics
FC_1	Select repositories with a long enough lifespan	Number of commits on main branch
FC_2	Select repositories result of multiple developers' efforts	Number of contributors
FC_3	Select repositories using Docker as containerization framework	Docker *compose file* presence along history
FC_4	Select real-world or industrial demo MSAs	Manual inspection of description and README
FC_5	Select repositories with an active evolution	Number of microservices along history

Table 1: Overview of querying and filtering Criteria used to select the dataset

4.3.2. Repositories Analysis

The repositories analysis is outlined in Figure 2 and involves iterating through all the commits in each repository. For each commit, we count the number of microservices and measure the TD. After collecting all the analysis data, we analyze them to answer the RQs.

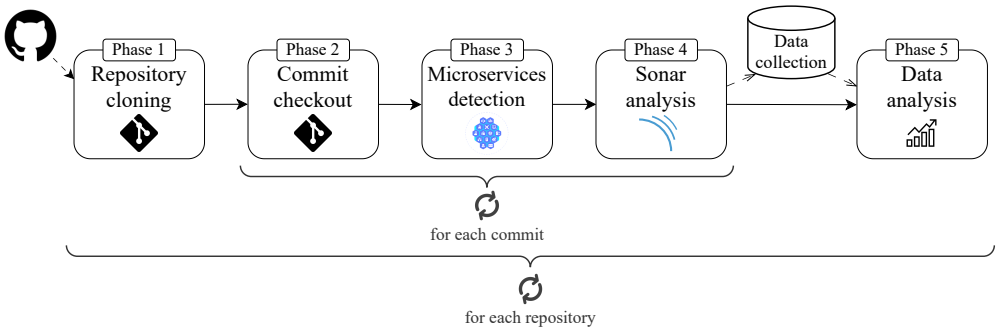

Figure 2: Dataset analysis process workflow

Phase 1: Repository cloning. During this step, we only clone the repository from GitHub.

Phase 2: Commit checkout. The second step is to checkout the commit to be analyzed, *i.e.,* restoring all the code artifact to the state they were in at that time. We consider only the commits of the main branch (and those of secondary branches already merged into the main, as highlighted by Kovalenko *et al.* [36]) linearized in temporal order. At this stage, we collect additional version-control-related data, such as the date, commit hash, and parent(s) hash, to preserve the graph topology information we lost during linearization.

Phase 3: Microservices detection. Before analyzing the TD, microservices composing the system are detected and enumerated. To perform this analysis, we use the CLAIM method [44], which exploits the inspection of Docker configuration files, *i.e.,* Docker *compose file*s and *Dockerfile*s. Specifically, CLAIM collects the *Dockerfile* generating the container image of each microservice, and identifies the Docker *compose file* used to spin up all the microservices and possible infrastructural components of the MSA architectural backbone. *Via* a set of empirical predefined rules, CLAIM determines which of the containers listed in the Docker *compose file* can be microservices, discarding other elements (*e.g.,* databases or gateways). As an additional check, CLAIM verifies that the candidate microservices are actually microservices by inspecting the correspondent Dockerfile to avoid false positive selections. Finally, CLAIM outputs the set of detected microservices, enabling us to count the total number detected.

The use of this tool for microservices detection sets limits on the repositories we can analyze, restricted to the ones using Docker as a containerization tool (see Section 4.3.1 for selection criteria); however, CLAIM is, to the best of our knowledge, the most accurate tool for this purpose (*i.e.,* language independent microservices detection) that is also lightweight to run, limiting the time effort to analyze thousands of commits. In Section 8, we discuss more thoroughly the possible effects on the validity of this study.

Phase 4: Sonar™ analysis. The last data to collect pertains to the analysis of code quality performed by the Sonar™ tool suite[5], one of the most known tools for TD measurement [11] (see Appendix B for the specific version of tools used). The metric of interest is the SQALE index, which measures the TD in minutes needed to pay off the identified Code TD [39].

The measurement includes all the languages supported by Sonar™ across the entire repository, not just the main one. However, for some languages, namely Java and C#, Sonar™ requires compiled code, so additional effort is needed to analyze such projects. Please note that in both cases, as per the specific Sonar™ versions work, the analyzed code, regardless of the language, is just those included by the building configuration files. Throughout the entire Sonar™ analysis, the default out-of-the-box configuration of the tool is utilized.

[5] `www.sonarsource.com` (accessed on 20th April 2024)

Phase 5: Data analysis. As the final step of the research process, we analyze the data collected. Before moving on, we clean the data by removing commits that incorrectly count no microservices due to a ill-formatted Docker *compose file*s (see Study Execution section, Section 5, for concrete values). This precaution ensures that spurious values do not influence the analysis.

To answer RQ_1, which is about the evolution trend of TD, various tests and investigations are conducted on TD evolution, trend, and seasonality:

- We test the overall trend presence through the Mann-Kendall test [35, 45], a non-parametric test based on Kendall's rank correlation coefficient (Kendall's τ). This test determines whether the TD time series measured with the Sonar™ tool suite exhibits a specific trend, either increasing or decreasing. While it can indicate the existence of a trend, it does not provide information about the slope of the trend, which we evaluated afterward relating to the microservices number. The non-parametric property does not assume a specific data distribution, *e.g.,* the Pearson correlation coefficient assumes normality, but our preliminary Shapiro-Wilk test [63] execution indicated that none of the systems exhibit normality in the TD values. Moreover, with respect to other tests for trend detection relying on other correlation coefficients (*e.g.,* Spearman), the Mann-Kendall test best fits the preconditions imposed by data; as a consequence, it is commonly employed in studies of software aging [21] and software evolution [47] (particularly concerning quality and smells [32, 48, 52]) with similar data characteristics and scope. An implementation of it is available as an open-source R library[6];
- We obtain the trend for graphical means analysis by applying LOESS regression [20], which smooths the TD evolution;
- We manually scrutinize the commits corresponding to "hotspots" of the TD time series. With "hotspots" we mean commits characterized by an anomalous value in the variation of TD:

  $$Hotspot \coloneqq \text{commit characterized by an outlier TD delta}$$

  We order the commits by the decreasing absolute value of introduced or paid-off TD, and we investigate the content of the top 10 for each repository considered;
- We conduct a seasonality test using the combined method by Ollech and Webel [49] (a QS test combined with a Kruskall-Wallis test). An implementation of this test is available as an open-source R library[7]. Note that this test requires at least two periods of data, namely two years; hence, we have to ignore systems with a shorter lifespan in this step, which constitutes a tradeoff between internal and external validity of our results.

The LOESS regression and the seasonality test require a regularly spaced time series, however commits are not. To address this potential impediment, we transform the series by (i) keeping only the last commit of each day and (ii) introducing synthetic linearly interpolated data points on missing days.

To answer RQ_2, which is about the relation between TD and the number of microservices, we examine the potential correlation and causation between them, in particular:

- We clarify whether a correlation exists between the two time series using the Cross-Correlation Function (CCF) [24]. This test checks the correlation between the two time series at different lags, *i.e.,* offset by some commits forward and backward;
- When a correlation at negative lags is detected, *i.e.,* TD time series correlates with delayed microservices time series, we conduct the Granger Causality test [34] to determine if a microservices number change causes an increment or decrement of TD. We calculate the optimal lag order using the Akaike Information Criterion [1];
- We analyze the potential correlation between the derivative of TD time series and the microservices number to understand if the growth speed of TD depends on the number of microservices. In this case, we use CCF as well.

[6] `www.rdocumentation.org/packages/Kendall/versions/2.2/topics/MannKendall` (accessed on 20th April 2024)
[7] `www.rdocumentation.org/packages/seastests/versions/0.15.4/topics/combined_test` (accessed on 20th April 2024)

Before conducting the Granger Causality test, it is necessary to ensure that data is stationary; similarly, the CCF may provide spurious results in the case of non-stationary series [23]. To test for stationarity, the Augmented Dickey-Fuller test [27] is used, and when non-stationarity is detected, the data is made stationary by differencing them.

The choice of studying correlation at different lags is due to the fact that we detect a microservice at its insertion as a Docker container, which does not necessarily coincide with the start of its development. This aspect might introduce a delay between the TD and the number of microservices; conversely, we have an advance when the insertion as a Docker container occurs after finalizing a working version of the microservice.

5. Study Execution

The composed query described in the previous section retrieved[8] a total of 2491 repositories: this number refers to single results, excluding duplicated repositories resulting from multiple subqueries. After applying FC_{1-3}, only 121 repositories remain. With FC_4, the manual criterion, only 46 repositories are identified as actual or industrial demo MSAs. While discerning MSAs from libraries and development framework is quite objective, distinguishing a real-world or industrial demo MSA from practitioners' exercise-in-style or didactic examples is more subjective. Our judgment is based primarily on the description from the author(s) or other clear clues in the README[9]. Finally, we fine-tuned the parameters of FC_5 to find the best configuration that would result in a final dataset containing an appropriate number of repositories to be considered for analysis. The concluding selection comprises 15 repositories; however, during the research process execution, we have to discard two initially included repositories from our analysis: `dotnet-architecture/eShopOnContainers` faces severe building issues that impede the building of almost all the commits except for the last period (likely due to incompatibility between different versions of .NET), while for `microrealestate/microrealestate`, a suspicious TD constantly equal to 0 for a prolonged period makes us realize that it was a multi-repositories project during its early life.

The final dataset comprises 13 repositories, summarized in Table 2. The dataset is heterogeneous in terms of size (both as microservices and as Source lines of code, acronymized as SLOC), lifespan, and main programming language used. Focusing on the distribution of the main languages, it appears to reflect the language diffusion in the MSA context[10]. Due to these factors, we deem the dataset satisfactory from an external validity point of view.

During the research process execution, some commits of the systems requiring the build get lost due to committed failing builds, although we adopt strategies to minimize these occurrences. First, we apport minor fixes to the building configuration files (*i.e.,* Maven POM files, Gradle build scripts, and .NET `global.json` files), concerning primarily dependency version updates and formatting corrections, to allow the build to start. Secondly, we "force" the build when possible, letting the build not fail on non-blocking errors. While this process could effect the functionalities of the compiled system, it does not effect the subsequent static analysis utilized in this study to collect the results (see Section 4.3.2). Nonetheless, in a few cases of build failure where the root cause is unclear or an objective solution might not exist, we decide to discard the affected commits instead of using a subjective heuristic to resolve the issues. Examples of this case are (i) dependencies that do not exist anymore, (ii) dependencies that make the build fail, (iii) the absence of a build file (especially in very initial commits), and (iv) errors in the build file that are not objectively fixable.

