Highlights

How do Microservices Evolve? An Empirical Analysis of Changes in Open-Source Microservice Repositories

Wesley K. G. Assunção,Jacob Krüger,Sébastien Mosser,Sofiane Selaoui

- We analyzed 11 open-source microservice repositories to study their evolution over time.
- The analysis covers $7,319$ commits, and up to seven years of activity. We labeled the associated atomic changes (almost $100,000$) in a semi-automated way, including a cross-validation.
- A lot of studies focus on architectural patterns or smells. Instead, we studied *(i)* how microservice-based architectures evolve (in a quantitative way) and *(ii)* why they evolve (in a qualitative way).
- We identified that, contrarily to the accepted idea that microservices are focused on business logic, their evolution in an open-source context is driven by technicalities.
- We studied how microservices co-evolve together, and identified that the evolution of each microservice in isolation from the others is not the main tendency for open-source systems.

How do Microservices Evolve? An Empirical Analysis of Changes in Open-Source Microservice Repositories

Wesley K. G. Assunção[a,b], Jacob Krüger[c], Sébastien Mosser[d,e] and Sofiane Selaoui[f]

[a]North Carolina State University, USA
[b]Johannes Kepler University Linz, Austria
[c]Eindhoven University of Technology, The Netherlands
[d]McMaster University, Canada
[e]McMaster Centre for Software Certification, Canada
[f]Université du Québec à Montréal, Canada

ARTICLE INFO

Keywords:
microservices
service-oriented architecture
software evolution
software architecture
mining software repositories

ABSTRACT

Context. Microservice architectures are an emergent service-oriented paradigm widely used in industry to develop and deploy scalable software systems. The underlying idea is to design highly independent services that implement small units of functionality and can interact with each other through lightweight interfaces. **Objective.** Even though microservices are often used with success, their design and maintenance pose novel challenges to software engineers. In particular, it is questionable whether the intended independence of microservices can actually be achieved in practice. **Method.** So, it is important to understand how and why microservices evolve during a system's life-cycle, for instance, to scope refactorings and improvements of a system's architecture or to develop supporting tools. To provide insights into how microservices evolve, we report a large-scale empirical study on the (co-)evolution of microservices in 11 open-source systems, involving quantitative and qualitative analyses of 7,319 commits. **Findings.** Our quantitative results show that there are recurring patterns of (co-)evolution across all systems, for instance, "shotgun surgery" commits and microservices that are largely independent, evolve in tuples, or are evolved in almost all changes. We refine our results by analyzing service-evolving commits qualitatively to explore the (in-)dependence of microservices and the causes for their specific evolution. **Conclusion.** The contributions in this article provide an understanding for practitioners and researchers on how microservices evolve in what way, and how microservice-based systems may be improved.

1. Introduction

Microservices are a concept for designing service-oriented software systems with a focus on small, independent, and replaceable units of functionality (Familiar, 2015; Newman, 2015; Thönes, 2015). Ideally, due to a high degree of independence and light coupling, microservices can be freely combined and can support various technologies (e.g., multiple databases, frameworks, and programming languages) within the same heterogeneous architecture (Fowler, 2016; Yuan, 2019). Such properties promise benefits, such as software reuse, improved availability, or continuous delivery (Luz et al., 2018; Taibi et al., 2017), and have led to various organizations successfully adopting microservices, for instance, Netfilx and Uber (Fowler, 2016; Vučković, 2020). However, microservice-based systems are not feasible for every organization, and the potential pitfalls (e.g., managing knowledge about the various technologies and dependencies between microservices) may prevent a successful adoption (Viggiato et al., 2018; Vučković, 2020).

wguezas@ncsu.edu (W.K.G. Assunção); j.kruger@tue.nl (J. Krüger); mossers@mcmaster.ca (S. Mosser); selaoui.sofiane@courrier.uqam.ca (S. Selaoui)
https://wesleyklewerton.github.io/ (W.K.G. Assunção); https://jacobkrueger.github.io/ (J. Krüger); http://mosser.github.io (S. Mosser)
ORCID(s): 0000-0002-7557-9091 (W.K.G. Assunção); 0000-0002-0283-248X (J. Krüger)

Microservices have originated from industry, and only recently researchers started to explore them (Joseph and Chandrasekaran, 2019). For instance, researchers have proposed methodologies for migrating systems towards microservices (Kecskemeti et al., 2016; Wolfart et al., 2021), developed tools for benchmarking (Sriraman and Wenisch, 2018) and identifying microservices (Assunção et al., 2022; Assunção et al., 2021; Carvalho et al., 2020), studied the pros and cons of microservices (Bogner et al., 2019; Viggiato et al., 2018; Vučković, 2020), or explored relations to other research fields, such as product-line engineering (Assunção et al., 2021, 2020; Benni et al., 2020; Krüger et al., 2022). In such works, maintaining and evolving microservices has been identified as a highly important challenge. Specifically, the high independence of microservices is intended to allow developers to separately evolve individual microservices. However, the technological heterogeneity and potential design misconceptions have caused concerns regarding this assumption—resulting in the perception of co-evolving microservices. For example, dependencies may be caused by the additional interfaces required to integrate heterogeneous technologies (e.g., breaking APIs (Zdun et al., 2020)), intertwined microservices that address similar functionality (e.g., not enough separation of concerns (Waseem et al., 2021)), or microservices heavily building on each other (e.g., feature interactions (Benni et al., 2020)). The consequent dependencies complicate the evolution and maintenance of

microservice-based systems, which can cause unnecessary costs, bugs, and design flaws (Bogner et al., 2019; Sampaio Jr. et al., 2017; Vučković, 2020).

In this article, we report an empirical study of how such dependencies manifest during the evolution of microservices, essentially asking: **How do microservices evolve?** For this purpose, we analyzed 7,319 commits of 11 open-source microservice-based systems. Particularly, we distinguished between different types of evolution (i.e., technological, services, miscellaneous) and recurring patterns (e.g., independent evolution, evolution in tuples) based on a quantitative analysis. The results of this analysis show that several microservices typically co-evolve with each other. However, we also observed that changes are commonly unrelated to a system's microservices that represent business logic (i.e., they primarily relate to technological or miscellaneous changes). To understand the reasons for microservice co-evolution, we performed a qualitative analysis on a subset of the commits to manually identify the respective causes.

In detail, we contribute to the following in this work:

- We report a quantitative analysis of *how* microservices (co-)evolve, separating different types of evolution by identifying recurring patterns.
- We describe a qualitative analysis to understand *why* microservice (co-)evolve along the patterns identified.
- We discuss the implications of our findings on the established notion of independent microservices.
- We publish our material in an open-access repository.[1]

Our insights help understand, assess, and potentially resolve unintended or expensive co-evolution of microservices, for instance, by refactoring and improving a system's architecture. Additionally, our findings can help researchers understand the evolution of microservice-based systems, which can guide the design of new techniques for supporting developers.

The remainder of this article is structured as follows. In Section 2, we describe the methodology we employed for our empirical study. We present and discuss the results to our research questions in Section 3. Then, we discuss threats to the validity of our study in Section 4. In Section 5, we describe the related work for this article. Finally, we conclude and present our lessons learned in Section 6.

2. Methodology

To address our research goal of understanding microservice evolution, we decided to conduct a *multi-case study* (Yin, 2018). Thus, we analyzed, in a homogeneous way, a number of microservice-based systems to obtain in-depth insights, while also improving on a single case study. In this section, we explain the design of our study. We display an overview of our methodology in Figure 1.

[1] Submitted as supplementary material, will be put on Zenodo.

2.1. Research Questions

When studying microservice architectures, state-of-the-art research focuses on aspects such as (i) migrating to microservices (Kecskemeti et al., 2016; Wolfart et al., 2021), (ii) structural and dynamic dependencies (Esparrachiari et al., 2018), or (iii) smells and patterns (Neri et al., 2019; Taibi et al., 2020). In contrast, we explore the evolution of microservices in open-source systems to understand whether and how these may co-evolve, complementing such research. To execute our multi-case study, we leveraged version-control data by analyzing *commits* as well as *atomic changes*. For example, a commit can comprise changes to multiple files, each of which we consider an individual atomic change (see Section 2.3 for more details).

We used this data to tackle two research questions:

RQ$_1$ *How do microservice-based systems evolve?*

First, we studied how microservices have evolved in our 11 open-source subject systems. To this end, we manually labeled (cf. Section 2.4 for more detailed definitions) the atomic changes of each system to distinguish whether the evolution was technical (e.g., library updates), service-driven (e.g., updates on a microservice's functionality), or due to something system specific that we classified as miscellaneous (e.g., updating icons). Using a quantitative analysis, we identified four recurring observations that help understand and manage the evolution of microservices.

RQ$_2$ *Why do microservice-based systems evolve in that way?*

Second, we aimed to understand the causes for our observations. To this end, we used a qualitative analysis in which we studied relevant commits (e.g., code changes, commit messages, issues, and pull-requests) and the systems' documentations (e.g., documented architectures). For instance, we identified technical issues (e.g., dependencies between microservices) and development practices (e.g., tangled commits) to cause some of our observations.

Based on our findings, we discuss implications for the evolution of microservice-based systems, and for the assumption of independence between individual microservices.

