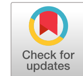
The Journal of Systems & Software 177 (2021) 110968

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study☆



Saulo S. de Toledo *, Antonio Martini, Dag I.K. Sjøberg

University of Oslo, Oslo, Norway

A R T I C L E I N F O

Article history:
Received 24 August 2020
Received in revised form 8 March 2021
Accepted 30 March 2021
Available online 9 April 2021

Keywords:
Cost of software
Cross-company study
Software quality
Software maintainability
Qualitative analysis

A B S T R A C T

Background: Using a microservices architecture is a popular strategy for software organizations to deliver value to their customers fast and continuously. However, scientific knowledge on how to manage architectural debt in microservices is scarce.
Objectives: In the context of microservices applications, this paper aims to identify architectural technical debts (ATDs), their costs, and their most common solutions.
Method: We conducted an exploratory multiple case study by conducting 25 interviews with practitioners working with microservices in seven large companies.
Results: We found 16 ATD issues, their negative impact (interest), and common solutions to repay each debt together with the related costs (principal). Two examples of critical ATD issues found were the use of shared databases that, if not properly planned, leads to potential breaks on services every time the database schema changes and bad API designs, which leads to coupling among teams. We identified ATDs occurring in different domains and stages of development and created a map of the relationships among those debts.
Conclusion: The findings may guide organizations in developing microservices systems that better manage and avoid architectural debts.
© 2021 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

Microservice architecture is a relatively new architectural style that is becoming increasingly popular in the industry. A microservice is a small component that can be developed and deployed independently, is easy to scale, and has a single responsibility (Dragoni et al., 2017). Such characteristics make microservices particularly convenient for continuous delivery (Thönes, 2015). Microservices are built around business capabilities and provide an architectural style capable of organizing cross-functional teams around services (Dragoni et al., 2017). Microservices are supposed to support companies to deliver value to their customers in a fast and continuous fashion.

Despite their advantages, microservices are still an emerging technology. There are still drawbacks of using such an architectural style, such as data inconsistency among various services (Furda et al., 2018). As a simple example, suppose an online bookstore has three services: one for managing the book catalog, another to manage orders, and a third for deliveries to the customers. Each of those services has its own database. When a client finishes an order, the book must be removed from the stock by the book service, and the delivery must be triggered. When the orders database is updated, the product remains in an inconsistent state until the other services finish updating their databases. Meanwhile, the client and the store own the book since it is still available in the stock and orders databases, and there is a chance the company will sell more books than it has, causing problems for the company.

Companies are still learning how to properly migrate from old monolithic software to systems that use microservices. There are still several challenges in implementing microservices from scratch to make them easy to maintain and evolve (Bogner et al., 2019b), which leads to a situation in which practitioners make architectural sub-optimal decisions that lead to a benefit in the short term, but increase the overall costs in the long run, i.e., a situation described by a metaphor known as Architectural Technical Debt (ATD) (Verdecchia et al., 2018).

Di Francesco et al. (2017, 2019) state that fundamental principles, claimed benefits, and quality (including maintainability) of microservices still must be proven by research and envisioned further qualitative studies with practitioners. Studies on managing ATD in microservices, which directly affects software maintainability, would be part of such a request.

There is a body of gray literature and books that concern microservices, migrations, and related practices. Still, such literature

☆ Editor: Gabriele Bavota.

* Corresponding author.
E-mail addresses: saulos@ifi.uio.no (S.S. de Toledo), antonima@ifi.uio.no (A. Martini), dagsj@ifi.uio.no (D.I.K. Sjøberg).

https://doi.org/10.1016/j.jss.2021.110968
0164-1212/© 2021 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

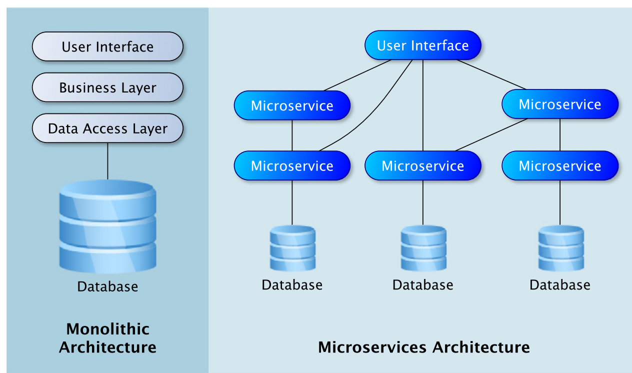


Fig. 1. Monolithic and microservice architectures.