The incidence of omitted commits can be found in Table 3. The number of omitted commits is relatively low, accounting for less than 1% of the entire dataset. Thus, we do not believe this factor could substantially impact the internal validity of our results. Additional considerations on this matter are documented in Section 8.

[8] Query executed on 11th Sept. 2023; filters applied on the same day referring to commits until 8th Sept. 2023.

[9] To enhance the transparency and reproducibility of this study, we provide two examples of MSAs considered as not meeting the required validity conditions: `NTHU-LSALAB/NTHU-Distributed-System` and `ahmsay/Solidvessel`.

[10] `www.jetbrains.com/lp/devecosystem-2022/` (accessed on 20th April 2024)

ID	Repository	Main lang(s)	Type	Comm.	Contrib.	SLOC	MS	From	To
$S01$	`1-Platform/one-platform`	TS	product	1502	21	417k	6	`a1c9466` 01/04/2020	`aa761bf` 09/08/2023
$S02$	`OpenCodeFoundation/eSchool`	C#	product	275	9	5k	3	`e556352` 03/30/2019	`46b19b5` 04/25/2023
$S03$	`ThoreauZZ/spring-cloud-example`	Java	demo	279	4	7k	9	`1ba3e35` 01/11/2017	`654cf9b` 08/14/2020
$S04$	`asc-lab/micronaut-microservices-poc`	Java	demo	384	9	39k	9	`5169e13` 07/25/2018	`9871a2e` 04/19/2021
$S05$	`bee-travels/bee-travels-node`	JS	demo	375	7	255k	5	`7cde267` 09/04/2019	`1228252` 12/09/2021
$S06$	`geoserver/geoserver-cloud`	Java	product	1027	11	67k	8	`9f6414d` 07/09/2020	`f8f3d90` 09/06/2023
$S07$	`go-saas/kit`	Go	product	629	2	116k	2	`e5a1488` 08/23/2021	`9194ce4` 09/07/2023
$S08$	`jvalue/ods`	TS, JS	product	1428	16	219k	6	`d4a4459` 03/26/2019	`843943f` 05/17/2022
$S09$	`learningOrchestra/mlToolKits`	Python	product	1214	14	7k	6	`b11a11a` 03/19/2020	`084699b` 05/11/2022
$S10$	`minos-framework/ecommerce-example`	Python	demo	1069	4	44k	9	`f4307ad` 06/02/2021	`9e3cdcc` 02/01/2022
$S11$	`nashtech-garage/yas`	Java, TS	demo	535	28	82k	8	`c88ec52` 01/18/2022	`4177b13` 09/08/2023
$S12$	`netcorebcn/quiz`	C#	demo	658	3	4k	4	`b70c960` 02/22/2017	`82fb546` 11/20/2018
$S13$	`open-telemetry/opentelemetry-demo`	TS	demo	572	88	55k	15	`75bf84f` 04/26/2022	`54aefe5` 09/07/2023

Comm.: *number of commits*; Contrib.: *number of contributors*; MS: *median number of microservices*

Table 2: Selected repositories overview

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%
$S06$	1027	11	1.07%
$S07$	629	0	0%
$S08$	1428	26	1.82%
$S09$	1214	0	0%
$S10$	1069	0	0%
$S11$	535	0	0%
$S12$	658	27	4.10%
$S13$	572	0	0%
Total	9947	92	0.92%

Table 3: Omitted commits per system

6. Results

In this section, we only intend to report the obtained results objectively, postponing a more in-depth discussion of the results and the RQs answers to Section 7, where also main lessons learnt and takeaways for developers and researchers are discussed. We accompany the report with details of the most representative data, while a comprehensive report can be found in the replication package as stated in Section 1.

6.1. RQ1: TD Evolution Trend

6.1.1. Overall Trend

By analyzing TD trends by graphical means, we can already get an overview of the overall trend (see Figures 3 for significant examples discussed later, while in Appendix A are reported all the other study cases). All the systems, with the exception of $S12$ and $S13$, show a generally accentuated increasing trend.

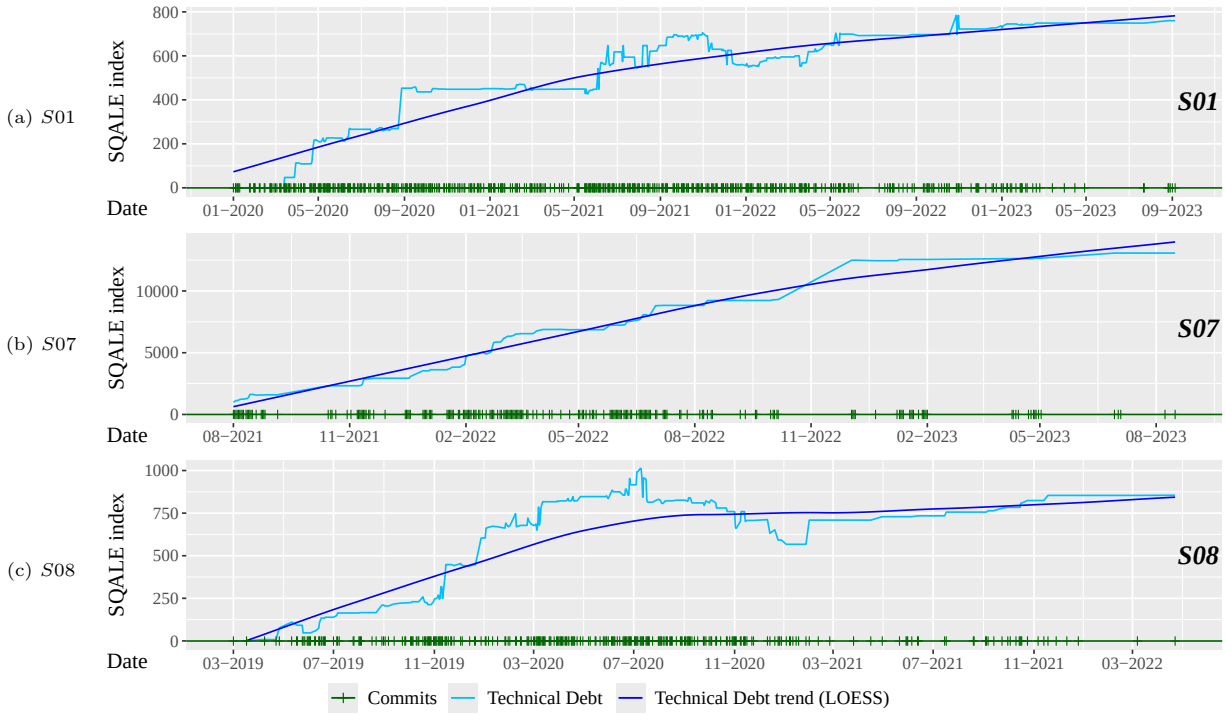

Figure 3: Trend of TD in systems a) $S01$, b) $S07$, and c) $S08$

The Mann-Kendall test gives a statistical verdict of the overall presence of a TD trend: all 13 analyzed MSAs present a trend (*i.e.,* a $p - value \leq 2.2e - 16$). Moreover, as shown in Table 4, almost all the systems have a considerably strong growing trend, *i.e.,* Kendall's τ close to 1. The only exceptions are $S12$ and $S13$, that show, respectively, a quite strong decreasing trend ($\tau = -0.58$) and a weak growing trend ($\tau = 0.23$).

IDs	Kendall's τ
$S01$, $S02$, $S03$, $S04$, $S06$, $S07$, $S09$, $S11$	$\tau \geq 0,79$
$S05$, $S08$, $S10$	$0,49 \geq \tau \geq 0,59$
$S12$	$\tau = -0,58$
$S13$	$\tau = 0,23$

Table 4: Kendall's τ on TD trend ($\tau = 1$: always increasing trend, $\tau = -1$: always decreasing trend)

Looking at the TD trend smoothed by LOESS regression, most of these systems (with the exception of $S06$ and $S11$, whose growth is quite linear) are characterized by two distinct phases: in their early phase of development, the TD growth rate is higher than in the later phase. However, for some cases, such as $S05$, $S01$, or $S08$ shown in Figures 3a and 3c, the difference is remarkable; for other cases, the sudden decline in the TD growth rate is milder, as in $S02$, $S04$, or $S07$ shown in Figure 3b. Systems $S12$ and $S13$ (see Figure 4), namely those with a weaker or negative trend, show a similar tendency characterized by a high initial TD followed by a constant decrease until a minimum and then a slow increase. While $S12$ finally stabilizes at a value lower than the initial, $S13$ keeps growing.

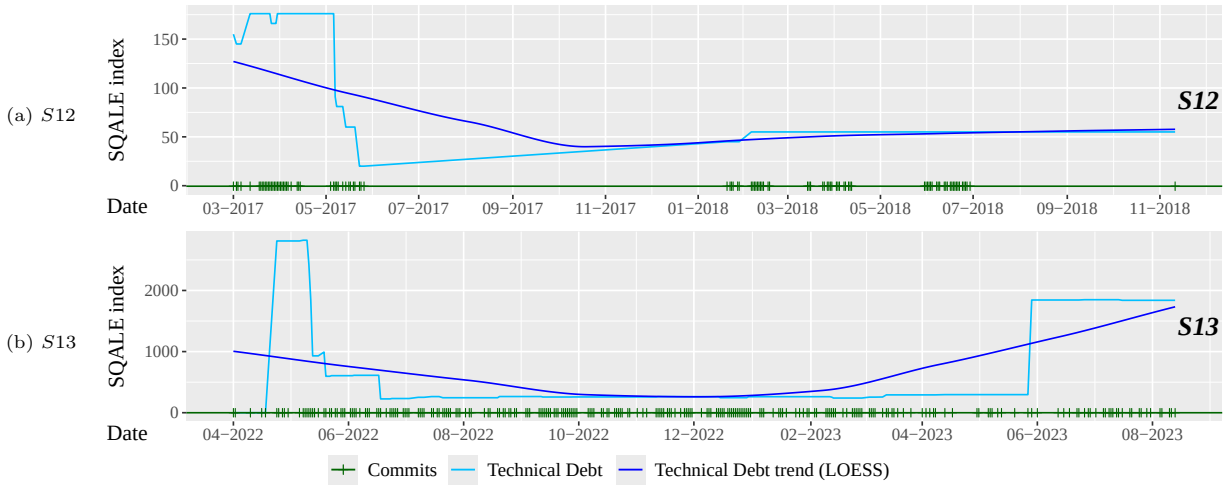

Figure 4: Trend of TD in systems a) $S12$ and b) $S13$

Further Investigation Observing these common patterns encourages us to delve deeper into possible underlying causes. Specifically, we focus on the two distinct phases with different TD growth rates. We examine the commit frequency and the age of the systems. After the elbow point, where TD begins to increase at a slower rate, it is often noted that the commit frequency decreases, and long periods of constant or decreasing TD occur. When considering the lifespan of the systems, projects with a shorter development, like $S07$ (which spans 2 years), seem to show a more subtle change than longer-lived systems, such as $S01$, which has nearly 4 years of development. However, this is not a property generalizable to all cases, as counterexamples exist, *e.g.,* systems $S04$ and $S05$ are against the trend. We discuss possible explanations for this phenomenon in Section 7.