2.2. Subject Systems

Since microservices are intended to represent the business logic of a system, there are not many architectures available outside of industrial settings in which non-disclosure agreements are required. In parallel, due to the popularity of the microservice paradigm, searching for open-source projects "in the wild" returns thousands of toy examples. To mitigate these problems of selecting suitable subject systems, we inspected all projects listed on a curated GitHub list[2] of microservice-based systems (Rahman et al., 2019). This list is becoming a *de facto* standard in research related to analyzing open-source microservice architectures. Using such a curated

[2] https://github.com/davidetaibi/Microservices_Project_List

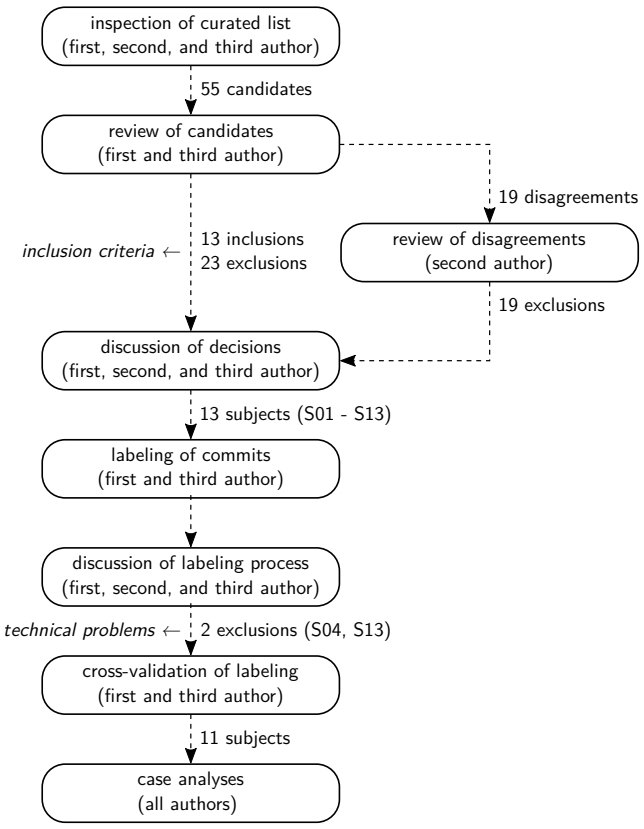

Figure 1: Overview of our multi-case study.

list instead of mining open-source repositories helped us restrict the list of relevant candidates to a manageable size. Moreover, evaluating whether a certain system is feasible for inclusion is a time-consuming task (e.g., ensuring the domain and quality), which is why using a list of projects already known, used, and considered feasible by the community is more efficient and reliable.

Inspecting the list led to a total of 55 candidate systems for our study. The first and third authors independently reviewed each system and decided whether it was a feasible subject for our study or not. Concretely, we defined the following three inclusion criteria (ICs):

IC_1 The system's developers must provide additional information in the form of pull-requests, issues, and discussions to allow us to understand the specifics of the system's evolution, for instance, whether a change was performed to improve the performance of a microservice or to update underlying frameworks and technologies.

IC_2 The system must have a feasible size in terms of microservices and commits to analyze evolution, excluding toy, exemplary, or idealized best-practice examples (e.g., those with only a single service). As each system uses different technologies and languages, this criterion cannot be quantified in an absolute way. We relied on our own development expertise to qualify whether a system could be manually analyzed in fewer than three working days, while still exhibiting microservice evolution.

IC_3 The system must have received updates at least until 2018 to reflect on recent microservice evolution. Note that we cloned the systems with all of their branches (which we also analyzed) in the beginning of 2021.

The first and third authors agreed that 13 systems fulfilled our criteria, while 23 did not.

Regarding 19 of the 55 systems, the two authors had opposing opinions (4) or were both unsure whether to include them (15), for instance, because the authors perceived the number of commits as too small (IC_2) or the additional documentation as too limited (IC_1). For these 19 cases, the second author performed an independent review, which recommended the exclusion of all of these cases. We derived a final decision for all 55 projects during a discussion among the three authors in which especially the second author justified why the 19 systems should be excluded. Note again that those were the cases for which at least one of the other two authors was tending towards exclusion already (with one exception tending towards inclusion before), and we agreed that the concerns were justified (e.g., that the additional documentation did not provide information on the microservices, but only on libraries or frameworks used).

In Table 1, we present the 13 systems we considered as potentially feasible for our multi-case study. These were maintained for a reasonable time (IC_3), had a size that is comparable to systems used in other empirical studies on microservices architectures (IC_2), and contain enough documentation to support the identification of the different services and infrastructure technologies in the source code (IC_1). We then continued with preparatory analyses to gain a first understanding of the 13 systems.

During these preparatory analyses (i.e., commit labeling as we describe in Section 2.4), we identified a few issues with two of the systems:

- S04 is larger than all other systems (3,917 commits; 113,418 atomic changes), and represents by itself 50 % of the dataset. This posed technical challenges (e.g., manually analyzing all changes in spreadsheets) and would introduce either imbalance or sampling bias into our dataset.

- S13 has been a multi-repository project in which each service resides within an own repository, and thus was unsuitable for our analysis workflow. Precisely, changes to the microservices are split into the different repositories, which means that the evolution looks completely independent, even though it may not be (i.e., co-evolution is not represented by atomic changes being tangled in commits and requires extensive expert knowledge about the system to identify).

In a consequent discussion of the labeling process, we decided to exclude these two systems (we list them in Table 1 for completeness and consistency). The rationale behind these two exclusions differs. We excluded S13, because of its multi-repository structure preventing the identification of any co-evolution in the version-control system—making it unsuitable

for our research methodology. In contrast, we excluded S04 to support a homogeneous study of the systems. Analyzing such a system would require to sample the change dataset to conduct a manual labeling with reasonable effort. Doing so, we would have to also sample all other systems following the same sampling methodology. As a result, and after having consulted experts in research data management, we preferred to provide a homogeneous and complete analysis of the remaining 11 systems, instead of analyzing a sampled dataset. It is also important to note that sampling can introduce bias when datasets are unbalanced, which is the case in this study.

Finally, we continued with the 11 systems we summarize in Table 1 into our actual study. We can see in Table 1 that the systems are highly diverse regarding their sizes, programming languages, number of contributors, and the number of commits we could analyze; spanning a period from 2014 until early 2021. Note that main languages refers to those programming languages (excluding HTML, CSS, etc.) that had contributed to more than 5 % to a system according to GitHub's statistics when we extracted the data. Furthermore, we included only a single academic system that provides a benchmark (S01), while the other systems have been developed by industry—or at least by practitioners (i.e., the repository is owned by a developer, not a company or institution). One of the systems (S02) is a real-world product of an organization, while most systems serve as reference systems. Thus, they provide a reference architecture for demonstrating how to implement certain systems (e.g., web-shops) based on microservice technologies. While potentially not ideal (i.e., they do not represent actual products, but these are rarely available), these systems have been developed by practitioners and are intended to guide the development of microservice-based systems.

Most systems are small to medium-sized in terms of lines of code (between ≈1,000 and ≈400,000) as well as microservices (between four to 16). However, if we identify problems with co-evolution already for such smaller and more manageable systems, we argue that it is likely that similar patterns occur for large systems with hundreds or thousands of developers and microservices. Consequently, we argue that these systems are suitable for our multi-case study. Additionally, the selected systems also involve authority projects (the academic and two reference systems), namely by the Descartes group (S01), Google (S05), and Microsoft (S11). Overall, we argue that our subject systems represent a diverse and appropriate sample to understand the evolution of real microservice-based systems.

2.3. Formalizing Changes

Version-control systems are designed to record and track the different versions of a given piece of software. Several formalisms exist to model how such versions are reified and managed (Ananieva et al., 2022a,b, 2020; Conradi and Westfechtel, 1998; Hindle and German, 2005). Industrial systems like CVS,[3] SVN,[4] or Git[5] follow an operational description method, by documenting their internal structure (Chacon and Straub, 2014). Other systems like Mercurial,[6] Darcs,[7] or more recently Pijul,[8] provide a fully-fledged formal representation. The most known one is the *patch theory* popularized by Darcs (Jacobson, 2009). In this model, software is considered as a sequence of patches, applied sequentially to build a given version. Denoting R_n a given revision of a software and $P = [p_1, \ldots, p_n]$ the sequence of patches recorded to build it, we can rebuild any intermediate release R_i of R_n by applying a sub-sequence of P (with $i < n$). This ability is essential when *mining* software history, as it gives access to each step used to develop the system.

A common point to all of these models is that they support the extraction of information regarding how a given software was written. We rely on Git, which is one of the most commonly used version-control systems at the time of writing. So, for the sake of conciseness, we rely on Git vocabulary[9] only. It is important to note that these concepts can be translated to any version-control systems. Using Git, developers have access to versioning operations at the file system level, such as adding or removing files from the versioning history. They can assemble all their changes (e.g., a new file was added, an old file was deleted, or some modification were made to an already versioned file) in a *commit*, which records a consistent set of *atomic* changes into the version-control system, in a transactional way. In a nutshell, a commit C_n records the status of the system at a given time t_n, storing all the changes made to the system between t_{n-1} and t_n.

As a consequence, from a descriptive point of view, we can consider a change as a pair (*file*, *modifier*), where *file* is a reference to a given file (e.g., a path) and *modifier* is one of the modification type recognized by the version-control system (e.g., add, delete, or modify). A commit C is defined as a triple binding together an unordered collection of independent and *atomic* changes $\{c_1, \ldots, c_n\}$, an author, and a timestamp. Formal models like the patch theory also include operational semantics to formalize how changes are applied to concretely build a piece of software. However, considering that this article is about observing which changes were made while developing microservices, we stay at the description level.