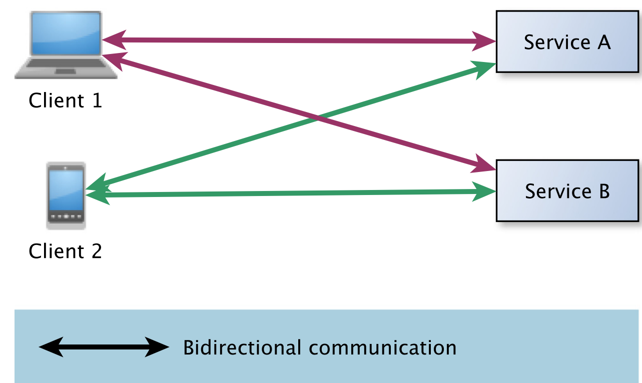


Fig. 2. Microservices synchronous communication.

does not focus specifically on ATD. In particular, it does not explore the diversity of challenges regarding ATD and microservices across companies.

As a contribution to meeting the needs described above, we conducted a multiple-case study in seven international Europe-based companies to investigate ATD in microservices through the following research questions:

- **RQ1:** What are the most critical ATD issues in microservices?
- **RQ2:** What are the negative impacts of such ATD issues?
- **RQ3:** What are possible solutions to repay or avoid such ATD issues?

For each of the identified ATDs, we outlined how to determine interest and principal, which is needed to develop metrics for quantifying the costs of the ATDs. Our contributions may also support practitioners' decision-making in projects involving microservices.

2. Background

This section describes the concepts of microservices and architectural technical debt.

2.1. Microservices

Lewis and Fowler (2014) provide the most accepted definition of the microservices architectural style: "an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API". Microservices are frequently described as an alternative to monolithic applications, built and deployed as a single unit (see Fig. 1), since well-known companies, such as Amazon and Netflix, have been using microservices to overcome difficulties with their previous monolithic architectures (Lewis and Fowler, 2014). Applications that use microservices are easier to scale, have shorter cycles for testing, build and release, and are less frequently affected by downtime than monolithic applications (Fowler, 2015). These and other characteristics make microservice applications particularly desirable for continuous delivery. In fact, new features can be independently and continuously tested and delivered by updating specific microservices without changing the whole product, which drastically reduces lead time. Still, there are challenges, such as the risk of increased data inconsistency and operational complexity (Fowler, 2015).

Microservice architecture may also be described as one way of implementing Service Oriented Architecture (SOA), although there are different views on this claim (Zimmermann, 2017). Certainly, SOA describes a set of applications that cannot be considered microservices. For example, many SOA applications are implemented using an Enterprise Service Bus (ESB), an infrastructure that mediates requests among services, intercepts communications, and provides transformation capabilities, among other functions (Niblett and Graham, 2005). ESBs can be a single monolithic artifact that can be deployed together with the services at the same place (Montesi and Weber, 2016). In contrast, microservices employ what is called a *dump pipe* or a communication layer without business logic. Other characteristics can also describe SOA but not microservice architectures. Rademacher et al. (2017) provide a list of such characteristics including the following: (i) there is no guidance about the service granularity in SOA, while a microservice architecture suggests that each service represents one capability only; (ii) SOA may support transport protocol transformation, while microservices usually apply REST over HTTP or a protocol supported by a message bus; (iii) there are several service types in SOA (e.g., business, enterprise, application), while there are only two types of microservices—that is, they are functional (representing business capabilities) or infrastructure (providing technical capabilities like authentication and authorization) services.

Despite such differences, there are several concepts and techniques in the area of microservices that were borrowed from SOA, such as the approaches for communication detailed in Section 2.1.1; the concepts of scalability, service discovery, and service registry detailed in Section 2.1.2; and the concepts of service availability and responsiveness detailed in Section 2.1.3. There are some adaptations of those concepts and techniques in a microservice architecture, such as a limited set of communication protocols. Other concepts such as Service Mesh, explained in Section 2.1.4, emerged to support microservice architectures (Li et al., 2019).

In summary, while there is an overlap, there are certainly many differences in techniques and concepts between SOA and microservice architecture. In this paper, we focus on microservices.

2.1.1. Microservices communication

In a microservice architecture, clients and services may communicate directly with each other synchronously (Newman, 2017) (Fig. 2) or through an API gateway (Montesi and Weber, 2016) (Fig. 3). They can also communicate asynchronously through a message bus (Newman, 2017), which holds the message in a queue until one or more services consume(s) it, as shown in Fig. 4. Such communication may also be mixed: for example, by using a synchronous request with an asynchronous response.

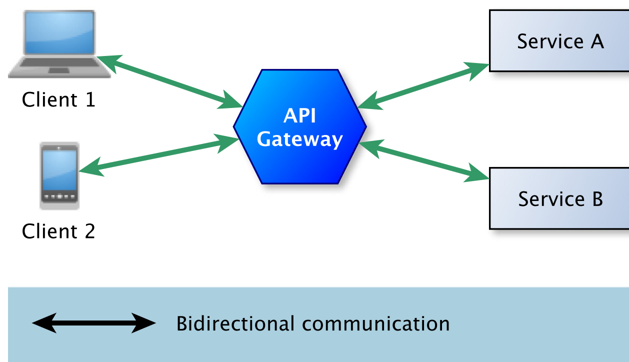


Fig. 3. The API gateway.