Regarding the $S12$ and $S13$ systems, the high initial TD is instead due, as revealed by a manual inspection, to the presence of a considerable amount of code already accumulated at the first commit. Subsequent commits involved heavy refactoring and caused a sudden drop in TD before further development brought it back to slow growth.

6.1.2. Hotspots Manual Scrutiny

This manual step concerns, as explained in Section 4.3.2, inspecting the content of the commits characterized by the highest (positive or negative) variations of the TD to understand which developing activities and interventions have the best or worst impact on TD. The scrutiny concerns only the type of changes made in commits, and not the architectural components involved. The investigation of the top 10 hotspots for each system reveals that adding a component (be it a microservice, an infrastructural component, or a fronted UI) is the activity that introduces more TD, immediately followed by an evolution of the business logic, recorded respectively in 40% and 34% of the analyzed hotspots that add TD (see Table 5). However, these are not the only activities that can introduce TD since, in other cases, it has also increased simply by adding or upgrading a library/package (4%), or even by refactoring (13%) or fixing (3%). However,

refactoring (41%), with library upgrade (15%), fixing (13%) and testing (13%) to a lesser extent, is the main activity when a significant part of TD is paid back.

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 or even single small changes (*e.g.,* Java version update, library upgrade) introduce a notable amount of TD.

Intervention type	Incidence
TD introduction	*93 hotspots*
Adding components	40%
Evolve business logic	34%
Refactoring	15%
Framework/library/package change	4%
Fixing	3%

Intervention type	Incidence
TD removal	*39 hotspots*
Refactoring	41%
Framework/library/package change	15%
Fixing	13%
Testing	13%
Removing components	8%
Configuration change	8%

Table 5: Hotspots main detected activity

6.1.3. Seasonality

Moving on to the seasonality, the Ollech & Webell combined test returns a negative response for all the considered systems (*i.e.,* $p - value \geq 0.01$ and $p - value \geq 0.002$ in the two individual tests). The negative result of this statistical test proves the absence of any seasonality in the collected data, meaning that the introduction of TD does not follow a periodic pattern over the year. Therefore, in microservices contexts, TD can be introduced independently from annual events, like seasonal or national holidays or periodic deadlines, which seem to have no influence on it.

6.2. RQ2: Relation Between TD and Microservices

Before presenting the statistical test results, we note that graphical analysis suggests a potential correlation between the two time series, although this is not consistent across all commits and systems (the entirety of plots are shown in Appendix A).

Most systems exhibit a trend in TD that resembles that of microservices. For example, system $S06$, illustrated in Figure 5a, shows similar trend shapes, even if they are not identical. Both time series demonstrate comparable trajectories over time, characterized by a rapid increase in the initial phases, followed by stable periods interrupted by two noticeable steps.

In contrast, a few systems, such as $S07$ depicted in Figure 5b, display distinct trends. In this case, the nearly constant growth in TD appears to be independent of the number of microservices, which experiences a peak in the middle of the development process.

Focusing on the cases with comparable trends, for some of them the similarity is impressive. Take system $S11$ shown in Figure 5c, for instance. Even though the removal of a microservice in early May 2023 does not coincide with any change in TD and a sudden decline in TD in late March 2023 does not correspond with a change of microservices, the overall TD trend closely follows the evolution of the number of microservices.

By utilizing the statistical tests presented in 4, in the following sections we analyze if the visual observation holds statistical significance.

6.2.1. TD and Microservices Correlation and Causality

As Table 6 summarizes, CCF reveals in 9 systems out of 13 a considerably strong correlation at some lag. All the systems, except for $S13$, showcase the uppermost correlation at negative or no lag (*i.e.,* the TD never precedes the microservices number). $S13$ showcases a very high correlation of ≈ 0.8 at a positive lag, as shown in Figure 6, albeit weaker ones also characterize negative lags, as depicted in Figure 6 along with two other example cases, $S10$ and $S11$.

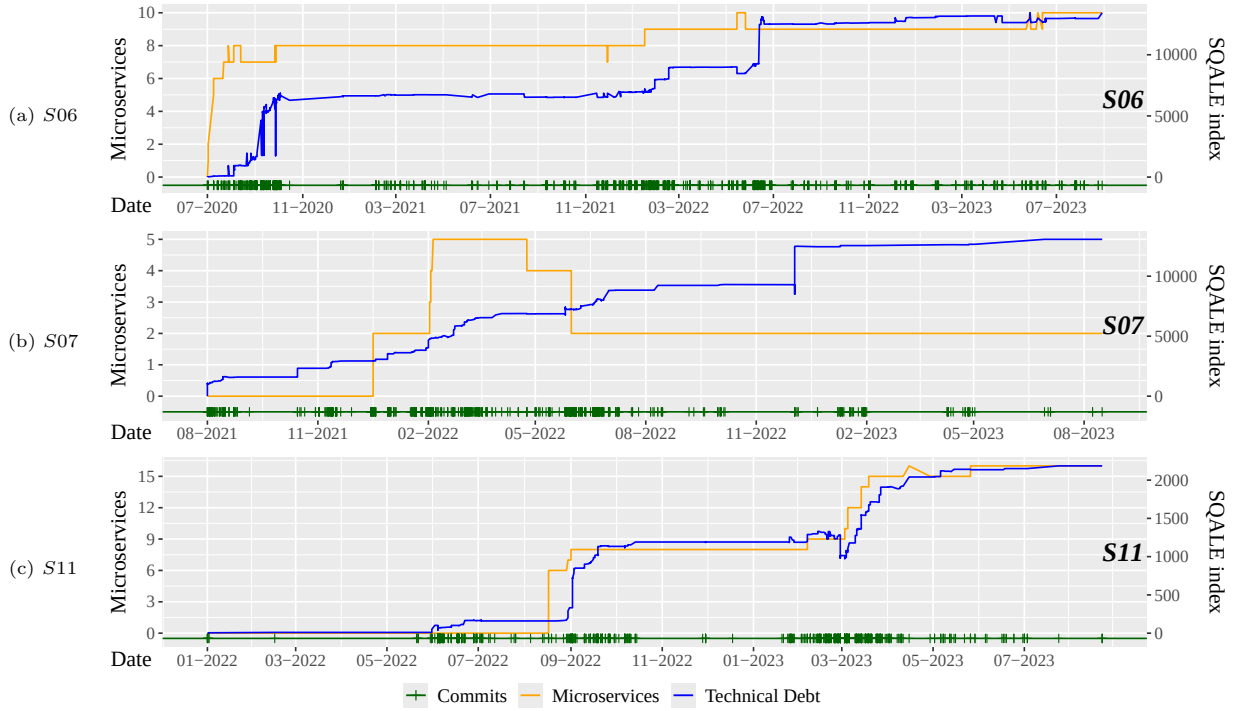

Figure 5: Evolution of TD and microservices number in systems a) $S06$, b) $S07$, and c) $S11$

This result indicates that not only are the two trends correlated but also that changes in the number of microservices typically occur before similar variations in TD. This observation confirms the correlation between the two time series and suggests a potential causality relationship where microservices may influence TD. It highlights the need for a causality test for systems that show a significant correlation.

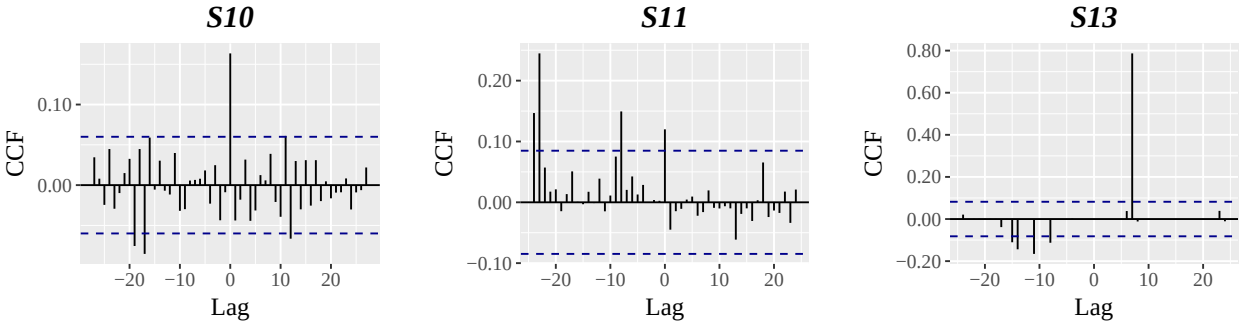

Figure 6: $S10$, $S11$, and $S13$ TD and microservices CCF

IDs	Cross-Correlation (at some lag)
$S01$, $S02$, $S08$, $S09$, $S13$	very strong ($>>$ confidence level)
$S06$, $S10$, $S11$, $S12$	strong ($>$ confidence level)
$S03$, $S04$, $S05$, $S07$	absent or very weak ($<$ *or* $\approx$ confidence level)

Table 6: Cross-Correlation between TD and microservices

The Granger causality test discloses that among the systems with a correlation at negative lags, only in 4 of them a causality exists (*i.e.,* $p - value \leq 0.01$), as outlined in Table 7. In other words, in 5 systems out of 9, although TD is correlated to the number of microservices, its variations are not caused by the introduction/removal of a microservice.