2.4. Commit Labeling

Before our actual case analyses, the first and third authors independently labeled the commits C_i of each system by labeling each atomic change $\{c_{i_1}, \ldots, c_{i_n}\}$ stored in these commits. For this purpose, they manually analyzed commit messages, code changes, as well as all associated documentation (e.g., pull requests or issues) to understand the evolution of each system. This step was essential to understand the architecture

[3] https://savannah.nongnu.org/projects/cvs
[4] https://subversion.apache.org/
[5] https://git-scm.com/
[6] https://www.mercurial-scm.org/
[7] http://darcs.net/
[8] https://pijul.org/
[9] https://git-scm.com/docs/gitglossary

Table 1
Overview of the 11 subject systems we included in our multi-case study.

id	origin	type	main languages	ms	loc	cont	commits			
							#	from	until	period
S01	Academia	Benchmark	Java	9	31,740	17	1,586	47e8fa7	a794fd0	2017-08-18 – 2021-02-16
	https://github.com/DescartesResearch/TeaStore									
S02	Industry	Product	Java	15	40,509	24	2,750	96a4296	8c875ee	2014-01-06 - 2021-02-12
	https://github.com/sitewhere/sitewhere									
S03	Industry	Reference	Java, JavaScript	9	14,228	17	265	f069222	b12d05b	2016-11-12 – 2021-02-28
	https://github.com/spring-petclinic/spring-petclinic-microservices									
~~S04~~	*<dropped during commit labeling>*									
	https://github.com/dotnet-architecture/eShopOnContainers									
S05	Industry	Reference	Python, Go, C#	12	29,223	81	563	2fd4967	3a1850d	2018-06-13 – 2021-02-19
	https://github.com/GoogleCloudPlatform/microservices-demo									
S06	Industry	Reference	Shell (C#, Java, TypeScript)	8	42,174	28	555	b7e35ba	cba9dcb	2017-10-19 – 2021-02-17
	https://github.com/mspnp/microservices-reference-implementation									
S07	Practitioner	Reference	Java, JavaScript	9	20,003	20	287	36179f0	fd5ee3c	2015-03-29 – 2021-01-19
	https://github.com/sqshq/PiggyMetrics									
S08	Practitioner	Reference	C#	10	92,132	14	291	d48abe8	e1266b8	2017-09-26 – 2021-02-13
	https://github.com/EdwinVW/pitstop									
S09	Practitioner	Reference	JavaScript, PHP, Java, Python, Shell	7	5,916	19	362	ae1a16f	a8f89df	2018-01-10 – 2021-02-11
	https://github.com/instana/robot-shop									
S10	Practitioner	Reference	Go	4	405,290	2	94	4a713f4	c2a16d5	2017-08-14 – 2018-10-14
	https://github.com/digota/digota									
S11	Industry	Reference	JavaScript, Java	6	89,182	12	31	44bb648	e83dec0	2017-04-22 – 2018-12-10
	https://github.com/microsoft/PartsUnlimitedMRPmicro									
S12	Practitioner	Reference	Java	16	1,140	4	90	7f7974c	d8c91ac	2016-04-26 – 2018-05-31
	https://github.com/sczyh30/vertx-blueprint-microservice									
~~S13~~	*<dropped during commit labeling>*									
	https://github.com/microservices-demo									

(...): languages in nested repository; ms: number of microservices; loc: lines of code; cont: number of contributors

of all systems we studied in this work, and to enable us to discriminate the nature of each change made to the system in a comprehensive way. After this manual analysis, both authors derived regular expressions (examples in quotations, see also our repository[1]) to semi-automatically distinguish three **types of atomic changes** based on file types, code changes, and commit messages.

We remark that these types of changes reflect on the state-of-practice distinguishing between technical and business-driven (i.e., service) changes:

Technical (T) changes represent changes to the technologies that enable the development of the actual microservices, for instance, to underlying frameworks, libraries, or the architecture. Consequently, such changes are not directly related to the (functional) business logic of microservices. This type of change includes changes related to the deployment environment (e.g., Docker, Kubernetes), the build process (e.g., update to the continuous integration pipeline), or dependencies (e.g., changing a dependency in a Maven descriptor). For example, after a manual analysis of the technologies used in a given system, we could capture its architectural changes by looking for specific configuration files. Specifically, we used "config" as keyword in combination with ".*\.xml$|.*\.properties$|.*\.config$|.*\.json$" as regular expression for file types.

Service (S) changes are concerned with evolving the business logic of the system, and not the underlying technology. We tailored the respective labels and expressions individually to each system we analyzed based on manually identified boundaries (i.e., using the system's documentation) of each bounded context that defines a microservice inside a certain system. All the analyzed systems follow the good practice of isolating services into modules (e.g., package or maven modules in Java, modules in Go and Python) that are reflected as directories at the file system level. Our manual analysis here was to identify which sub-parts of these directories where containing the business code, and to capture such information into a regular expression. For example, in a system where the `customers` package contains all the business logic related to business-driven customer management as Java classes, we tracked the evolution of the services with the following regular expression: ".*\.customers.*\.java$".

Miscellaneous (M) changes are associated to file types that are highly specific to the system and not concerned with technical or service changes, such as updating icons or publishing a static web user interface. For example, the user interface layer of a system using the Java Server Face technology (where web pages are implemented as Java classes) can be tracked using the following regular expression when such a component is stored in a package named `web_ui`: ".*\.web_ui.*\.java$".

Note that we derived the respective expressions for each system individually (since each system used different architectures and technologies), and manually validated the automatic classification. Each expression (see Figure 2 for an example overview) is a more specific description of a change's purpose, and we relied on an open-coding-like process to derive the expressions from the investigated artifacts. Through this step, for each system, we obtained a label for each unique atomic change c_i. The labeling process ensures that only one label is associated to each atomic change, and that all changes are labeled inside a given system to ensure consistency. In a functional way, we rely on a function $label(c_i) = T|S|M$.

After labeling all changes, the two authors cross-validated whether the regular expressions of each system were reasonable by manually comparing the other authors' expressions to a sample of labeled commits and the atomic changes therein of the respective systems. The inspection did not reveal unfeasible or wrongly assigned labels. As commits are defined as sets of atomic changes, a given commit can contain changes of different nature (i.e., with different label). Since all changes are labeled, a commit has at least one (it is uniform, all changes are of the same nature) and up to three labels. The labels associated to a given commits are the union of the labels associated to each of its atomic changes. As a result, the label(s) associated to a given commit C_1 are the powerset of S, T, M (excluding the empty set): $labels(C_i) \in \{\{T\}, \{S\}, \{M\}, \{T, S\}, \{T, M\}, \{S, M\}, \{T, S, M\}\}$. In summary, we can investigate *atomic changes* (i.e., individual code changes) and their tangling in *commits* (i.e., a set of atomic changes that have been submitted together) when analyzing the evolution of microservices.

2.5. Case Analyses

To address our research questions, in a first step, we performed descriptive (e.g., descriptive statistics) and exploratory (e.g., plotting microservice co-evolution) analyses of the labeled commits and atomic changes to obtain an overall understanding of the systems' evolution. In particular, we investigated the distributions of change types across our dataset, in each of the systems, and regarding their (co-)evolution (see Figure 7 for visualizations we used). Based on this analysis, we obtained a deeper understanding of the microservice evolution in our subject systems, which we used to observe trends. Each of these trends represents observations we made multiple times in our dataset. For instance, some microservices evolve almost always independently, while others almost exclusively co-evolve.

In a second step, we aimed to understand the causes for the recurring observations. For instance, we found that most (co-)evolution in our subject systems has been caused by technical changes. To focus on actual microservice evolution (i.e., the business logic of the systems), we extended our analysis and focused only on this type of change. For this purpose, we re-visited each system's documentation (e.g., wiki pages, pull-requests, discussions, and readme) and architecture (e.g., reverse engineered class diagrams, models in the documentation) to identify microservices defined by the developers using an open-coding-like process (i.e., we collected services the developers explicitly named in the documentation and causes for each service change, such as performance issues or bugs, reported in commit messages or other documentation). Then, we assigned each service change to its corresponding microservice by mapping the identified names to pull requests, commit messages, code comments, or the actual source code and by considering whether the evolved code or files belong to a specific service. Using this mapping, we created UpSet plots (Lex et al., 2014) to visualize the co-evolution of the microservices (cf. Figure 9 for an example). Based on the documentation, our plots, and the source code, we refined our previous observations. In particular, we discussed the manually elicited causes for service changes we identified during our analysis to understand why the recurring observations occur.

3. Findings & Discussion

In this section, we describe our observations on microservice evolution and aggregate them into findings (F_i). We then discuss possible causes for these findings based on the insights we obtained while analyzing the evolution histories of the systems manually. Building on the first step of our case analysis (cf. Section 2.5), we focus on the quantitative analysis of our data, aiming to understand coarse-grained evolution trends (e.g., what type of changes are the main drivers of evolution and how they co-evolve). Then, based on removing the technical noise and incorporating more of our qualitative analysis during the second step, we describe more fine-grained observations regarding the evolution of actual microservices. Note again that we relied on atomic changes or commits during our analysis, depending on the respective goals (e.g., we considered commits to investigate the tangling of service atomic changes). For simplicity, we use tuples to express a certain combination of atomic changes, for instance, `<t,s,_>` refers to technical (`t`) and service (`s`) changes (e.g., when comparing them or investigating their tangling in commits) and excludes (`_`) miscellaneous (`m`) changes.

3.1. Coarse-Grained Evolution

To obtain a first understanding of how microservice-based systems evolve, we consider individual (i.e., each system individually) and global (i.e., across all systems) evolution trends in our 11 subject systems. In detail, we observe these trends in terms of what the primary type of changes is in each subject system. For this analysis, we focus on the amount of atomic changes to each system's artifacts (e.g., modifications of configurations or libraries represent technical atomic changes).

Observations (RQ_1). At first, we compared the distributions of labels we assigned to each atomic change within each system in our dataset. In Table 2, we summarize for each subject system the amount (in per cent) of each type of change: technical, service, and miscellaneous. For each system, we defined that it is driven by a certain evolution trend based on the type of change that accounts for the

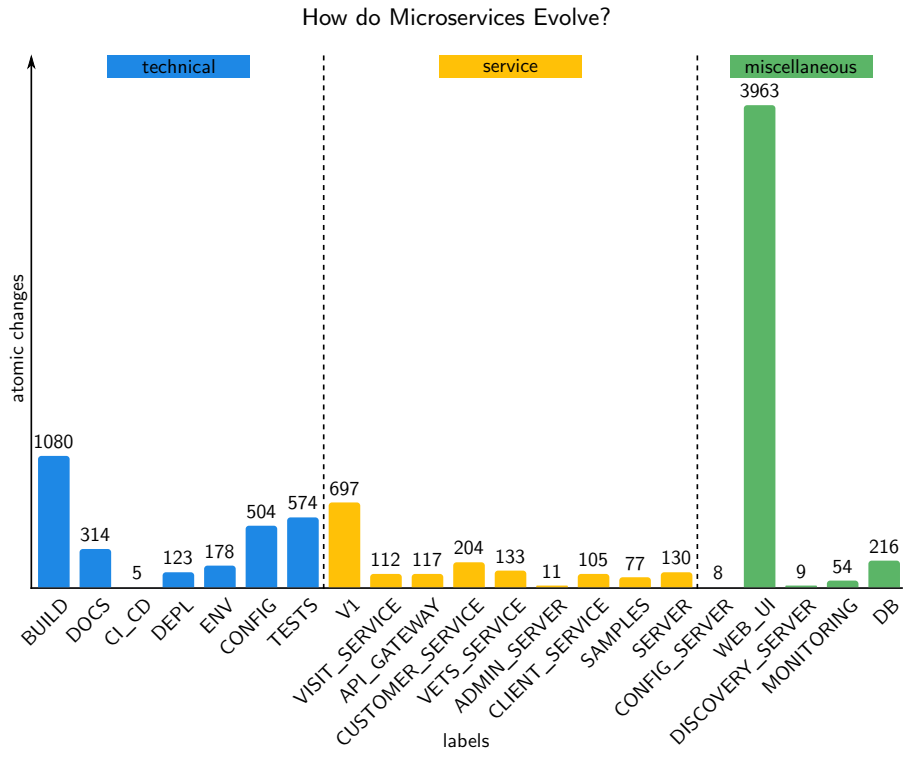

Figure 2: Labels assigned to the atomic changes of S03, highlighting for the type of change (technical, service, miscellaneous).