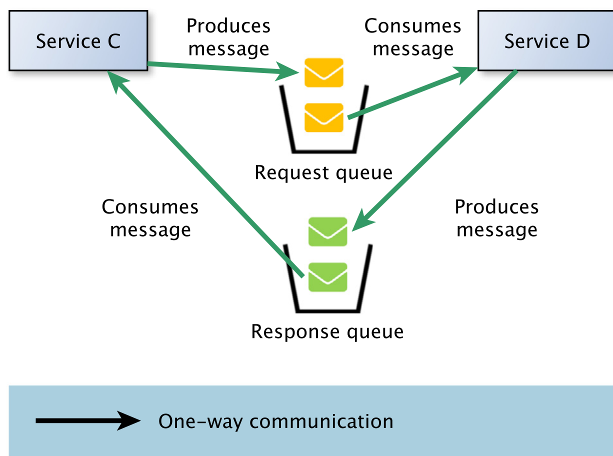


Fig. 4. Microservices asynchronous communication.

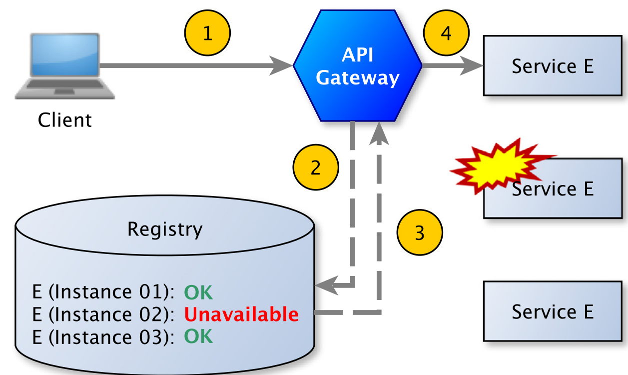


Fig. 5. Service discovery, registry and services instances.

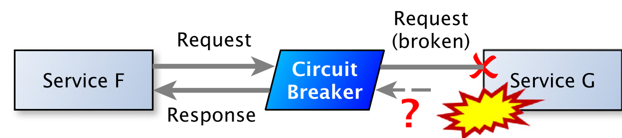


Fig. 6. The circuit breaker.

2.1.2. Scalability and service discovery

In the microservices context, scalability is the service's ability to cope and perform under high demand. A scalable microservice adapts itself to the needs of its consumers (Márquez et al., 2018). It is possible to scale those services by running multiple instances of them; each instance of the same service should be able to replace any other so that if one fails or already has high traffic, another working instance may be used in its place (see service *E* in Fig. 5). An available instance of the service may be found by a service discovery mechanism in such situations. The service discovery (i.e., the act of finding a running instance of a service) may be performed by consulting a service registry (i.e., a service that stores information about other services and their available and unavailable instances; for example, service *E* instances 01–03 in Fig. 5) (Montesi and Weber, 2016). One way of performing the service discovery is exemplified in Fig. 5. The API gateway can query the registry to find a running instance of service *E* that the client requires. The service discovery method exemplified before is called server-side discovery, as opposed to the client-side discovery, in which the client is responsible for querying the service registry (Montesi and Weber, 2016).

2.1.3. Service availability and responsiveness

In the setting where one service (consumer) requests information from a remote service (producer), *service availability* is the producer's ability to accept the request in a timely manner (Richards, 2016). Inversely, *service responsiveness* is the consumer's ability to receive a timely response (Richards, 2016). When a consumer makes a request to a producer, the consumer does not know whether they will receive a response. Since they cannot wait indefinitely, it is common for the consumer to wait a specific period of time (the timeout) until they give up and consider the request as a failure. At this point, the consumer may try the request again. When the producer has a very high load or is entirely inaccessible, consumers may keep (i) repeatedly try to connect until the producer is available or (ii) wait the whole timeout period until they can take some other action (Montesi and Weber, 2016). Both cases waste resources.

A better technique is to use *circuit breakers*. When services are down or demonstrate high latency and are mostly unusable, a circuit breaker takes the lead and immediately responds to the consumer (Fig. 6). The consumer uses fewer resources waiting for services that failed (i.e., they will not keep running and waiting for an unavailable service that may never send a response). Such a solution prevents network or service failure from cascading to other services since some may depend on the consumer in our example (Montesi and Weber, 2016).

2.1.4. Service mesh

Finally, *service meshes* have emerged with the popularization of microservice architectures. Service meshes are dedicated infrastructure layers acting on the service-to-service communication, designed to make the services safe, reliable, and more observable. They usually implement several of the mechanisms introduced in previous sections, as well as others such as service discovery mechanisms, load balancing, encryption, circuit breaking, and service observability (Li et al., 2019). These mechanisms are not strictly required when implementing microservices, but they might be beneficial, especially when there are many microservices.