In light of this result, we cannot affirm that a causal relationship between the evolution of microservices and the introduction of TD exists as a general rule, rather it could be a characteristic of some specific MSA systems.

IDs	Granger causality	Summary
$S01$, $S06$, $S09$, $S13$	Yes	correlated and caused
$S02$, $S03$, $S08$, $S10$, $S11$	No	correlated but not caused
$S03$, $S04$, $S05$, $S07$	—	not even correlated

Table 7: Granger causality of TD by microservices number

Further Investigation These partially divergent results, which vary depending on the systems, encourage us to search for common patterns to understand the underlying reasons. Firstly, we observe that the systems where the two time series are not correlated ($S03$, $S04$, $S05$, and $S07$) share a common characteristic: they begin to accumulate TD since their very first commits, but a Docker *compose file* is introduced only after a significant number of commits. This suggests that the initial stages of development may have occurred without a clear architectural overview.

Then, concerning the systems that exhibit a causality relationship, their nature could be the key: in fact, systems $S01$, $S05$, and $S09$ are all real-world systems. Interestingly, while $S02$, $S07$, and $S08$ are also real-world systems, the causality test produced a negative result for them (in the case of $S07$, even a correlation is not detected). Although this pattern is apparent in some instances and could serve as a distinguishing factor, it cannot be generalized across all systems, as each one may represent a unique case.

6.2.2. TD Growth Rate and Microservices Correlation

Finally, the CCF between TD derivatives and microservices number suggests a possible correlation between them, existing in 6 systems out of 13, listed in Table 8. However, we can see in the plots in Figure 7 that a correlation at positive lags is always counterbalanced by one or more correlation at negative lags. As a take of this scenario, we can conclude that, from the statistic analysis, a significant correlation cannot be observed in the collected data. TD, therefore, grows at the same rate regardless of the amount of microservices present in the architecture.

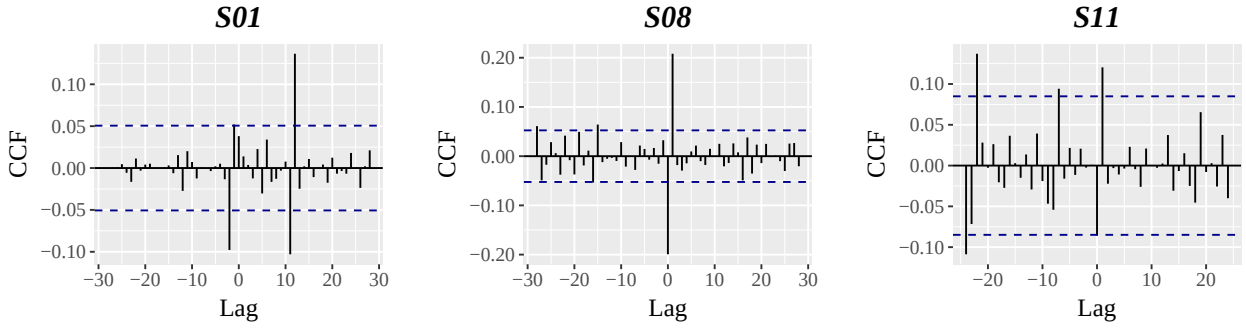

Figure 7: $S01$, $S08$, and $S11$ TD growth rate and microservices CCF

IDs	Cross-Correlation (at some lag)
$S06$, $S08$, $S09$	very strong (>> confidence level)
$S01$, $S02$, $S10$	strong (> confidence level)
$S03$, $S04$, $S05$, $S07$, $S11$, $S12$, $S13$	absent or very weak ($<$ *or* $\approx$ confidence level)

Table 8: Cross-Correlation between TD growth rate and microservices

7. Discussion

The results obtained allow us to answer the RQs, shedding light on many aspects of interest, also thanks to the additional qualitative inspections conducted to complement statistical results. Instead, on some other points, they do not lead to an absolute answer; nonetheless, they offer meaningful clues enabling conjectures, which can be further investigated in the future as suggested in Section 9.

7.1. RQ1: TD Evolution Trend

Undeniably, a growing trend exists for the TD since only 2 systems out of 13 do not showcase a strong Kendall's τ. Indeed, intuitively, TD tends to accumulate over time unless specific actions are adopted to manage it. We observed by graphical meaning the TD trend of $S12$ and $S13$, both characterized by a high initial value for TD. A further investigation reveals a common cause: both the systems start with a certain quantity of already developed code, and a non-null TD; the following phase consists of various refactoring that inevitably lower the TD accumulated in early commits or, in practice, even before the first commit.

While these responses already allow us to reject the null hypothesis $H_0^{1.1}$ with statistical confidence, concluding that in MSA systems, TD generally grows over time, the fact that in early phases, it grows faster suggests more complex dynamics.

As a subjective interpretation, the elbow point we observed in Section 6.1.1 could correspond to the passage from the initial intensive development to the maintenance phase, where adding new code is less frequent and refactoring, which can lower the TD, is much more common. Looking at Figure 3, this conjecture seems to find confirmations: (i) in younger systems, likely to be still in the development phase, the elbow point is less prominent, and (ii) after the elbow point, the commits frequency decreases significantly.

During the later stages of development, long decreasing periods can also be noticed, likely induced by efforts to pay back the TD. While this can be a virtuous practice, the real problem is in the early stages of development, where intense development pace, as commit frequency proves, and tight deadlines force developers to resort to sub-optimal expedients, as highlighted by [72].

As lesson learnt, developers should not worry too much about accumulating TD at the start of a project, as we observe it is an intrinsic phenomenon dictated by the initial development phases. At the same time, researcher should not obsess on tackling TD in the early phases of development, as it could counter the natural evolution of shaping a software system, leading to a loss of development effectiveness.

The most controversial point comes from the manual scrutiny, which detected refactoring as an activity that can either introduce or pay back TD, contrary to the common perception that a refactor inevitably reduces TD. This double face of refactoring is probably related to its intent either to clean up the code (thus paying back TD) or to prepare it for further development.

As takeaway, from our study we observed that refactoring activities do not necessarily guarantee a decrease of Code TD. In fact, we observed that refactoring activities can often result also in the opposite, i.e., a TD increase caused by refactoring activities. For developers, this implies that active effort should be spent not only on general code refactoring, but also on carrying out refactoring activities explicitly considering Code TD. For researchers instead, this entails that we cannot rely on the intuitive assumption that refactoring is always associated to a TD decrease.

About the seasonality of the TD trend, data are overwhelming, giving us enough statistical confidence to reject null hypothesis $H_0^{1.2}$ on the absence of seasonality. Therefore, contrary to what one might expect, TD is not more likely to be introduced/paid back in certain periods of the year (*e.g.,* before/after seasonal holidays).

RQ_1: *What is the evolution trend of Code TD in microservice-based software-intensive systems?*

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

TD variations can happen either by adding microservices or evolving the business logic, but many other activities can also be minor causes. Refactoring is the primary means of paying back TD, but only if undertaken with this purpose in mind, otherwise it will prove to be a double-edged sword.

TD does not show seasonal variations and can manifest throughout the year without any difference.

7.2. RQ2: Relation Between TD and Microservices

The values of the CCF between the TD measurement and the number of detected microservices are significant enough to affirm that a relation between the two metrics exists. Indeed, as seen in Section 6.2, only 4 systems out of 13 do not showcase a significant correlation, *i.e.,* greater than the confidence level. In light of this, we could already reject the null hypothesis H_0^2, concluding that a relation exists.

To learn more about the identified relationship, we need to look at the results of subsequent tests. The systems showcasing a weak or no correlation ($S03$, $S04$, $S05$, and $S07$) seem to have a common history, characterized by an early accumulation of TD but a late definition of architecture. Such a result highlights the importance of the architectural overview since the beginning of the development, otherwise the consequent uncontrolled accumulation of TD unrelated to architectural aspects will result.

The causality test, at first look, does not have such an overwhelming result: in 4 systems, a causality is detected, while in 5 not. Looking better, among the systems with a causation effect, we also found $S13$, where the correlation is at a positive lag, meaning that a change in the TD amount precedes a variation in the microservices number. As a result, albeit a causality relation exists, the latter metric can hardly be used to forecast future values of TD, *e.g.,* by framing and training a model (if enough data will ever be available) that guides developers and project managers to schedule maintenance in advance based on planned architectural intervention appropriately. It is challenging to detect a pattern characterizing the remaining systems (namely $S01$, $S06$, and $S09$) where a causality relation exists. The only possible connection among them is their nature of real-world systems. However, in other systems of this type, no causality ($S02$ and $S08$) or even correlation ($S07$) is detected. We can suppose that attested industrial workflows might be rigorous enough to establish, in some cases, a strict connection between the two metrics; however, further study is required to clarify such conjecture. Another possible explanation involves the system stability: although it is a property strictly related to the architectural level, its impact range can also extend to code quality, with developers forced to resort to suboptimal code expedients because of unhealthy architectural situations.

Regarding the TD growth rate, we started from the hypothesis that as the number of microservices increases, TD grows at a higher rate due to increased system complexity. However, CCF values disagree, showing a general non-correlation between the two metrics. Therefore, a desirable property appears: adding or removing a microservice has similar impacts on TD growth rate irrespective of the already present microservices. As a subjective interpretation, we hypothesize that this effect is a benefit of an appropriate adherence to the MSA principles [69], *i.e.,* componentization, business tailored microservices, decentralized governance, and evolutionary design (see Section 2.1). Through these principles, microservices are developed independently and in isolation by following a loosely coupled and highly cohesive architecture.

For developers, the observed relationship between TD and microservices implies that, by adhering to rigorous MSA principles, it is possible to keep TD compartmentalized at the microservice level. For researchers, this opens a new research direction, aimed at rigorously understanding how and why TD evolves within different microservices.

RQ_2: *Is there a relation between Code TD evolution and the number of microservices?*

Answer. There is usually a strong correlation between the TD and the number of microservices, although this may not always be the case in isolated situations. Generally, microservices number variations tend to be shifted in advance over the TD ones.

While there may be cases where even a causality relationship exists, our observations suggest it is not a general tendency.

The TD growth rate appears not to be significantly related to the microservices number. Consequently, adding or removing a microservice does not impact the growing rate of TD.