Table 2
Coarse-grained evolution trends of atomic changes.

system	t	s	m	trend
S01	41.74 %	37.59 %	20,67 %	—
S02	14.09 %	84.73 %	1.19 %	s
S03	32.25 %	18.41 %	49.34 %	m
S05	58.50 %	41.27 %	0.23 %	t
S06	55.74 %	38.93 %	5.33 %	t
S07	48.17 %	31.73 %	20.10 %	t
S08	76.75 %	20.81 %	2.44 %	t
S09	63.42 %	23.70 %	12.87 %	t
S10	40.27 %	23.15 %	36.58 %	—
S11	57.58 %	7.66 %	34.76 %	t
S12	51.60 %	31.65 %	16.76 %	t

t: technical, s: service, m: miscellaneous

highest amount of atomic changes. Note that we consider a system to follow a certain trend only if the predominant type of change occurs at least 15 % more often than the next type of change (typically, it is much more, e.g., 61.35 % for S08). These 15 % serve as a sanity check to discard small majorities (e.g., 4.15 % for S01) and increase our confidence in the trend, since we cannot conduct a statistical test for this purpose. Out of the 11 systems, and despite the claim of microservices being "business-oriented" (i.e., they should be primary concerns of evolution), we only observed a single service-driven evolution trend (S02). In contrast, we can see seven technical-driven (S05, S06, S07, S08, S09, S11, S12) and one miscellaneous-driven evolution trends (S03). Two of the systems do not exhibit a certain trend (S01, S10), since the difference between at least two types of changes is less than 5 % in both (i.e., <t,s,_> and <t,_,m>, respectively).

In Figure 2, we display a bar chart with the amount of atomic changes in S03 together with their more fine-grained expressions. We can see that miscellaneous changes are predominant, primarily driven by changes to the web user interface (WEB_UI). For this case, we considered the user interface to represent miscellaneous changes, since it has been implemented as an independent application instead of a reusable microservice, and thus it is system-specific. The evolution of this user interface contributes to around 46 % of the changes in the system (3,963 atomic changes).

For two systems, namely S01 (cf. Figure 3) and S10 (cf. Figure 4), we found no type of change that clearly outperforms the other two. Regarding S01, we can see a tie between technical and service atomic changes, while for S10 the tie is between technical and miscellaneous atomic changes. Note that both ties involve the technical type of change. In the same direction, technical atomic changes are predominant in seven systems and occur more often than service atomic changes in all but one system (S02). As we can see in Figure 3, technical atomic changes include building (BUILD), deploying (DEPL), configuring (CONFIG), and testing (TESTS) a system, as well as setting up its environment (ENG, LIBS). Interestingly, our more detailed analysis revealed that even S02 with 16 microservices may not be as dominantly service-driven as it seems. In fact, many of the service atomic changes represent refactorings, because S02 seems to have been involved in a large re-engineering.

In Figure 5, we display box-plots to compare between the three types of changes across all of our subject systems. We can see that, on average, more than half of all atomic

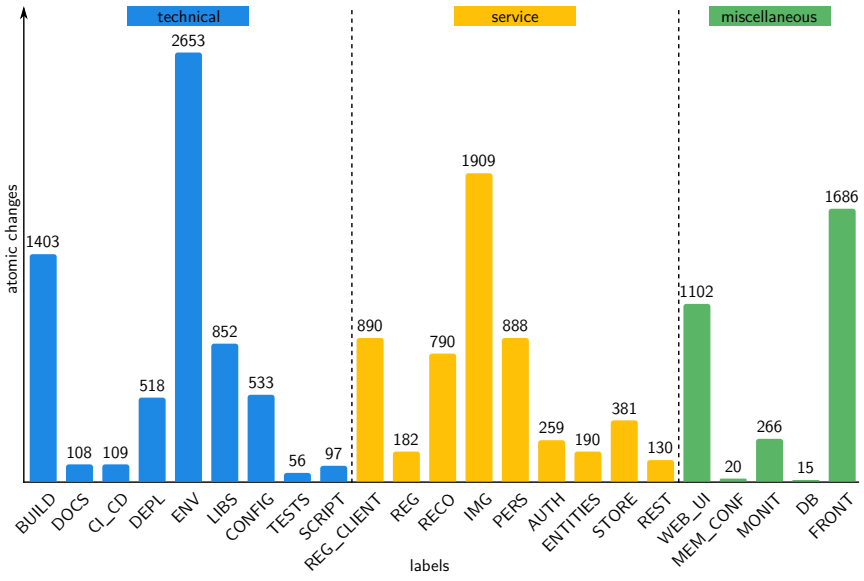

Figure 3: Labels assigned to the atomic changes of S01, highlighting for the type of change (technical, service, miscellaneous).

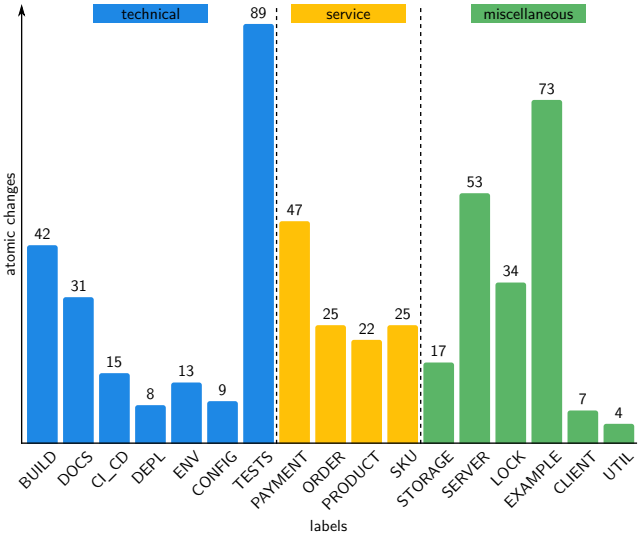

Figure 4: Labels assigned to the atomic changes of S10, highlighting for the type of change (technical, service, miscellaneous).

changes in our dataset represent technical changes. Changes to the actual business logic (i.e., microservices) or system specifics (i.e., miscellaneous) each occur in less than 25 % of the atomic changes. Moreover, we can see that miscellaneous changes vary the most, while service changes are comparably constant across the systems (with S02 being a drastic outlier).

> F_1: Evolution is Rarely Service-Driven
>
> Out of the 11 systems, only one exhibits a service-driven evolution trend, while technical atomic changes represent the majority of evolution across all systems.

Discussion (RQ_2). The fact that technical atomic changes are predominant in the life-cycle of our subject systems (seven out of 11 trends) is caused by the distributed and heterogeneous nature of microservice-based systems (Dragoni et al., 2017; Soldani et al., 2018). For this reason, microservice-based systems involve many technical components, for instance, to support communication (e.g., commits with a focus on implementing or changing proxies), deployment (e.g., atomic changes in Docker files), or to improve a system's security (e.g., changes related to authentication and cryptography mechanisms). In the seven systems driven by technical changes, a common trend is that a large amount of changes is related to deployment (e.g., Dockerfiles, deployment scripts) and configuration (e.g., properties or JSON file containing configuration options for service registry, database, or external providers). For instance, this represents up to 40 % of all the changes involved in S11. Two outlier projects (S07, S12) exhibit a significant amount of changes in their build system (a mix of Makefiles and Maven configurations), which we can explain due to these systems relying on external dependencies for concerns, such as load-balancing, API gateway exposition, and circuit breaking. A tremendous effort is spent in these projects to keep up to date with the release cycle of such external libraries. Interestingly, none of these two projects is using automatic tooling like Dependabot[10] to keep track of such dependencies.

Other systems that exhibit miscellaneous-driven trends (e.g., S03) differ from this trend, due to their mono-repository culture that integrates microservices with other applications (i.e., web user interface). Consequently, the version history is unbalanced towards miscellaneous changes. S01 is particularly interesting to observe, since its technical dependency heavily relies on modifying the build configuration as well as its environment. This system does not exhibit a technical trend, because it is "polluted" by the inclusion of the web user interface in the system. It is interesting to notice that

[10] https://github.com/dependabot

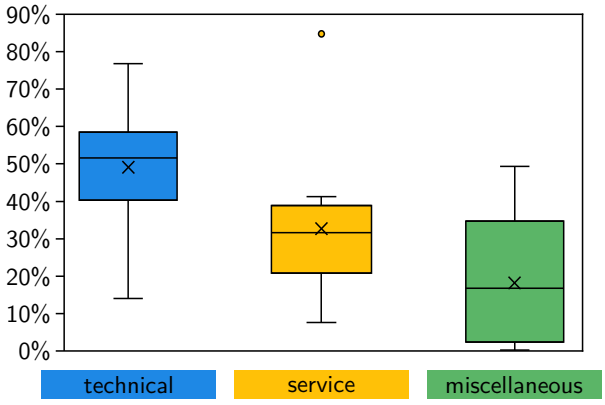

Figure 5: Distribution of atomic changes across all systems.

the objective of the developers of S01 to develop a reference infrastructure that is highly configurable for research purpose is reflected in its change history.