2.2. Architectural technical debt

This section gives an overview of ATD, its management, and its relationship to related concepts.

2.2.1. An overview of ATD

ATD is a type of technical debt (TD) consisting of suboptimal architectural solutions, which deliver benefits in the short-term but increase overall costs in the long run. Identifying ATD is particularly important since problems in the architecture may slow down new functionalities and raise the related costs. Several authors (Besker et al., 2017; Ernst et al., 2015; Kruchten et al., 2012) describe ATD as the most challenging type of TD to be unveiled and managed due to the lack of research and practical tool support.

The three main concepts of TD are the debt itself and its interest and principal (Avgeriou et al., 2016):

- **Debt:** A sub-optimal solution that has short-term benefits but will generate future interest payment is called a debt. For example, suppose a functionality is implemented using different components. If the developers create the components without carefully planning their interfaces in order to develop them faster, the solution may end up with tightly coupled components. Such a solution may be easy and fast to develop, but the product's maintenance may be costly. Due to tightly coupled components, changes in one of the components may cause consequential changes in other components. When the product is updated, it may take a long time to change and test all the components, slowing down the delivery of new functionalities.
- **Interest:** The extra cost that must be paid because of a debt, or the amount that will be saved if there is no such debt, is called interest. In the previous example, the interest is the cost of the additional effort needed to test and update all the dependent components each time a component is changed.
- **Principal:** The cost of developing a solution that avoids the debt, or the cost of refactoring a solution to avoid the debt, is called the principal. In the previous example, the principal is the cost of the effort (e.g., time and resources) required to develop the involved components' interfaces so that they are not tightly coupled and can be changed and tested independently from each other.

It can be profitable in some particular circumstances to accumulate the debt (Besker et al., 2018b). In theory, deciding whether to accumulate the debt is supported by a simple calculation: If the interest is less than the principal, it is better to accumulate the debt. If the interest is more significant than the principal, the debt should be avoided (Schmid, 2013; Martini and Bosch, 2016). However, it is not easy to know—or measure—the actual costs in practice regarding either the principal or the interest. It is still important for the involved stakeholders to be conscious of a debt's principal and interest. Practitioners need to make decisions on their ATD.

It is difficult to avoid the accumulation of some of the ATDs during the software's life cycle (Martini et al., 2015). Thus, it is important to know when these debts should be repaid and when to avoid their accumulation. Areas like microservices still lack ways of identifying and measuring ATD (de Toledo et al., 2019), which motivates this study.

2.2.2. ATD management

Managing ATD is difficult (Besker et al., 2018a) but it is important to repay the debt (Li et al., 2014). Li et al. (2014) describe the ATD management process through the following activities:

- **ATD identification:** In this phase, the ATD items (including their interest and principal) are detected and described.
- **ATD measurement:** In this phase, the debts' costs and benefits are analyzed and estimated.
- **ATD prioritization:** In this phase, the items are sorted by some criteria (e.g., importance) to decide which ATD item must be repaid first or if ATD should be repaid instead of investing in other activities, such as developing new features.
- **ATD repayment:** In this phase, architectural decisions are made to repay the debt, even if partially.
- **ATD monitoring:** In this phase, ATD items are monitored over time regarding their costs and benefits.

In this study, we start this process by identifying ATD in microservices. We then indicate what should be measured and contribute with information helpful for ATD prioritization and monitoring. We also present solutions for ATD repayment.

2.2.3. ATD versus related concepts

Fig. 7 presents the relationship between ATD and other concepts such as architectural patterns, anti-patterns, erosion, drift, and smells. All these concepts have been associated with ATD. A few others, such as defects and degraded system qualities, are also discussed in the TD literature and are briefly discussed in this section. Although various studies exist on these different concepts, there is no comprehensive work clarifying their relationships. Therefore, we report our interpretation of such concepts based on the available literature.

The most up-to-date definition of TD, available in the Dagstuhl seminar report 16162 (Avgeriou et al., 2016), states that "technical debt is 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." When discussing architecture, we focus on design constructs. Fig. 7 shows the relationship between what we call sub-optimal design constructs and some well-known terms we discuss next.

Architectural smells are indicators of design problems; as such, they may be symptoms of the presence of ATD (Martini et al., 2018) (see Fig. 7). However, architectural smells might not point to ATD in certain contexts. Martini et al. (2018) reported a situation in which a set of cyclic architectural dependencies, found in a particular graphical user interface (GUI) component, was not considered suboptimal. In that particular example, such a smell was fairly common as a normal (good) pattern. The reported smells do not represent ATD, as there is no interest and principal in such cases.

Architectural patterns are general, reusable architectural solutions (Marquez and Astudillo, 2018). When used correctly and with other context-defined needs, such as an appropriate architecture design, they may be solutions to existing ATD. However, many solutions are context-specific and should be discussed in the context of the respective debts; for example, the design of good APIs (see, for example, Mosqueira-Rey et al., 2018). An architectural pattern may or may not be a proper solution to a known issue in such contexts. Each solution has a cost, which in turn represents the principal of the debt it is removing (Fig. 7).