8. Threats to Validity

Despite our best efforts, the results of this study need to be interpreted in light of potential threats to validity, which might have influenced them. Certain factors may have affected the execution of the research or the collection and interpretation of results. In this section, we discuss these factors following the categorization of Runenson *et al.* [58].

8.1. Construct Validity

This category focuses on whether the measurements adopted are suitable for addressing the RQs. The TD measurement is conducted using SonarQube™, which employs the SQALE index. This tool and index are widely used in literature as well in industrial practice [10], but we are aware that using a different tool could have led to different measurements; however, according to Amanatidis et al. [5], the difference should not be significant.

The detection of microservices deserves a more detailed discussion. A static tool such as Claim may affect this metric since it can exploit less information than a dynamic method. However, a dynamic approach would have required excessive effort to configure each repository (and potentially commit) individually. To mitigate potential issues, we use the most accurate static tool available in the literature to the best of our knowledge. One limitation of Claim is the possibility of delay or advance in the microservices time series due to its underlying detection strategy. We address this by examining the correlation between the two time series at different lags with CCF.

8.2. Internal Validity

Here, we addressed potential confounding factors that could have influenced the recorded results to ensure the accuracy of our findings. To minimize any potential "noise" in the results introduced by TD measurement, we (i) discard all commits failing during the build for which we cannot find an objective fix without altering the original intentions of authors; (ii) carefully review commits characterized by an anomalous TD value and (iii) perform a rigorous statistical analysis on the collected data. As a last note, being the number of omitted commits relatively low, *i.e.,* $< 1\%$, we deem this factor could not affect results.

8.3. External Validity

In terms of generalizing the findings, we can reasonably assert that comparable results might be observed in other microservices-based software-intensive systems. Although there is a lack of MSA systems because most of them are commercial products and finding an open-source one is uncommon, the dataset is heterogeneous in terms of language, typology, size, and scope (see Section 5).

The choice of Claim as a microservices detection method represents a tradeoff between internal and external validity. While language-independent for microservices, it relies on Docker as a containerization tool, limiting the sets of analyzable repositories and representing a potential threat to external validity. Since the containerization tool (or even its absence) should not impact significantly Code TD, we can conjecture that the results might be similar considering other containerization tools, *e.g.,* Kubernetes, adopting other

microservices detection methods; however, further studies should be conducted to prove this, although, in the current state of the art, dynamic methods have infeasible execution times for this kind of study.

A minor concern is the decision to consider only single-repository systems, *i.e.,* those contained entirely in one repository, as opposed to multi-repository ones that spread components over multiple repositories. This choice is made to avoid missing the intermediate changes of the minor repositories between one commit and the next in the aggregator repository. However, this aspect should be related only to the size and organization of the system and should not affect its development mode and other factors that could influence TD.

8.4. Reliability

Given the almost purely quantitative nature of the study, our results are highly likely replicable by other researchers. The only exception might be the manual scrutiny conducted to analyze commits with outliers TD values, which constitutes only a minimal fraction. Apart from this limited portion, the study is based entirely on mining and data analysis scripts made available for scrutiny and replication (see Section 1), complemented with raw and final data.

9. Conclusion

In this study, we present a multiple case study to investigate, for the first time, the evolution of Technical Debt in Microservices Architectures. The investigation includes 13 microservice-based systems selected heterogeneously by different aspects, such as language, size, number of microservices, and nature (real cases or industrial demos), totaling over 10k commits. The study is primarily quantitative, based on mining software repositories, source code quality analysis, and statistical data analysis, although some results are complemented with manual qualitative inspection of commits.

The results show that an overall growing trend, particularly accentuated in early development phases, characterizes TD evolution; moreover, the evolution does not present periodic trends. TD variations can happen due to multiple activities, but the more impacting ones are adding/removing a microservice, evolving the business logic, and refactoring when not performed with a TD reduction in mind. Moreover, the extent of TD variations is independent of the number of microservices already present, although the overall TD evolution is strongly correlated with it, meaning that microservices independence is helpful in this aspect.

As concluding remarks, we note that adhering to microservices architecture principles might keep TD compartmentalized within microservices, avoiding its surge and making it more manageable, differently from other types of architectures (*e.g.,* monolithic ones). It is crucial for developers to remain aware of the potential TD they may incur, regardless of the activities they undertake, because even a trivial change, like a function or class refactoring, could significantly impact the TD of systems.

Although adopting a code quality analysis step in the development workflow at the initial stage certainly reduces the development speed, it could allow for the early isolation of the principal causes of TD that will characterize the whole system lifecycle. Without such a precaution, even if it is feasible to maintain a consistent level of TD during the system evolution, an increase in TD may be inevitable as the system grows in size and complexity.

Future Work This study paves the way for a series of in-depth analyses. Several facets have not been addressed yet, which could provide more insights into the phenomenon under investigation. As future work, this study can be complemented by considering the following factors: (i) measurement of the individual contributions of microservices to the TD; (ii) in-depth and more systematic analysis of TD hotspots to understand which kinds of activities expose more developers to increased TD; (iii) assessment of the involvement of each developer in the total TD by measuring the TD they introduce or pay off during the implementation of microservices to evaluate if the TD is evenly distributed or not; (iv) evaluation of different types of TD, other than Code TD, can be measured, especially ATD, which could give inspiring insights into the development of MSAs (*via* the tools mentioned in Section 3, *i.e.,* Arcan [6] and ATDx [50]).

CRediT authorship contribution statement

Kevin Maggi: Conceptualization, Methodology, Software, Formal Analysis, Investigation, Data Curation, Writing - Original Draft, Visualization. **Roberto Verdecchia**: Conceptualization, Writing - Review & Editing, Supervision. **Leonardo Scommegna**: Investigation, Writing - Review & Editing. **Enrico Vicario** Supervision, Writing - Review & Editing.

Declaration of competing interest

None of the authors have declared that they have known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Akaike, H., 1973. Information theory and an extension of the maximum likelihood principle, in: Second International Symposium on Information Theory, Akademia Kiado. pp. 267–281.

[2] Alenezi, M., 2016. Software Architecture Quality Measurement Stability and Understandability. International Journal of Advanced Computer Science and Applications 7. doi:10.14569/IJACSA.2016.070775.

[3] Alves, N.S.R., Mendes, T.S., de Mendonça, M.G., Spínola, R.O., Shull, F., Seaman, C., 2016. Identification and management of technical debt: A systematic mapping study. Information and Software Technology 70, 100–121. doi:10.1016/j.infsof.2015.10.008.

[4] Alves, N.S.R., Ribeiro, L.F., Caires, V., Mendes, T.S., Spínola, R.O., 2014. Towards an Ontology of Terms on Technical Debt, in: 2014 Sixth International Workshop on Managing Technical Debt, IEEE Computer Society. pp. 1–7. doi:10.1109/MTD.2014.9.

[5] Amanatidis, T., Mittas, N., Moschou, A., Chatzigeorgiou, A., Ampatzoglou, A., Angelis, L., 2020. Evaluating the Agreement among Technical Debt Measurement Tools: Building an Empirical Benchmark of Technical Debt Liabilities. Empirical Software Engineering 25, 4161–4204. doi:10.1007/s10664-020-09869-w.

[6] Arcelli Fontana, F., Pigazzini, I., Roveda, R., Tamburri, D., Zanoni, M., Di Nitto, E., 2017. Arcan: A Tool for Architectural Smells Detection, in: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), IEEE Computer Society. pp. 282–285. doi:10.1109/ICSAW.2017.16.

[7] Arcelli Fontana, F., Pigazzini, I., Roveda, R., Zanoni, M., 2016a. Automatic Detection of Instability Architectural Smells, in: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE Computer Society. pp. 433–437. doi:10.1109/ICSME.2016.33.

[8] Arcelli Fontana, F., Roveda, R., Zanoni, M., 2016b. Technical Debt Indexes Provided by Tools: A Preliminary Discussion, in: 2016 IEEE 8th International Workshop on Managing Technical Debt (MTD), IEEE Computer Society. pp. 28–31. doi:10.1109/MTD.2016.11.

[9] Assunção, W.K.G., Krüger, J., Mosser, S., Selaoui, S., 2023. How do microservices evolve? An empirical analysis of changes in open-source microservice repositories. Journal of Systems and Software 204, 111788. doi:10.1016/j.jss.2023.111788.

[10] Avgeriou, P., Kruchten, P., Ozkaya, I., Seaman, C., 2016. Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). Dagstuhl Reports 6, 110–138. doi:10.4230/DagRep.6.4.110.

[11] Avgeriou, P.C., Taibi, D., Ampatzoglou, A., Arcelli Fontana, F., Besker, T., Chatzigeorgiou, A., Lenarduzzi, V., Martini, A., Moschou, A., Pigazzini, I., Saarimaki, N., Sas, D.D., de Toledo, S.S., Tsintzira, A.A., 2021. An Overview and Comparison of Technical Debt Measurement Tools. IEEE Software 38, 61–71. doi:10.1109/MS.2020.3024958.

[12] Bacchiega, P., Pigazzini, I., Arcelli Fontana, F., 2022. Microservices smell detection through dynamic analysis, in: 2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE Computer Society. pp. 290–293. doi:10.1109/SEAA56994.2022.00052.

[13] Basili, V.R., Caldiera, G., Rombach, D., 1994. The Goal Question Metric Approach, in: Encyclopedia of software engineering. Wiley. volume 1, pp. 528–532. doi:10.1002/0471028959.sof142.

[14] Besker, T., Martini, A., Bosch, J., 2016. A Systematic Literature Review and a Unified Model of ATD, in: 2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE Computer Society. pp. 189–197. doi:10.1109/SEAA.2016.42.

[15] Bogner, J., Fritzsch, J., Wagner, S., Zimmermann, A., 2018. 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), Association for Computing Machinery. pp. 125–133. doi:10.1145/3194164.3194166.

[16] Bogner, J., Fritzsch, J., Wagner, S., Zimmermann, A., 2019a. Assuring the Evolvability of Microservices: Insights into Industry Practices and Challenges, in: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE Computer Society. pp. 546–556. doi:10.1109/ICSME.2019.00089.

[17] Bogner, J., Fritzsch, J., Wagner, S., Zimmermann, A., 2019b. Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality, in: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), IEEE Computer Society. pp. 187–195. doi:10.1109/ICSA-C.2019.00041.