Summary. It is most interesting that our observations contradict the proclaimed service-driven development and evolution of microservice-based systems. Instead, the evolution of our microservice-based systems is primarily technical-driven (**RQ$_1$**). This implies that developers' efforts are not focused on developing the actual microservices, but go primarily into technical changes. Since such technical changes are often related to updates in underlying technologies (**RQ$_2$**), further automation to propagate and manage updates (e.g., to avoid breaking API changes) would be helpful. In fact, the many technical changes we identified even for such smaller systems indicate that the technologies used for combining heterogeneous microservice architectures are not yet mature enough for maintaining microservices and require further research as well as developer support. Then, developers could pay less attention to technical changes and instead could focus on developing the actual microservices.

3.2. Coarse-Grained Co-Evolution

In the previous section, we considered atomic changes to unveil fundamental evolution trends. Next, we focus on commits and the types of atomic changes they comprise to understand the co-evolution of change types. To this end, we distinguish sets of commits based on the tuples of our change types. For instance, a commit classified as `<_,s,m>` is a commit that contains atomic changes related to the service and the miscellaneous type. Note that we classified a few commits as comprising none of the three types of changes (`<_,_,_>`), for instance, merges, initial commits, or other version-control specifics that are not relevant for our study.

Observations (RQ$_1$). We display the different tuples of types of changes and the ratios with which they occur across our subject systems in Figure 6. Again, we can see that most evolution relates to technical changes, and commits with only technical atomic changes (`<t,_,_>`) are predominant; representing more than half of all commits. All other tuples (except for the empty one) are more similar, contributing to roughly between 2 % and 15 % of the commits on average. We can also see that service atomic changes are more often part of commits with multiple change types (`<t,s,_>`, `<_,s,m>`, `<t,s,m>`) than an independent commit (`<_,s,_>`).

F$_2$: Technical Burden Drives Evolution

Reinforcing F$_1$, commits containing only technical atomic changes occur far more often than all other possible combinations. For some systems, up to 70 % of the commits only involve purely technical atomic changes, which do not modify any microservices.

Discussion (RQ$_2$). The overwhelming amount of technical-only commits is surprising in the context of microservice architectures, in which business-oriented development is promoted. This enforces F$_1$, showing that even if technical changes are an important part of the architectural history, they are primarily contributed standalone. For example, we can study commit `581415`[11] from S01. This commit is a good example of a technical set of atomic changes, specifically adding new libraries, modifying the configuration files to refer to the new libraries, and updating the build artifact with the new information—while, at the same time, making a very small modification to a servlet involved in a business service to change how the business logic handles a parameter. In the same system, commit `bc1f66a`[12] represents service-driven evolution for the persistence, authentication, and web interface that has to include a technical dimension by modifying a configuration file named `aop.xml` to support the evolution of the persistence service. From a broader point of view, we can explain this discrepancy with the interleaving of technical concerns inside a system, where modifying one artifact (e.g., container networking) can impact others (e.g., distributed logging). Consequently, any change in a system often causes additional technical changes to handle the diversity of existing artifacts (e.g., to update dependencies, interactions, or interfaces between them). By investigating the version-control history, we found that this is due to developers apparently not distinguishing the types of changes. Thus, even though microservices are often advocated to be business-oriented, their evolution is heavily entangled with other concerns; preventing developers to focus on the actual business concerns. Similar results have been found in related studies, which indicate that despite their benefits, microservices lead to huge technical complexity (Bogner et al., 2021; Wang et al., 2021).

Summary. Our observations contradict the advocated business orientation that microservice-based systems would allow for. In contrast, we found that the evolution of microservice-based systems is heavily driven by technical(-only) changes (**RQ$_1$**). As a result, developers seem to be almost always concerned with technicalities of their systems, which asks for better tool support to help them focus on the actual

[11] https://github.com/DescartesResearch/TeaStore/commit/581415134b2e76f12153751fcda02ade8c4dd451

[12] https://github.com/DescartesResearch/TeaStore/commit/bc1f66a0002f1d6027456cadfdb3e07888680044

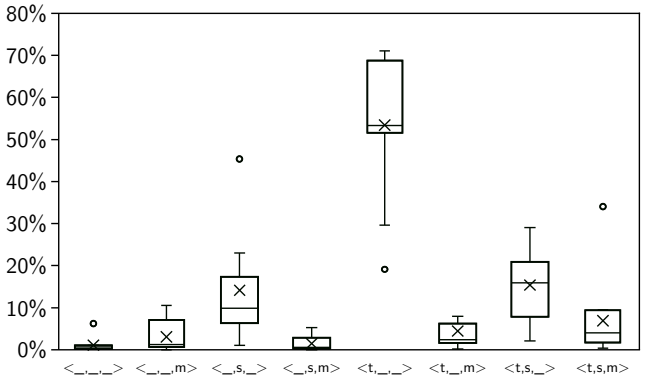

Figure 6: Tuples of (co-)evolution at commit level.

microservices. As the primary cause for this issue, we identified the heavy tangling of technical concerns with all other concerns of a microservice-based system, which seems to stem form a conflict between aiming for independent—yet interacting, diverse, and depending—microservices that can build on various technologies (**RQ$_2$**).

3.3. Coarse-Grained Evolution Over Time

Until now, we investigated the total amounts of changes occurring in our subject systems. However, we did not yet analyze whether the identified trends and observations change over time (i.e., in a system's life-cycle). For this purpose, we now study to what extent each type of atomic changes occurs during each system's evolution. We present a corresponding overview in Figure 7, showing the ratios of changes over the period of time covered by our analysis. Please note that these plots do not represent the total amount of changes in a period, but only their distribution (i.e., at different points in time the 100 % may represent 10 or 100 commits). Similarly, we remark that S12 seems to be completely stable for a long period, but this is caused by a single commit to a branch roughly one year after the last previous commit.

Observation (RQ$_1$). We can observe two primary trends in Figure 7. First, for most systems, service atomic changes occur more frequently in earlier phases of their life-cycle (i.e., S01, S03, S06, S07, S08, and S10). In some systems, service atomic changes even constitute the main type of changes at this point in time. This softens our previous observations that evolution is not driven by business concerns. Instead, the business logic seems to drive the initial development of most systems. A prime example for the evolution shift over time is S06 (cf. Figure 7e). Early on, service atomic change contribute to roughly 45 % of all changes, with a decline through the middle of the life-cycle, until all atomic changes become purely technical. In between, miscellaneous atomic changes occur.

Second, the other systems (i.e., S02, S05, S09, S11, and S12) show a more stable distribution of the types of changes over their whole life-cycle—with some exhibiting interesting outliers (e.g., S11). For instance, we can see for S09 (cf. Figure 7h) that the amount of service atomic changes starts small, expands, collapses, and expands again. Consequently, there is quite some variation in the extent of service evolution throughout the system's life-cycle. Interestingly, S11 (cf. Figure 7j) starts with almost no service atomic changes, which only occur in the second half of the system's life-cycle before completely diminishing again. These observations could be explained with developers starting by setting up the technical environment before implementing the actual microservices and typical differences in a system's life-cycle.

> F$_3$: Evolution Changes Over Time
>
> We can observe two different trends of evolution: (i) a "decreasing" amount of service changes over a system's life-cycle, and (ii) an almost "stable" distribution of all change types across a system's life-cycle.

Discussion (RQ$_2$). Typically, microservices are advocated as a concept to align business logic and system implementation. Hence, we expected that the business logic would play an important role in the development and evolution of microservice-based systems. Based on our detailed analysis, the more "stable" trend is sometimes close to this expectation: The developers find a routine in terms of development, working in increments that add business logic and update the underlying technology to support the incremental advancement. Abstracting our previous observations that most changes are related to technical concerns, this evolution trend is similar to the advocated assumption and indicates a constant activity in developing business logic.

On the contrary, and predominant in our subject systems, the "decrease" trend does not align to the expectation, since service changes constitute less and less of a system's evolution. Instead, the systems are primarily maintained and updated on a technical level, which may be a bias of observing some open-source systems that serve as reference architectures. In such cases, the initial effort may go into developing the showcase and reference architecture, until only dependencies and libraries are updated. That S02 (cf. Figure 7b) as a real-world product exhibits the "stable" evolution trend would support this assumption, but the data again seems skewed due to the large re-engineering in this system. It is interesting future work to explore which of these evolution trends characterizes what type of microservice-based system (e.g., showcase versus real product), whether the trends depend only on user requests (e.g., the reference architectures may be forked and evolved independently by users), or imply a certain workflow implemented by the developers.

Summary. The novel observations in this section somewhat soften the previous observations we obtained, indicating that there are two recurring evolution trends: a "decrease" in service atomic changes and a "stable" distribution of all types of changes (**RQ$_1$**). It is unclear which of these trends is valuable or representative for what types of systems. Building on our understanding of the systems, we speculate that these trends depend on the purpose of and interest in a system (**RQ$_2$**), but to what extent this is the case and what best practices may be relevant in each trend is an opportunity for future research.