Architectural anti-patterns are repeatable suboptimal design constructs that violate design principles and increase the likelihood of having bugs and changes (Mo et al., 2019). An anti-pattern might represent a debt if it generates interest, but there might be cases in which the anti-pattern does not. Besides, the suffered interest of ATD can consist of something else than bugs or changes, for example, a loss of development speed or the degradation of other software qualities (Martini et al., 2018). In the example by Martini et al. (2018) that we reported when

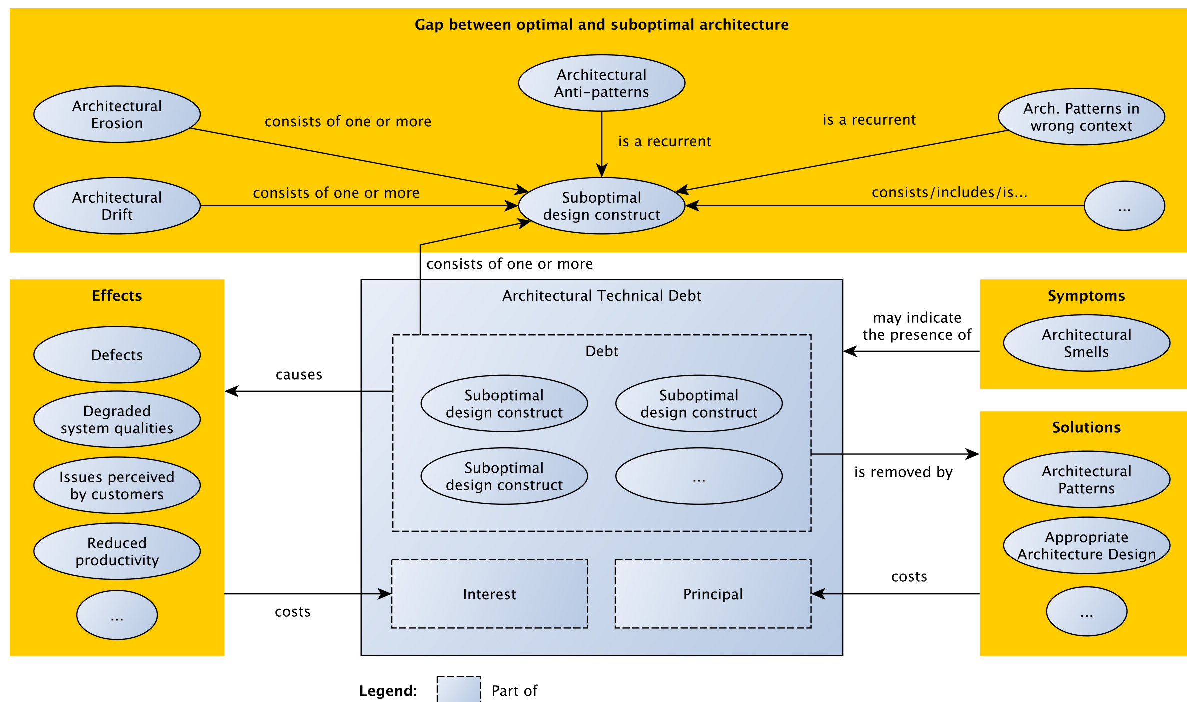


Fig. 7. ATD and the related concepts.

discussing architectural smells above, an anti-pattern could be causing the cyclic dependencies disclosed, but no interest was reported. In fact, the practitioners said the opposite: it was a solution (a pattern). The research on anti-patterns in microservices is still in its infancy; for example, there is no well-defined taxonomy (Bogner et al., 2019a). Bogner et al. (2019a) identified 14 studies on SOA patterns. Only one book and one paper focused on microservices, although many of the SOA patterns presented in the studies seem relevant to microservices.

Architectural erosion describes design changes in a system's architecture that violate the original architecture (Van Gurp and Bosch, 2002). An example of architectural erosion is the addition of workarounds to bypass the decided architecture. *Architectural drift* describes the situation in which the rules implied by the architecture are unclear, leading to divergences between parts of the architecture and making it easier to have violations (i.e., erosion) (Van Gurp and Bosch, 2002). Both terms represent how the amount of suboptimal design constructs increases over time, but they do not consider interest and principal, as in TD. Also, they become worse over time, accumulating more debts. The debt, however, remains the same. Still, its costs (interest and principal) may vary depending on the context (e.g., a debt leading to a data leak in a bank system is far worse than the same debt in a newsletter system in which, only the email addresses are compromised). The notions of architectural erosion and drift are discussed in the literature through different terms, such as architectural degeneration, software or design erosion, and architectural or design decay (De Silva and Balasubramaniam, 2012).