[18] Brogi, A., Neri, D., Soldani, J., 2020. Freshening the Air in Microservices: Resolving Architectural Smells via Refactoring, in: Service-Oriented Computing – ICSOC 2019 Workshops, Springer International Publishing. pp. 17–29. doi:10.1007/978-3-030-45989-5_2.

[19] Capilla, R., Arcelli Fontana, F., Mikkonen, T., Bacchiega, P., Salamanca, V., 2023. Detecting Architecture Debt in Micro-Service Open-Source Projects, in: 2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE Computer Society. pp. 394–401. doi:10.1109/SEAA60479.2023.00066.

[20] Cleveland, W.S., Devlin, S.J., 1988. Locally Weighted Regression: An Approach to Regression Analysis by Local Fitting. Journal of the American Statistical Association 83, 596–610. doi:10.1080/01621459.1988.10478639.

[21] Cotroneo, D., Natella, R., Pietrantuono, R., Russo, S., 2014. A survey of software aging and rejuvenation studies. J. Emerg. Technol. Comput. Syst. 10. doi:10.1145/2539117.

[22] De Almeida, R.R., Treude, C., Kulesza, U., 2023. What's behind tight deadlines? Business causes of technical debt, in: 2023 IEEE/ACM 16th International Conference on Cooperative and Human Aspects of Software Engineering (CHASE), pp. 25–30. doi:10.1109/CHASE58964.2023.00011.

[23] Dean, R., Dunsmuir, W., 2015. 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 48. doi:10.3758/s13428-015-0611-2.

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

[25] Di Francesco, P., Lago, P., Malavolta, I., 2019. Architecting with microservices: A systematic mapping study. Journal of Systems and Software 150, 77–97. doi:10.1016/j.jss.2019.01.001.

[26] Diaz-Pace, J.A., Tommasel, A., Pigazzini, I., Arcelli Fontana, F., 2020. Sen4Smells: A Tool for Ranking Sensitive Smells for an Architecture Debt Index, in: 2020 IEEE Congreso Bienal de Argentina (ARGENCON), IEEE Computer Society. pp. 1–7. doi:10.1109/ARGENCON49523.2020.9505535.

[27] Dickey, D., Fuller, W., 1979. Distribution of the Estimators for Autoregressive Time Series With a Unit Root. JASA. Journal of the American Statistical Association 74, 427–431. doi:10.2307/2286348.

[28] Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L., 2017. Microservices: Yesterday, Today, and Tomorrow, in: Mazzara, M., Meyer, B. (Eds.), Present and Ulterior Software Engineering. Springer International Publishing, pp. 195–216. doi:10.1007/978-3-319-67425-4_12.

[29] Evans, E., 2004. Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional.

[30] Freire, S., Rios, N., Pérez, B., Castellanos, C., Correal, D., Ramač, R., Mandić, V., Taušan, N., López, G., Pacheco, A., Mendonça, M., Falessi, D., Izurieta, C., Seaman, C., Spínola, R., 2023. Software practitioners' point of view on technical debt payment. Journal of Systems and Software 196, 111554. doi:10.1016/j.jss.2022.111554.

[31] Gama, E., Freire, S., Mendonça, M., Spínola, R.O., Paixao, M., Cortés, M.I., 2020. Using Stack Overflow to Assess Technical Debt Identification on Software Projects, in: Proceedings of the XXXIV Brazilian Symposium on Software Engineering, Association for Computing Machinery. p. 730–739. doi:10.1145/3422392.3422429.

[32] Gezici, B., Tarhan, A., Chouseinoglou, O., 2019. Internal and external quality in the evolution of mobile software: An exploratory study in open-source market. Information and Software Technology 112, 178–200. doi:10.1016/j.infsof.2019.04.002.

[33] Goulão, M., Fonte, N., Wermelinger, M., Brito e Abreu, F., 2012. Software Evolution Prediction Using Seasonal Time Analysis: A Comparative Study, in: 2012 16th European Conference on Software Maintenance and Reengineering, pp. 213–222. doi:10.1109/CSMR.2012.30.

[34] Granger, C.W., 1969. Investigating Causal Relations by Econometric Models and Cross-spectral Methods. Econometrica: journal of the Econometric Society , 424–438doi:10.2307/1912791.

[35] Kendall, M.G., 1948. Rank correlation methods. Griffin.

[36] Kovalenko, V., Palomba, F., Bacchelli, A., 2018. Mining File Histories: Should We Consider Branches?, in: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), Association for Computing Machinery. pp. 202–213. doi:10.1145/3238147.3238169.

[37] Kozanidis, N., Verdecchia, R., Guzman, E., 2022. 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. p. 45–56. doi:10.1145/3544902.3546245.

[38] Lenarduzzi, V., Lomio, F., Saarimäki, N., Taibi, D., 2020. Does migrating a monolithic system to microservices decrease the technical debt? Journal of Systems and Software 169, 110710. doi:10.1016/j.jss.2020.110710.

[39] Letouzey, J.L., 2012. The SQALE method for evaluating Technical Debt, in: 2012 Third International Workshop on Managing Technical Debt (MTD), IEEE Computer Society. pp. 31–36. doi:10.1109/MTD.2012.6225997.

[40] Li, Z., Avgeriou, P., Liang, P., 2015a. A systematic mapping study on technical debt and its management. Journal of Systems and Software 101, 193–220. doi:10.1016/j.jss.2014.12.027.

[41] Li, Z., Avgeriou, P., Liang, P., 2015b. A systematic mapping study on technical debt and its management. Journal of Systems and Software 101, 193–220. doi:10.1016/j.jss.2014.12.027.

[42] Liu, G., Huang, B., Liang, Z., Qin, M., Zhou, H., Li, Z., 2020. Microservices: architecture, container, and challenges, in: 2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C), IEEE Computer Society. pp. 629–635. doi:10.1109/QRS-C51114.2020.00107.

[43] M. Bomfim, M., A. Santos, V., 2017. Strategies for Reducing Technical Debt in Agile Teams, in: Agile Methods: 7th Brazilian Workshop, WBMA 2016, Curitiba, Brazil, November 7-9, 2016, Revised Selected Papers 7, Springer International Publishing. pp. 60–71. doi:10.1007/978-3-319-55907-0_6.

[44] Maggi, K., Verdecchia, R., Scommegna, L., Vicario, E., 2024. CLAIM: a Lightweight Approach to Identify Microservices in Dockerized Environments, in: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, Association for Computing Machinery. pp. 357–362. doi:10.1145/3661167.3661206.
[45] Mann, H.B., 1945. Nonparametric Tests Against Trend. Econometrica 13, 245–259.
[46] Martin, R., 1995. Object oriented design quality metrics: An analysis of dependencies. ROAD 2.
[47] Menezes, G., Braga, W., Fontão, A., Hora, A., Cafeo, B., 2022. Assessing the Impact of Code Samples Evolution on Developers' Questions, in: Proceedings of the XXXVI Brazilian Symposium on Software Engineering, Association for Computing Machinery. p. 321–330. doi:10.1145/3555228.3555250.
[48] Mumtaz, H., Singh, P., Blincoe, K., 2022. Analyzing the Relationship between Community and Design Smells in Open-Source Software Projects: An Empirical Study, in: Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Association for Computing Machinery. p. 23–33. doi:10.1145/3544902.3546249.
[49] Ollech, D., Webel, K., 2020. A random forest-based approach to identifying the most informative seasonality tests. Discussion Papers 55. Deutsche Bundesbank. doi:10.2139/ssrn.3721055.
[50] Ospina, S., Verdecchia, R., Malavolta, I., Lago, P., 2021. ATDx: A tool for Providing a Data-driven Overview of Architectural Technical Debt in Software-intensive Systems, in: ECSA-C 2021 Companion Proceedings of the 15th European Conference on Software Architecture, CEUR-WS.
[51] Pigazzini, I., Arcelli Fontana, F., Lenarduzzi, V., Taibi, D., 2020. Towards microservice smells detection, in: Proceedings of the 3rd International Conference on Technical Debt, Association for Computing Machinery. p. 92–97. doi:10.1145/3387906.3388625.
[52] Pigazzini, I., Foppiani, D., Arcelli Fontana, F., 2021. Two Different Facets of Architectural Smells Criticality: An Empirical Study, in: Companion Proceedings of the 15th European Conference on Software Architecture, CEUR-C.
[53] Rahman, M.I., Taibi, D., 2019. A Curated Dataset of Microservices-Based Systems, in: Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution, CEUR-WS.
[54] Ramač, R., Mandić, V., Taušan, N., Rios, N., Freire, S., Pérez, B., Castellanos, C., Correal, D., Pacheco, A., Lopez, G., Izurieta, C., Seaman, C., Spinola, R., 2022. Prevalence, common causes and effects of technical debt: Results from a family of surveys with the IT industry. Journal of Systems and Software 184, 111114. doi:10.1016/j.jss.2021.111114.
[55] Richardson, C., 2018. Microservices Patterns. Manning Publications.
[56] Rios, N., de Mendonça Neto, M.G., Spínola, R.O., 2018. A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. Information and Software Technology 102, 117–145. doi:10.1016/j.infsof.2018.05.010.
[57] Roveda, R., Arcelli Fontana, F., Pigazzini, I., Zanoni, M., 2018. Towards an Architectural Debt Index, in: 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE Computer Society. pp. 408–416. doi:10.1109/SEAA.2018.00073.
[58] Runeson, P., Höst, M., 2009. Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering 14, 131–164. doi:10.1007/s10664-008-9102-8.
[59] Santos, N., Salgado, C.E., Morais, F., Melo, M., Silva, S., Martins, R., Pereira, M., Rodrigues, H., Machado, R.J., Ferreira, N., Pereira, M., 2019. A logical architecture design method for microservices architectures, in: Proceedings of the 13th European Conference on Software Architecture - Volume 2, Association for Computing Machinery. p. 145–151. doi:10.1145/3344948.3344991.
[60] Sas, D., Avgeriou, P., 2023. An Architectural Technical Debt Index Based on Machine Learning and Architectural Smells. IEEE Transactions on Software Engineering 49, 4169–4195. doi:10.1109/TSE.2023.3286179.
[61] Sas, D., Avgeriou, P., Arcelli Fontana, F., 2019. Investigating Instability Architectural Smells Evolution: An Exploratory Case Study, in: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 557–567. doi:10.1109/ICSME.2019.00090.
[62] Saša Baškarada, V.N., Koronios, A., 2020. Architecting Microservices: Practical Opportunities and Challenges. Journal of Computer Information Systems 60, 428–436. doi:10.1080/08874417.2018.1520056.
[63] Shapiro, S.S., Wilk, M.B., 1965. An analysis of variance test for normality (complete samples). Biometrika 52, 591–611. doi:10.1093/biomet/52.3-4.591.
[64] Soldani, J., Muntoni, G., Neri, D., Brogi, A., 2021. The μTOSCA toolchain: Mining, analyzing, and refactoring microservice-based architectures. Software: Practice and Experience 51, 1591–1621. doi:10.1002/spe.2974.
[65] Soldani, J., Tamburri, D., Heuvel, W.J., 2018. The Pains and Gains of Microservices: A Systematic Grey Literature Review. Journal of Systems and Software 146, 215–232. doi:10.1016/j.jss.2018.09.082.
[66] de Toledo, S.S., Martini, A., Przybyszewska, A., Sjøberg, D.I.K., 2019. Architectural Technical Debt in Microservices: A Case Study in a Large Company, in: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), IEEE Computer Society. pp. 78–87. doi:10.1109/TechDebt.2019.00026.
[67] de Toledo, S.S., Martini, A., Sjøberg, D.I., Przybyszewska, A., Frandsen, J.S., 2021a. Reducing Incidents in Microservices by Repaying Architectural Technical Debt, in: 2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE Computer Society. pp. 196–205. doi:10.1109/SEAA53835.2021.00033.
[68] de Toledo, S.S., Martini, A., Sjøberg, D.I.K., 2021b. Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study. Journal of Systems and Software 177, 110968. doi:10.1016/j.jss.2021.110968.
[69] Velepucha, V., Flores, P., 2023. A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges. IEEE Access 11, 88339–88358. doi:10.1109/ACCESS.2023.3305687.
[70] Verdecchia, R., Kruchten, P., Lago, P., Malavolta, I., 2021. Building and evaluating a theory of architectural technical debt in software-intensive systems. Journal of Systems and Software 176, 110925. doi:10.1016/j.jss.2021.110925.