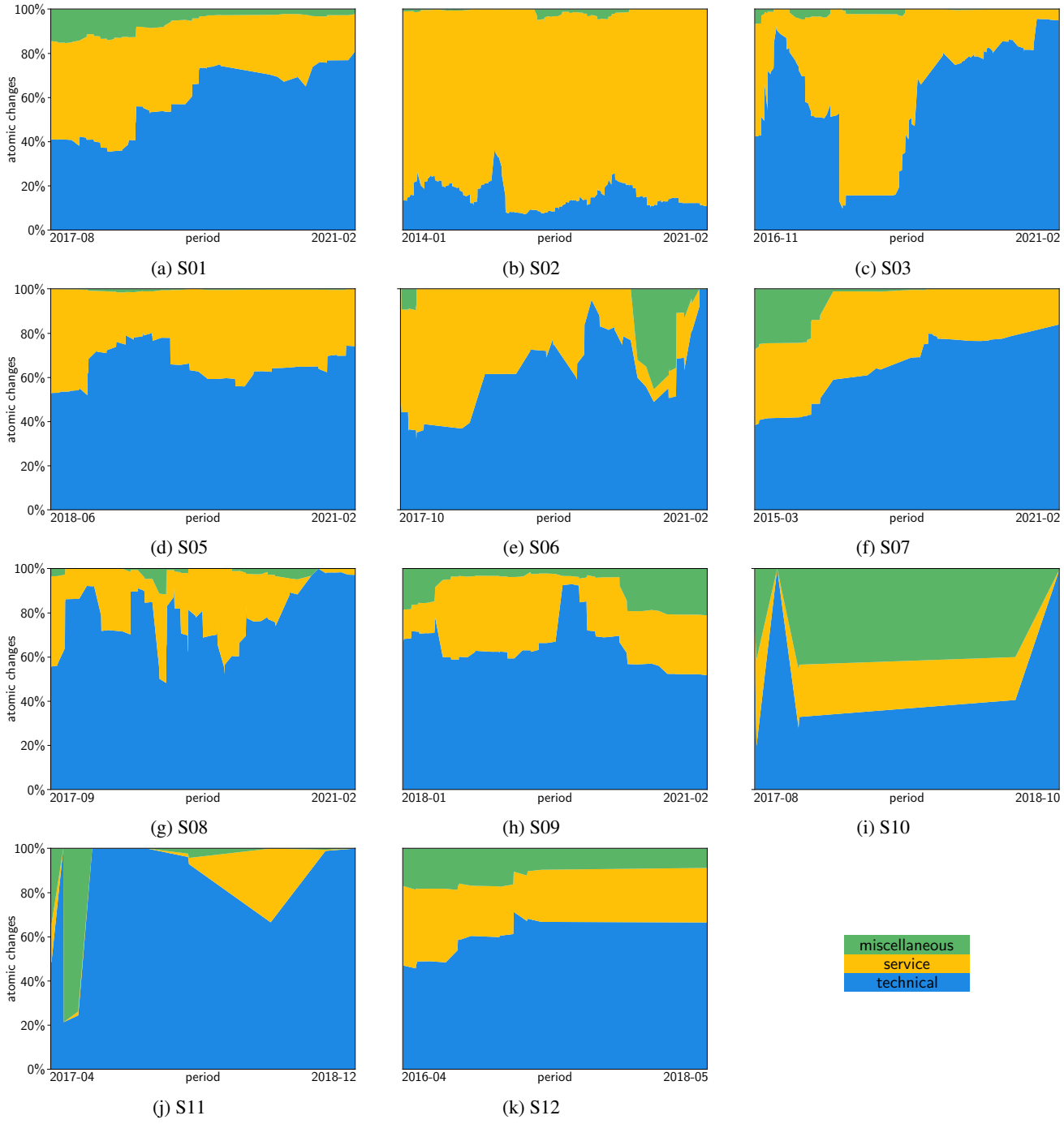

Figure 7: Evolution of atomic changes over time.

3.4. Microservice Co-Evolution

To focus on the co-evolution of microservices themselves, we next analyzed the subset of atomic changes that are related to the business logic of a system. For this purpose, we filtered all atomic changes to only involve those labeled as service. Then, we considered two (or more) microservices as co-evolving if atomic changes to both were part of the same commit. Visualizing the resulting co-evolution sets of a complete evolution history using Venn diagrams would be incomprehensible, due to the large number of intersections. So, we relied on the more structured alternative representation of UpSet plots (Lex et al., 2014) used in biology research to study genomics.

As an example, consider the UpSet plot in Figure 8. Each line represents one microservice, with the total number of commits involving that microservice on the left. For instance, the line `checkout` indicates that 48 commits involved atomic changes to that microservice. Each column represents a subset of that service set, highlighting the intersection between microservices (i.e., atomic changes to these microservices occur in the same commits) as well as how often these occur. For instance, the first column displays that the microservice `cart` occurs 28 times on its own in a commit without any

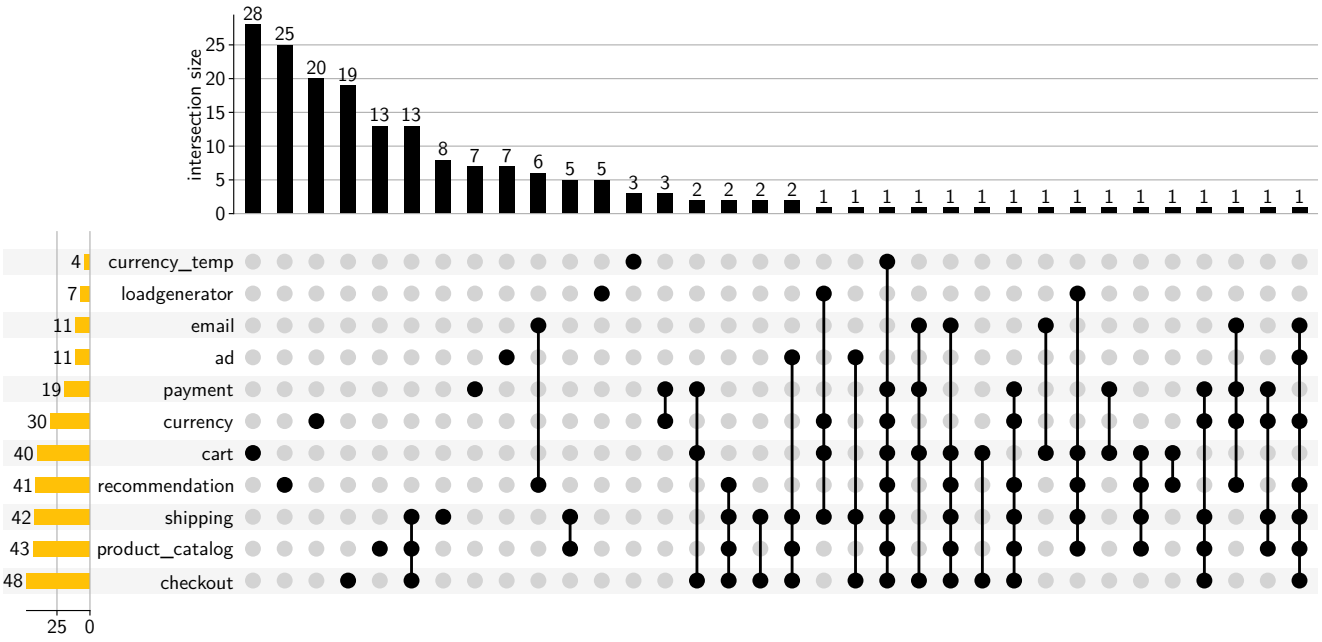

Figure 8: Co-evolution of atomic service changes in commits of S05 (co-evolving).

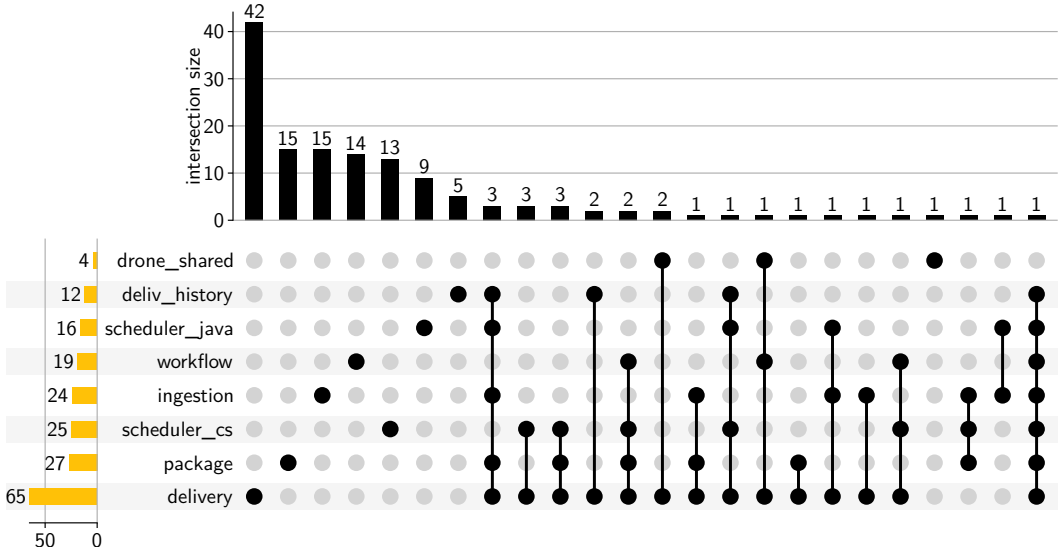

Figure 9: Co-evolution of atomic service changes in commits of S06 (independent evolution).

other microservice. In contrast, the last column shows that one commit involved atomic changes to seven (out of 11) different microservices (i.e., a "shotgun surgery" commit).

Observations (RQ_1). Investigating the UpSet plots for all systems, we identified two classes of systems:

1. those in which microservices evolve primarily independently from each other, and
2. those in which microservices primarily co-evolve.

We display examples for these two classes in Figure 8 and Figure 9, respectively. A common point we observed for all systems is that the co-evolution plots have a long-tailed distribution: Whatever the actual co-evolution pattern, commits with co-evolving services occur always—even if not frequent.

On the left side of the distributions, we can observe whether some microservices co-evolve frequently. For example, S05 (see Figure 8) has a subset of three regularly (13 commits) co-evolving microservices: `shipping`, `product_catalog`, and `checkout`. Moreover, these microservices are involved in much more co-evolution with the other microservices, too. Overall, they evolve less often independently than in combination with other microservices—namely, `shipping` alone occurs only in eight commits, which are fewer commits than the three-service subset we exemplified alone. Additionally, `email` has never been evolved independently. Finally,

we can see that the core microservices of S05 (`checkout`, `shipping`, `product_catalog`, `recommendation`, `cart`) evolve the most. Since these represent typical functionalities of a web-shop, this observation supports the idea that microservices reflect the business logic of a system if we remove technical and miscellaneous changes.

In Figure 9, we display S06, in which microservices are evolved much more independently. We can see that seven out of eight microservices evolve primarily alone, contributing to the majority of commits (113 of 139). Still, `delivery` (last line) is not only the microservice with the most independent evolution (42 commits), but also the one with the most co-evolution (23 commits). It is only missing from two co-evolution commits for the whole history of S06, and it co-evolves with every other microservice. Nonetheless, the evolution in this system is primarily independent, particularly compared to other systems (e.g., S05).

F_4: MICROSERVICE CO-EVOLUTION IS THE NORM

Even when focusing only on business logic, microservices do rarely evolve in isolation.