Defects are conditions in a software product that need to be fixed because they cause the software's malfunction or produce unexpected results. As explained by Kruchten et al. (2012), defects are visible for the customers and are, therefore, different from any kind of TD, such as ATD. Defects, as well as degraded system qualities and other issues perceived by customers, or even internal issues such as reduced productivity, might be an effect caused by the existence of some kind of TD. All these effects have a cost—the interest of the debt which caused them (Fig. 7).

Other concepts apart from those we discussed may also be used to perceive the gap between the suboptimal and optimal constructs or solutions. For example, misused architectural patterns (i.e., their use in a context they are not suitable for) may also be responsible for such a gap. However, describing such a gap between optimal and suboptimal design is not enough to formulate the problem as ATD, which is focused on the financial variables related to its costs (principal and interest) and dependent on the contexts.

3. Methodology

This study aims to identify the most common and critical ATD issues, interests, and principals in products using microservice architectures. We investigated which circumstances led to ATD and identified solutions and insights related to its occurrence. We conducted an *exploratory multiple-case study*, where each analyzed product represents a case. The remainder of this section presents the cases and how the data was collected and analyzed. Fig. 8 outlines our methodology.

3.1. Case selection

We studied seven different software products in seven large international companies. Two products were provided by different sub-companies within the same multinational conglomerate. All the products had a microservice architecture. Although in some of the products, minor parts were previously developed using a monolith design or SOA approaches, the overall architecture was considered a microservice architecture, and such parts were in the process of being migrated.

The various application domains of the products gave us diversity in investigated contexts, which helped to understand whether the found problems and solutions were widespread or domain-specific. Table 1 shows a summary of the studied companies and products. For confidentiality reasons, we named the companies *A, B, C, D, E, F,* and *G*, respectively. The application domains of the microservices projects we investigated are as follows:

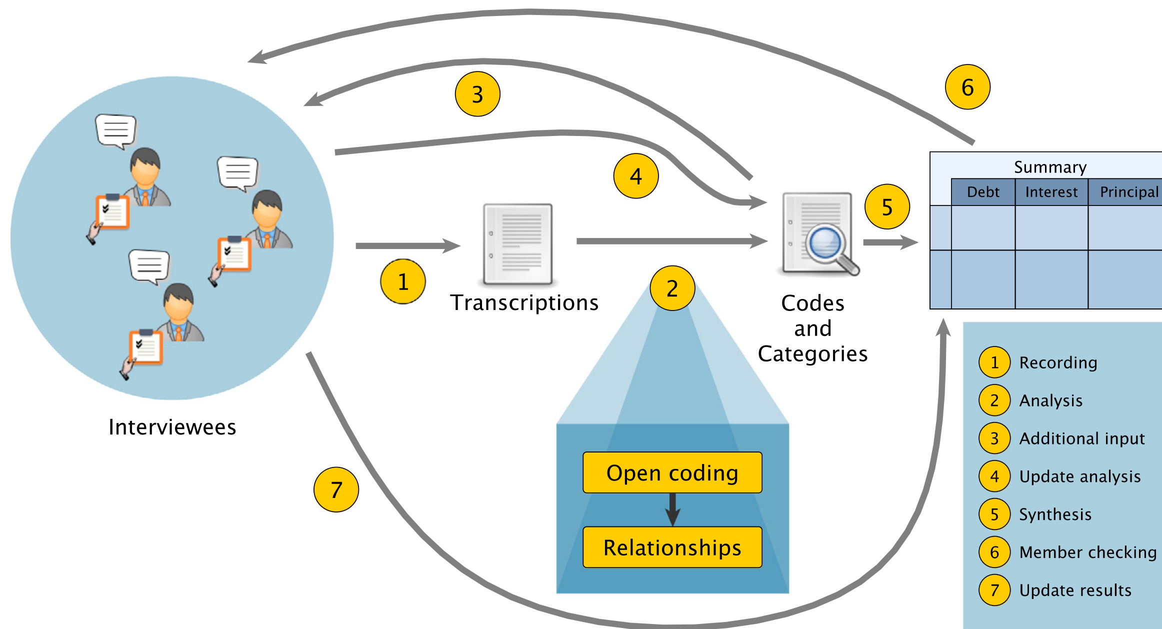


Fig. 8. Methodology overview.

Table 1
Companies context.

Context	Company						
	A	B	C	D	E	F	G
Application domain	Finance	Cloud	IoT	Health	Public services	Transport	Transport
Approx. number of employees	>30 000	>7000	>30 000	>30 000	>30 000	320	>20 000
Approx. number of employees on IT	>2000		20 000	1200	250	150	
Approx. number of employees in the case	2000	200	500	250	150	150	
Approx. number of unique microservices in the case	1000	50	80	40	400	600	3000
Age of the product	>10 years	>2 years	>2 years	>1.5 years	>10 years	>4 years	>10 years
Stage of development	Migration	Initial development	Initial development	Initial development	Evolving	Evolving	Evolving

- **Finance:** The software was used to assist users with financial operations, money management, payments, insurance, and investments.
- **Cloud:** The software provided a set of services to be used in the Cloud by third-party consumers.
- **IoT:** A software in the Internet of Things domain that was used to control, share information, and/or gather data from devices connected to the internet.
- **Health:** The software was used to provide health services, such as user profiles and medical information.
- **Public services:** The software was used to provide public services, such as payslip management and taxes.
- **Transport:** The software was used to assist users of public and private transport of both passengers and goods.