[71] Verdecchia, R., Lago, P., Malavolta, I., Ozkaya, I., 2020. ATDx: Building an Architectural Technical Debt Index, in: Ali, R., Kaindl, H., Maciaszek, L. (Eds.), ENASE 2020, SciTePress. pp. 531–539. doi:10.5220/0009577805310539.
[72] Verdecchia, R., Maggi, K., Scommegna, L., Vicario, E., 2024. Technical Debt in Microservices: A Mixed-Method Case Study, in: Tekinerdoğan, B., Spalazzese, R., Sözer, H., Bonfanti, S., Weyns, D. (Eds.), Software Architecture. ECSA 2023 Tracks, Workshops, and Doctoral Symposium, Springer Nature Switzerland, Cham, Switzerland. pp. 217–236. doi:10.1007/978-3-031-66326-0_14.
[73] Verdecchia, R., Malavolta, I., Lago, P., 2018. Architectural technical debt identification: the research landscape, in: Proceedings of the 2018 International Conference on Technical Debt, Association for Computing Machinery. p. 11–20. doi:10.1145/3194164.3194176.
[74] Villa, A., Ocharán-Hernández, J.O., Pérez-Arriaga, J.C., Limón, X., 2022. A Systematic Mapping Study on Technical Debt in Microservices, in: 2022 10th International Conference in Software Engineering Research and Innovation (CONISOFT), IEEE Computer Society. pp. 182–191. doi:10.1109/CONISOFT55708.2022.00032.
[75] Wang, Y., Kadiyala, H., Rubin, J., 2021. Promises and challenges of microservices: an exploratory study. Empirical Software Engineering 26, 63. doi:10.1007/s10664-020-09910-y.
[76] Waseem, M., Liang, P., Márquez, G., Salle, A.D., 2020. Testing Microservices Architecture-Based Applications: A Systematic Mapping Study, in: 2020 27th Asia-Pacific Software Engineering Conference (APSEC), IEEE Computer Society. pp. 119–128. doi:10.1109/APSEC51365.2020.00020.
[77] Zampetti, F., Noiseux, C., Antoniol, G., Khomh, F., Di Penta, M., 2017. Recommending when Design Technical Debt Should be Self-Admitted, in: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE Computer Society. pp. 216–226. doi:10.1109/ICSME.2017.44.
[78] Zhong, C., Huang, H., Zhang, H., Li, S., 2022. Impacts, causes, and solutions of architectural smells in microservices: An industrial investigation. Software: Practice and Experience 52, 2574–2597. doi:10.1002/spe.3138.