Discussion (RQ_2). Microservices emphasize the idea of loose coupling and lightweight interfaces. Some books dedicate entire chapters to such concerns (Newman, 2015), but based on our study results, we doubt that the envisioned independence of microservices can be achieved in real-world systems. For instance, consider the service `delivery` from S06 (cf. Figure 9). During our detailed analysis, we found that this is a fundamental microservice for the system (an e-shop)—but even though the developers aimed to evolve it independently (42 commits), it also co-evolves with all other microservices. We found that refactorings to shared APIs are a primary cause for this situation. Arguably, such co-evolution patterns will occur even more frequently if we consider systems with many more microservices.

An interesting situation occurs when a new feature is introduced into a system, resulting in changes to multiple microservices to support the feature. For example, commit `d08a58` of S05 introduces a reliable process for ordering. Thereby, it requires changes to `product catalog`, `shipping`, and `checkout`. According to the design philosophy of microservices, such cross-cutting features should not exist in microservices. Instead, they should be designed within one or as a new, individual microservice. However, in S05, there is no microservice dedicated to ordering goods from the e-shop, which is why changes to multiple microservices are needed. While it may seem counter-intuitive that this microservice does not exist in an e-shop, we found that it can be easily explained by the way microservice APIs are used. Introducing a microservice for ordering would cause a central bottleneck, which would hamper the scalability and lead to static coupling between microservices through API calls. Thus, the developers decided to introduce the cross-cutting feature instead of creating a new microservice—resulting in co-evolution and hidden (but implicit) coupling.

Summary. We observed that even in such systems in which microservices are evolved somewhat independently, co-evolution occurs (RQ_1). There are differences in the degree of co-evolution, but it seems an emergent problem how to manage or resolve the co-evolution between subsets of microservices (e.g., "shotgun surgery" commits). Inspecting the causes, we found that intentional design decisions about a system's architecture can cause co-evolution, for instance, when introducing cross-cutting features (RQ_2). Reflecting on all of our observations, we argue that the independence of microservices is challenging to achieve in practice, and we require tool support to facilitate developers' tasks.

4. Threats to Validity

In the following, we present the internal and external threats to validity that may bias our findings.

Internal Validity. Since we performed a large-scale analysis (i.e., 99,804 atomic changes by hundreds of developers), it was not possible to involve the original developers to validate the labeling. So, there may be wrongly labeled changes in our dataset, for instance, because we are missing specific domain knowledge. We aimed to mitigate this threat by cross-checking our labels and using regular expressions, also ensuring that they were comparable between the systems. Moreover, the large number of changes we analyzed means that our results are less error-prone to such faults.

We considered commits and the involved files to represent a system's (co-)evolution. As we found when aiming to label and analyze the systems (cf. Section 2.2), this does prevent us from analyzing certain projects, for instance, those employing a multi-repository philosophy. Moreover, developers may misuse commits and commit policies can vary between projects in a way that is not immediately explicit when analyzing open-source projects. Both issues threaten our results, since we could not consider all commits and cannot be absolutely sure that they are all reliable. Still, studying software evolution based on commits is an established research method, and this threat is mitigated by the fact that we considered systems that should have legit version-control histories (e.g., by Microsoft and Google). The data for such systems is similar to the remaining ones, increasing our confidence in our results. By analyzing 11 systems from the state-of-practice, extracted from a curated list of projects known for their intrinsic quality, we believe that we mitigated this threat as good as we can.

External Validity. The main threat to the validity of our study is whether the initial corpus of systems we analyzed is representative. We mitigated this risk by starting from a curated list of 55 systems and using a rigorous selection process to select 11 feasible ones. As a result, we cover a diverse set of microservice-based systems, including authority and productive systems. Still, we considered only open-source systems (due to availability) with a smaller number of microservices and primarily reference architectures. So, our results may not be fully transferable into practice—but parts

of our results align to experience reports and related research (cf. Section 5), which increases our confidence.

Like any empirical study relying on repository mining, our study relies on commits as atomic units for version control, which are the only source of information available at large scale. The immediate threat is that the commit history is not immutable, and thus follows the Napoleon principle: "*History is a set of lies agreed upon.*" In our case, such *lies* can be `squash` operations (which squash commits into a single one) or git commands employed with a `--force` argument. There is no way to identify such cases, since the history only contains the resulting commit graph and not the way it was build. We mitigated this risk by looking at (i) the way developers were integrating merge requests in the main project and (ii) the commit frequency in each system's main repository. Based on this information, we believe that the history of the 11 projects are viable for our study, since they exhibit regular commit frequency and no signs for squashing policies.

Finally, we focused on single-repository systems only, because identifying co-evolution of changes requires that the changes are committed at the same time. This leads to a validity threat, since we did not consider multi-repository projects. Still, it is not possible to investigate such projects using our methodology. That being said, we did not define multi-repository projects as an exclusion criterion, and we only had one multi-repository project in our 13 candidates. So, we argue that the exclusion of (one) multi-repository systems does not threaten our results.

5. Related Work

Microservice principles advocate for an evolutionary design, since they are supposed to be independent and autonomous (Balalaie et al., 2018; Lewis and Fowler, 2014). However, similar to any architectural style, the concrete implementation of microservices can be of arbitrary quality (Bogner et al., 2021). That is, the design of microservices is a fundamental aspect of their evolvability (Haselbock et al., 2018). For example, wrongly defining microservice boundaries and responsibilities can cause different anti-patterns (Martin, 2002; Taibi et al., 2020), such as the ones we discussed. Despite this, very few research has focused on the evolution, maintenance, or technical debt of microservice (Bogner et al., 2021). This reinforces the need for studies like the one we have presented in this article.

Esparrachiari et al. (2018) discuss dependency management as a crucial activity for microservice-based systems. In contrast to our study, the authors focus on dependencies to third-party software and services. Similarly, Neri et al. (2019) conducted a multi-vocal review on design principles, architectural smells, and refactorings for microservices. Again, the authors are concerned with dependencies, but only in the context of multiple microservices in one container contradicting advocated design principles. Our study complements such works with a focus on dependencies among the microservices within the same system. In a recent study Taibi et al. (2020), present a taxonomy of 20 microservice anti-patterns based on an extensive mixed-methods study involving an industry survey, a literature review, and interviews. Four of those anti-patterns are related to dependencies, namely (i) cyclic dependencies due to cyclic chain calls, (ii) miss-scoped microservices, (iii) shared libraries, and (iv) shared persistence. We observed and discussed similar problems in our study, complementing such research with empirical evidence. Lastly, Bogner et al. (2019, 2021, 2017) conducted different empirical studies to understand the maintenance and evolution of microserivces. For instance, the authors performed a gray literature review and interviews with practitioners to identify challenges in assuring the evolvability of microservices. Their research highlights the need for providing specialized techniques, tools, and metrics to developers to manage microservices and their dependencies. However, Bogner et al. have not investigated the actual evolution of microservices, which is a gap our study fills—emphasizing similar problems, but with more in-depth insights into how and why real microservice-based systems evolve.

A recent literature survey (Tran et al., 2020) summarizes the diverse research that has been conducted on the (co-)evolution in service-oriented architectures in general. The authors find that various types of changes have been defined (e.g., deep versus shallow), and that tool support is required to help developers keeping services compatible. Similarly, we argue that developers require further tool support to facilitate the technical evolution of microservice-based systems and allow them to focus on (co-)evolving the actual microservices. Moreover, Tran et al. discuss how the literature they surveyed relates to microservices, indicating that managing co-evolution of microservices is likely challenging developers even more due to the heterogeneous nature and complexity of such systems. Our article provides empirical data that underpins this hypothesis, and thus highlights the need for further investigating (co-)evolution and its management in microservice-based systems—which exhibit different properties than other service-oriented architectures (e.g., loose coupling of small services).

6. Conclusion

As a consequence of large organizations (e.g., Netflix and Uber) adopting them, microservices have gained a lot of attention in both industry and research in recent years. In the literature, we can find many pieces of work that provide recommendations on how to design microservice-based systems from scratch, however the (co-)evolution of microservices has rarely been discussed. Also, there is a lack of evidence whether the intended independence of microservices, which is proclaimed by existing design recommendations, can actually be achieved in practice. To fill these gaps, we reported a large-scale empirical study on the (co-)evolution of microservices in 11 open-source systems, covering a history of 7,319 commits. Based on our analysis, we presented and discussed four key findings:

F_1 Technical changes drive the evolution of microservices more than the actual business logic.

F_2 Even when changes are related to business logic, they are mostly intertwined with technical changes.

F_3 The evolution of microservice-based systems can vary heavily over time, and it is unclear which trends represent best practices.

F_4 Even when focusing on business logic alone, isolated microservices are an illusion.

Our insights can guide practitioners when evolving their microservices and researchers in designing new techniques for managing microservice evolution.

By deriving these findings through our study, we learned two key lessons. First, while business logic is essential when defining microservice, the business logic's role fades over time and is often subsumed by technicalities to keep up with the pace of software dependencies and technological updates. The costs of such technical maintenance must be investigated in an industrial context rather than an open-source one. Second, the co-evolution of microservices is something that is not taken into account by classical architecture smells for microservices. This is an important problem in terms of the architectural designs, since microservices are supposed to be independent by design. We believe that further studying what types of smells exist in microservice-based systems (e.g., disposable co-evolution) and how to identify them as soon as possible by leveraging version histories is an urgent research direction to support software developers.

As future work, we plan to validate our findings with practitioners by replicating our study in industry and conducting other confirmatory studies (e.g., surveys). Furthermore, there is a need for tool support to identify and monitor undesired co-evolution of microservices, and to improve the independence of microservices (e.g., using specialized refactorings). Finally, we need to understand what impact certain microservice smells exhibit in practice to identify what problems they can cause.

CRediT authorship contribution statement

Wesley K. G. Assunção: Conceptualization, Methodology, Software, Validation, Investigation, Data Curation, Writing - Original Draft, Writing - Review & Editing, Visualization. **Jacob Krüger:** Conceptualization, Methodology, Validation, Investigation, Data Curation, Writing - Original Draft, Writing - Review & Editing, Visualization. **Sébastien Mosser:** Conceptualization, Methodology, Software, Validation, Investigation, Data Curation, Writing - Original Draft, Writing - Review & Editing, Visualization. **Sofiane Selaoui:** Methodology, Software, Validation, Investigation, Data Curation, Visualization.