We also present the stage of each software project according to the following classifications:

- **Initial development:** The software was developed using microservices from the beginning.
- **Migration:** The software is migrating from an old solution, such as a monolith or other service approach, to microservices.
- **Evolving:** The system is consolidated as a microservice approach and is currently being maintained and evolved.

In this multiple-case study, a case is a given company's specific product. For simplicity reasons, cases are referred to by their company's name.

3.2. Data collection

We performed 25 interviews with 22 employees in different roles, as detailed in Table 2. We selected the interviewees and companies through convenience sampling (i.e., selecting from the collaboration network we had access to). All the interviewees had several years of experience in their roles. They all gave consent for the interviews to be recorded and transcribed (Step 1 in Fig. 8). We used the semi-structured interview guide presented in Appendix. The interviews lasted between one and two hours.

The study started with Company A, where we could access several employees. This helped us have a solid understanding and a rich amount of details about the initial set of existing ATDs. We then continued the study with additional companies to investigate whether the results were general or differed across contexts.

As we progressed with the interviews in different contexts, new aspects emerged, such as additional ATD instances or further details for specific instances. We updated the interview guide along the course of interviews regarding Debts 8, 10.2, 11, and

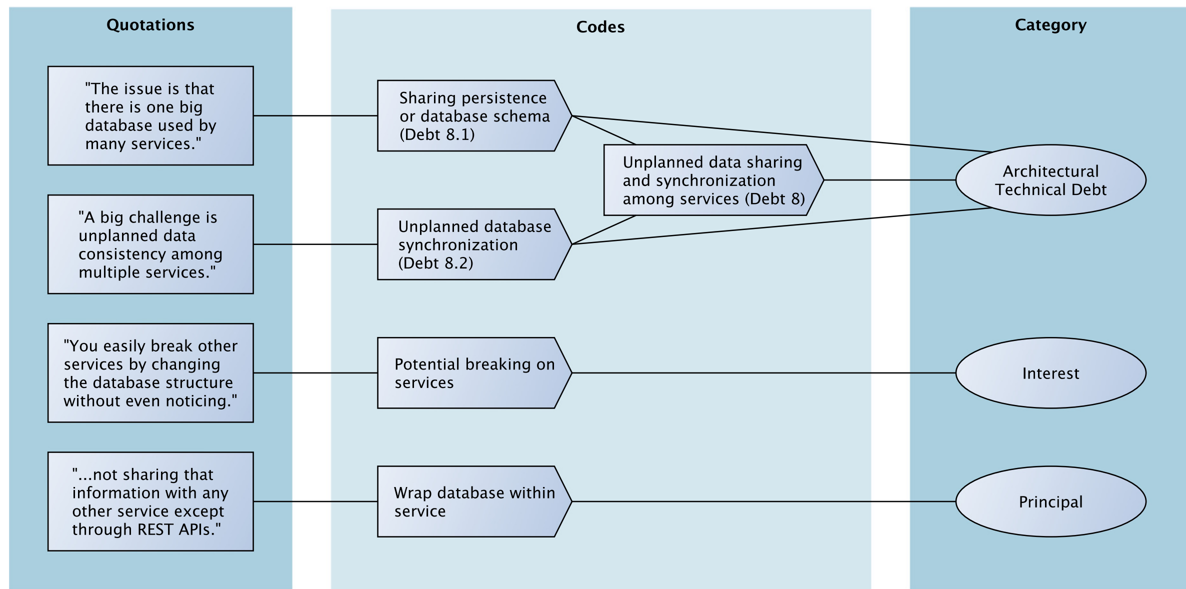


Fig. 9. Transforming quotations from practitioners into codes through open coding, and classifying them into categories. This is an example extracted from our analysis, so the number of the debts emphasized in the figure are references to our results in Section 4. The rest of the analysis is done in the same way.

Table 2
Type and number of interviews and interviewees by company.