Appendix A. Additional Data Plots

Appendix A.1. TD Trend

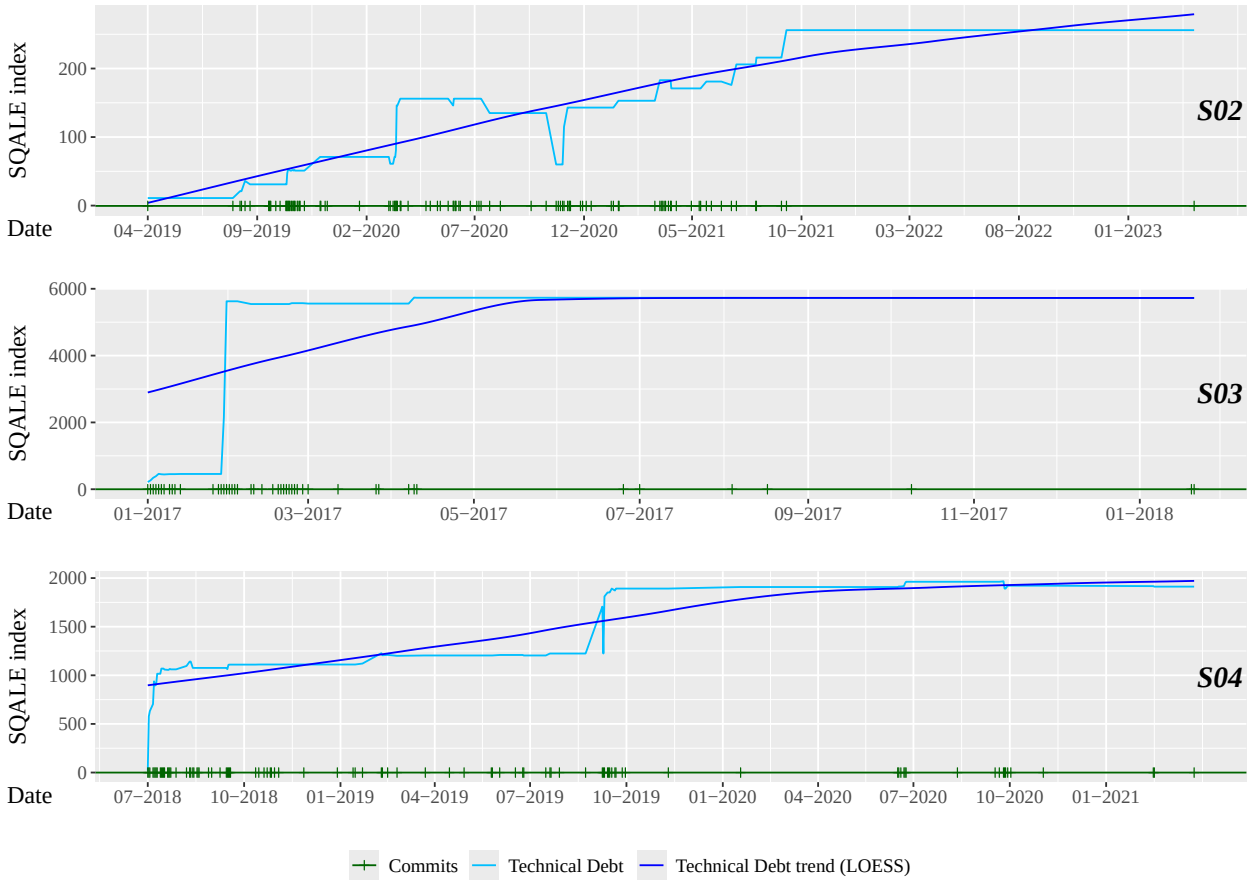

Figure A.8: Trend of TD in systems $S02$, $S03$, and $S04$

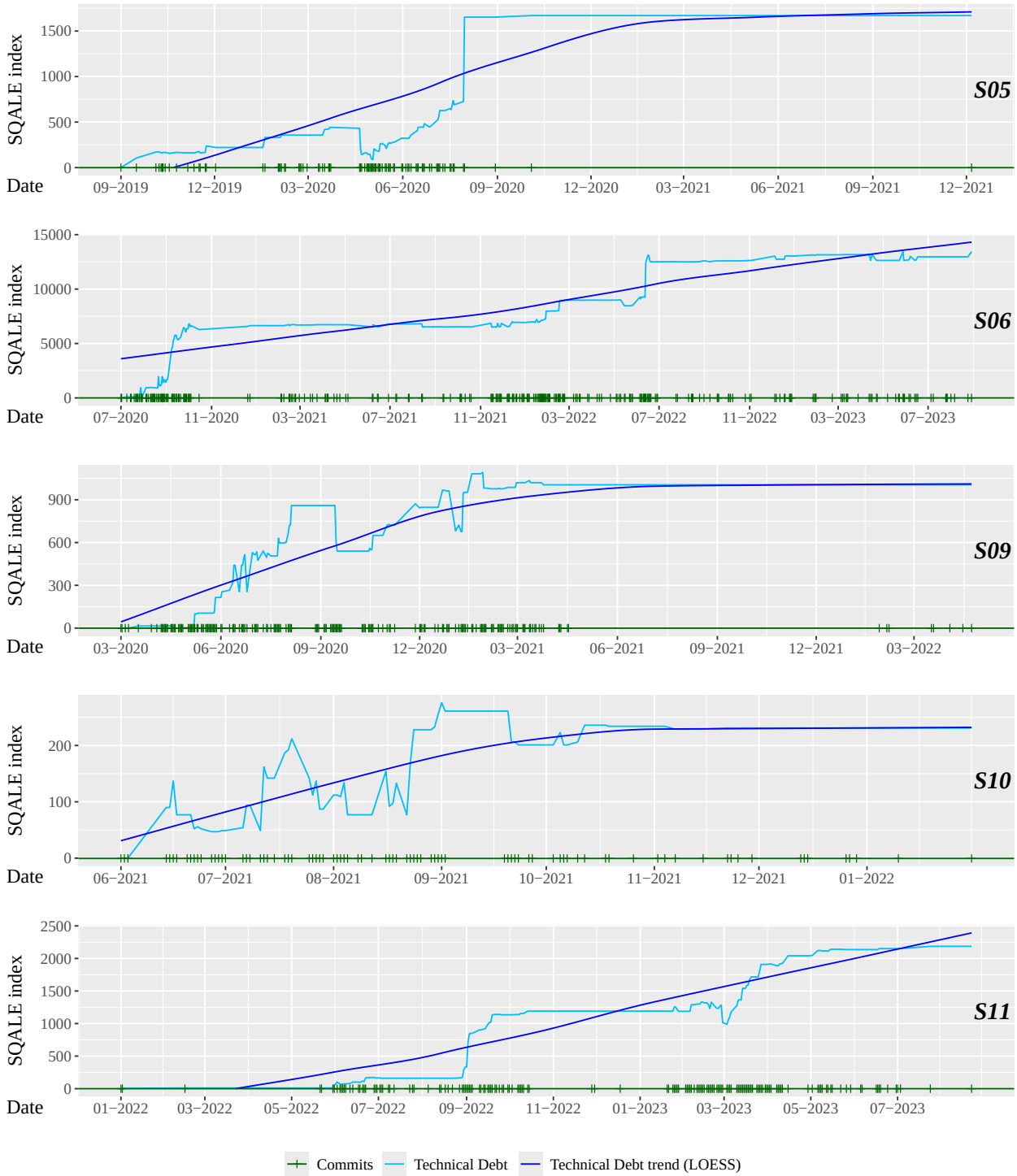

Figure A.9: Trend of TD in systems $S05$, $S06$, $S09$, $S10$ and $S11$

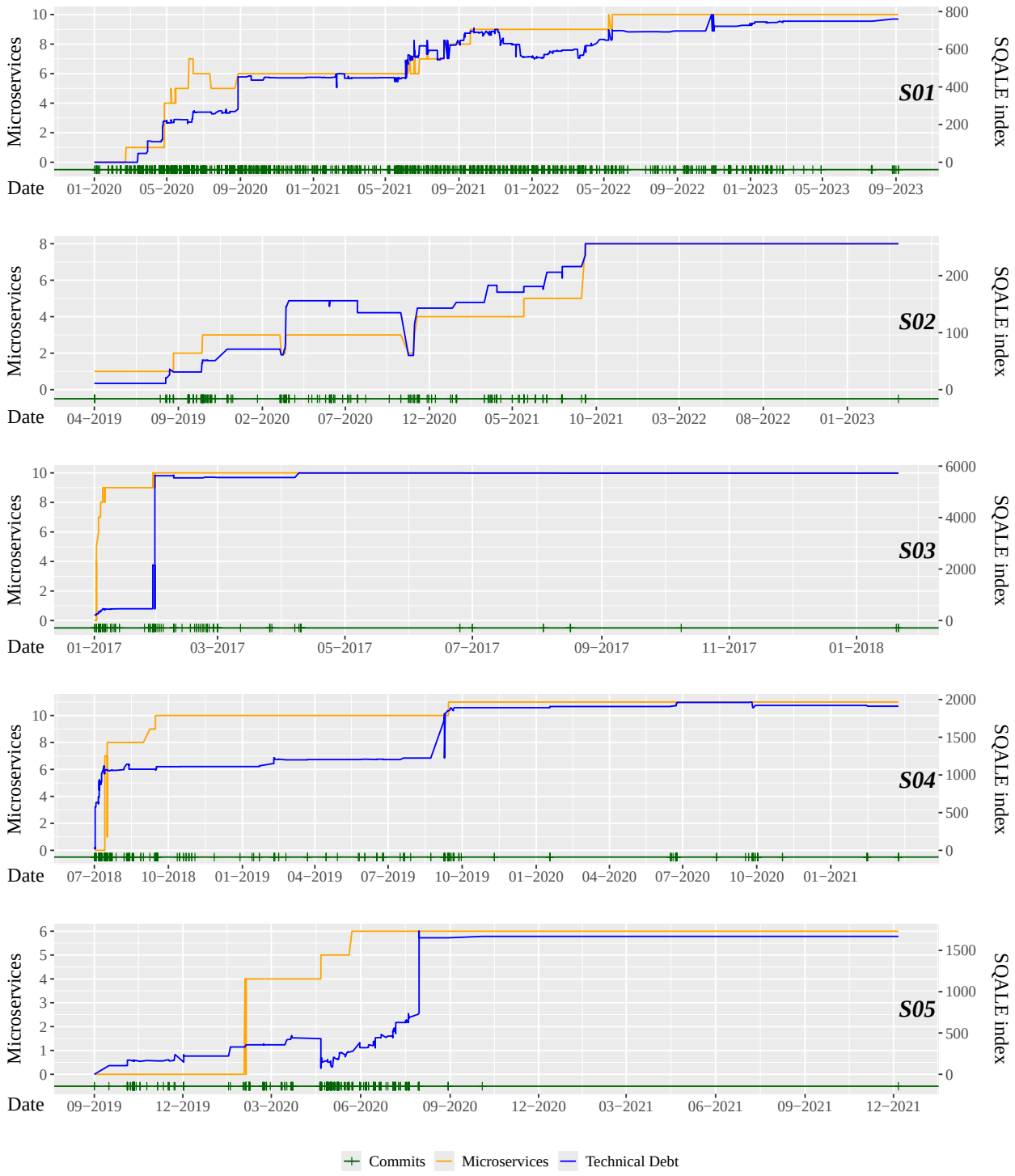

Figure A.10: Evolution of TD and microservices number in systems $S01$, $S02$, $S03$, $S04$ and $S05$

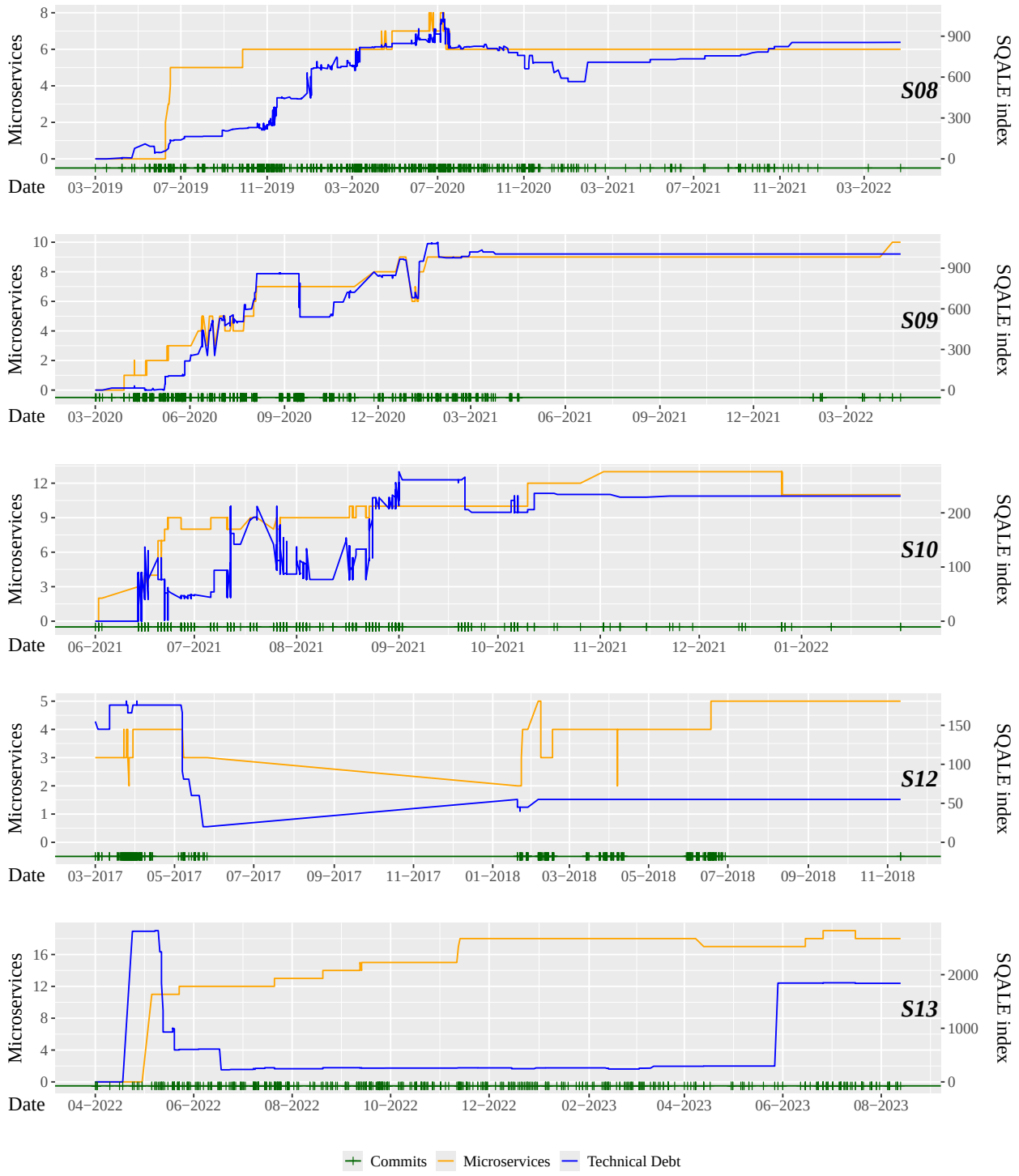

Figure A.11: Evolution of TD and microservices number in systems $S08$, $S09$, $S10$, $S12$ and $S13$

Appendix B. Replicability

A replication package made openly available online (see Section 1). The package contains all scripts and data used by the results presented in this manuscript, supported by a brief set of instruction on how to replicate all research steps executed for this study. For completeness, following the adopted version for the main tools used:

- Python v3.10.6
- SonarScanner CLI v4.8.0.2856
- SonarScanner for Maven v3.9.1
- SonarScanner for .NET v5.13.1
- SonarQube v9.9 LTS