References

Ananieva, S., Greiner, S., Kehrer, T., Krüger, J., Kühn, T., Linsbauer, L., Grüner, S., Koziolek, A., Lönn, H., Ramesh, S., et al., 2022a. A conceptual model for unifying variability in space and time: Rationale, validation, and illustrative applications. Empirical Software Engineering 27, 101.

Ananieva, S., Greiner, S., Krüger, J., Linsbauer, L., Grüner, S., Kehrer, T., Kühn, T., Seidl, C., Reussner, R., 2022b. Unified operations for variability in space and time, in: Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems, pp. 1–10.

Ananieva, S., Greiner, S., Kühn, T., Krüger, J., Linsbauer, L., Grüner, S., Kehrer, T., Klare, H., Koziolek, A., Lönn, H., et al., 2020. A conceptual model for unifying variability in space and time, in: International Systems and Software Product Line Conference (SPLC), pp. 1–12.

Assunção, W.K.G., Ayala, I., Krüger, J., Mosser, S., 2021. International Workshop on Variability Management for Modern Technologies (VM4ModernTech 2021), in: International Systems and Software Product Line Conference (SPLC), ACM.

Assunção, W.K.G., Colanzi, T.E., Carvalho, L., Garcia, A., Pereira, J.A., de Lima, M.J., Lucena, C., 2022. Analysis of a many-objective optimization approach for identifying microservices from legacy systems. Empirical Software Engineering 27. doi:10.1007/s10664-021-10049-7.

Assunção, W.K.G., Krüger, J., Mendonça, W.D.F., 2020. Variability Management meets Microservices: Six Challenges of Re-Engineering Microservice-Based Webshops, in: International Systems and Software Product Line Conference (SPLC), ACM. pp. 22:1–6.

Assunção, W.K.G., Colanzi, T.E., Carvalho, L., Pereira, J.A., Garcia, A., de Lima, M.J., Lucena, C., 2021. A multi-criteria strategy for redesigning legacy features as microservices: An industrial case study, in: IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE. pp. 377–387. doi:10.1109/SANER50967.2021.00042.

Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D.A., Lynn, T., 2018. Microservices migration patterns. Software Practice and Experience 48, 2019–2042.

Benni, B., Mosser, S., Caissy, J.P., Guéhéneuc, Y.G., 2020. Can Microservice-Based Online-Retailers be Used as an SPL? A Study of Six Reference Architectures, in: International Systems and Software Product Line Conference (SPLC), ACM. pp. 24:1–6. doi:10.1145/3382025.3414979.

Bogner, J., Fritzsch, J., Wagner, S., Zimmermann, A., 2019. Assuring the Evolvability of Microservices: Insights into Industry Practices and Challenges, in: International Conference on Software Maintenance and Evolution (ICSME), IEEE. pp. 546–556.

Bogner, J., Fritzsch, J., Wagner, S., Zimmermann, A., 2021. Industry practices and challenges for the evolvability assurance of microservices. Empirical Software Engineering 26. doi:10.1007/s10664-021-09999-9.

Bogner, J., Wagner, S., Zimmermann, A., 2017. Automatically measuring the maintainability of service- and microservice-based systems: A literature review, in: 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement, Association for Computing Machinery, New York, NY, USA. p. 107–115. doi:10.1145/3143434.3143443.

Carvalho, L., Garcia, A., Colanzi, T.E., Assunção, W.K.G., Pereira, J.A., Fonseca, B., Ribeiro, M., de Lima, M.J., Lucena, C., 2020. On the performance and adoption of search-based microservice identification with tomicroservices, in: IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 569–580. doi:10.1109/ICSME46990.2020.00060.

Chacon, S., Straub, B., 2014. Pro git: Everything you need to know about Git. Second ed., Apress. URL: https://git-scm.com/book/en/v2.

Conradi, R., Westfechtel, B., 1998. Version Models for Software Configuration Management. ACM Computing Surveys 30, 232–282. doi:10.1145/280277.280280.

Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L., 2017. Microservices: Yesterday, Today, and Tomorrow. Springer International Publishing, Cham. pp. 195–216. doi:10.1007/978-3-319-67425-4_12.

Esparrachiari, S., Reilly, T., Rentz, A., 2018. Tracking and controlling microservice dependencies: Dependency management is a crucial part of system and software design. Queue 16, 44–65. doi:10.1145/3277539.3277541.

Familiar, B., 2015. Microservices, IoT, and Azure. Springer. doi:10.1007/978-1-4842-1275-2.

Fowler, S.J., 2016. Production-ready microservices: building standardized systems across an engineering organization. O'Reilly.

Haselbock, S., Weinreich, R., Buchgeher, G., 2018. An expert interview study on areas of microservice design, in: IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA), IEEE. doi:10.1109/soca.2018.00028.

Hindle, A., German, D.M., 2005. Scql: A formal model and a query language for source control repositories, in: International Workshop on Mining Software Repositories (MSR), pp. 1–5.

Jacobson, J., 2009. A formalization of darcs patch theory using inverse semigroups. Available fro m ftp://ftp. math. ucla. edu/pub/camreport/cam09-83. pdf .

Joseph, C.T., Chandrasekaran, K., 2019. Straddling the Crevasse: A Review of Microservice Software Architecture Foundations and Recent Advancements. Software: Practice and Experience 49, 1448–1484.

Kecskemeti, G., Marosi, A.C., Kertesz, A., 2016. The entice approach to decompose monolithic services into microservices, in: 2016 International Conference on High Performance Computing & Simulation (HPCS), IEEE. pp. 591–596.

Krüger, J., Assunção, W.K.G., Ayala, I., Mosser, S., 2022. VM4ModernTech: International Workshop on Variability Management for Modern Technologies, in: International Systems and Software Product Line Conference (SPLC), ACM.

Lewis, J., Fowler, M., 2014. Microservices: a definition of this new architectural term. URL: https://www.martinfowler.com/articles/microservices.html.

Lex, A., Gehlenborg, N., Strobelt, H., Vuillemot, R., Pfister, H., 2014. UpSet: Visualization of Intersecting Sets. IEEE Transactions on Visualization and Computer Graphics 20, 1983–1992. doi:10.1109/tvcg.2014.2346248.

Luz, W., Agilar, E., de Oliveira, M.C., de Melo, C.E.R., Pinto, G., Bonifácio, R., 2018. An experience report on the adoption of microservices in three brazilian government institutions, in: Proceedings of the XXXII Brazilian Symposium on Software Engineering, pp. 32–41.

Martin, R.C., 2002. The single responsibility principle. The principles, patterns, and practices of Agile Software Development , 149–154.

Neri, D., Soldani, J., Zimmermann, O., Brogi, A., 2019. Design principles, architectural smells and refactorings for microservices: a multivocal review. SICS Software-Intensive Cyber-Physical Systems 35, 3–15. doi:10.1007/s00450-019-00407-8.

Newman, S., 2015. Building Microservices. 1st ed., O'Reilly Media.

Rahman, M.I., Panichella, S., 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.

Sampaio Jr., A.R., Kadiyala, H., Hu, B., Steinbacher, J., Erwin, T., Rosa, N., Beschastnikh, I., Rubin, J., 2017. Supporting Microservice Evolution, in: International Conference on Software Maintenance and Evolution (ICSME), IEEE. pp. 539–543.

Soldani, J., Tamburri, D.A., Van Den Heuvel, W.J., 2018. The pains and gains of microservices: A systematic grey literature review. Journal of Systems and Software 146, 215–232. URL: https://www.sciencedirect.com/science/article/pii/S0164121218302139, doi:https://doi.org/10.1016/j.jss.2018.09.082.

Sriraman, A., Wenisch, T.F., 2018. μ suite: a benchmark suite for microservices, in: 2018 IEEE International Symposium on Workload Characterization (IISWC), IEEE. pp. 1–12.

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

Taibi, D., Lenarduzzi, V., Pahl, C., 2020. Microservices Anti-patterns: A Taxonomy. Springer International Publishing, Cham. pp. 111–128. doi:10.1007/978-3-030-31646-4_5.

Thönes, J., 2015. Microservices. IEEE Software 32, 116–116.

Tran, H.T., Nguyen, V.T., Phan, C.V., 2020. Towards Service Co-Evolution in SOA Environments: A Survey, in: International Conference on Context-Aware Systems and Applications & International Conference on Nature of Computation and Communication (ICCASA & ICTCC), Springer. pp. 233–254. doi:10.1007/978-3-030-67101-3_19.

Viggiato, M., Terra, R., Rocha, H., Valente, M.T., Figueiredo, E., 2018. Microservices in practice: A survey study. arXiv preprint arXiv:1808.04836 .

Vučković, J., 2020. You are not netflix, in: Microservices. Springer, pp. 333–346.

Wang, Y., Kadiyala, H., Rubin, J., 2021. Promises and challenges of microservices: an exploratory study. Empirical Software Engineering 26. doi:10.1007/s10664-020-09910-y.

Waseem, M., Liang, P., Shahin, M., Ahmad, A., Nassab, A.R., 2021. On the Nature of Issues in Five Open Source Microservices Systems: An Empirical Study, in: International Conference on Evaluation and Assessment in Software Engineering (EASE), ACM. pp. 201–210. doi:10.1145/3463274.3463337.

Wolfart, D., Assunção, W.K.G., da Silva, I.F., Domingos, D.C.P., Schmeing, E., Villaca, G.L.D., Paza, D.d.N., 2021. Modernizing Legacy Systems with Microservices: A Roadmap. Association for Computing Machinery, New York, NY, USA. p. 149–159.

Yin, R.K., 2018. Case Study Research and Applications: Design and Methods. Sage.

Yuan, E., 2019. Architecture interoperability and repeatability with microservices: an industry perspective, in: 2019 IEEE/ACM 2nd International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE), IEEE. pp. 26–33.

Zdun, U., Wittern, E., Leitner, P., 2020. Emerging trends, challenges, and experiences in devops and microservice apis. IEEE Software 37, 87–91. doi:10.1109/ms.2019.2947982.