C.	Num. Interviews	Interviewees	Interviewers	Type of interview
A	11	3 product owners/managers 2 architects 5 developers	1st and 2nd authors *(main interviews)* 1st author *(returning interview)*	Face-to-face
B	3	1 product leader 1 architect	1st and 2nd authors *(main interviews)* 1st author *(returning interview)*	Audioconference
C	4	2 architects 1 software engineer	1st and 2nd authors *(first 2 interviews)* 1st author *(third and returning interviews)*	Audioconference
D	3	1 product manager 2 architects	1st and 2nd authors *(first 2 interviews)* 1st author *(third interview)*	Audioconference
E	2	2 architects	1st and 2nd authors	Face-to-face
F	1	1 architect	1st and 2nd authors	Face-to-face
G	1	1 software engineer	1st and 2nd authors	Videoconference
TOTAL:				
7	25	22		

12 and added Questions 7, 9, 10, and 11 for the ensuing interviews. When all the cases were investigated, we returned to the previous companies to interview additional subjects. If they were not available, we asked for shorter complementary interviews (20–30 min) with the subjects we had met before. We covered newly discovered aspects in these interviews, as represented by Step 3 in Fig. 8. For example, a later interviewee clearly distinguished between internal shared libraries (produced by the team) and external dependencies (produced by external parties), such as frameworks and open-source software. Since the previous interviews did not clearly make this distinction, we returned to previous interviewees to ask for additional clarifications and details.

We also returned to previous interviewees to ask about newly discovered ATD items identified in later interviews. For example, we asked all the interviewees whether they perceived the heterogeneity of approaches caused by the services' implementation neutrality nature as harmful.

During all the initial interviews, two researchers were present. For four companies, where the distance did not allow us to have face-to-face interviews, we conducted the interviews using audio or video conferencing tools, as detailed in Table 2.

We did not follow Steps 3 and 4 of our methodology (Fig. 8 going back to the interviewees for additional input) for Companies F and G. We did not find information missing from the other contexts that required additional interviews.

The final interview guide is presented in Appendix.

3.3. Data analysis

Steps 2 and 4 in Fig. 8 show our data analysis, which was mainly performed using *open coding*, an approach that is part of *grounded theory* (Corbin and Strauss, 2015). Grounded theory is a rich systematic methodology that involves several other steps not followed in this study.

Open coding is usually the first step of coding in exploratory studies and aims to produce a set of concepts that fit the data (Corbin and Strauss, 2015). Fig. 9 presents a fragment of this analysis step: selected quotations in the transcriptions or audio recordings are flagged with a label (a code). Later, we found that, despite different wording, some findings were related to a more general topic coded at a higher level category, as in the example in Fig. 9. This last step allowed us to identify related ATD issues. Finally, we associated these codes to categories such as "ATD", "Interest", and "Principal" in a deductive manner.

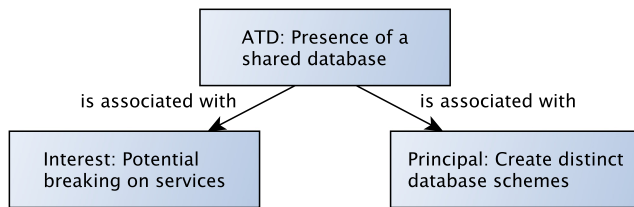


Fig. 10. Identifying the relationship among debt, interest and principal.

We performed the open coding phase by finding relationships between the codes in the categories above. Fig. 10 shows an example. The open coding provided us with three different codes: one for an instance of debt, another for an instance of interest, and the last for an instance of principal. After the open coding phase, we found that the interest and the principal were, respectively, the consequence and the solution for the debt (e.g., there were costs with multiple API versions (interest) because the APIs were poorly designed (debt), so they were grouped together). The set of codes and their relationships were used in our report phase that synthesized our results (Step 5, Fig. 8).

We performed the qualitative analysis with NVivo,[1] which tracks the links between codes created on top of the data and to the original quotations to which they were grounded.

Finally, we performed *member checking*, in which study participants could review the findings (Runeson et al., 2012) to increase reliability (Step 6 in Fig. 8). We sent out a summary of our findings to at least one interviewee in each company and asked for review and feedback. We updated our descriptions according to the comments we received (Step 7 in Fig. 8).

4. Results

Table 3 shows the companies' most critical ATDs and how they are distributed. For each company, an "X" in columns *D (debt)*, *I (interest)*, or *P (principal)* indicates, respectively, that the company has accumulated the debt and has identified (or paid/is paying) its interest and/or its principal. An empty cell in the table means that the interviewees did not report the related debt, interest, or principal for that company.

Table 4 shows the negative impact and the solutions for each ATD according to each company. As our respondents could not provide an actual numerical cost for the interest (i.e, the negative impact cost) and the principal (i.e., the solution's cost), we present the closest possible qualitative description of these costs that we could extract from our data. All the proposed solutions were applied successfully by practitioners in their projects.

The remainder of this section describes the identified debts and their interest and principals.

Debt 1: Insufficient metadata in the messages

An ATD is present when messages contain insufficient metadata. A data packet used in the communication through APIs, such as REST, may be considered messages. However, all data packets were sent through a message bus in our study. In such cases, the metadata is typically used to track messages and add other useful information at the cost of increasing overall message size.

This debt may be accumulated because developers want to keep the message lighter for performance reasons (Company A) or due to poor service planning (Company E).

[1] https://www.qsrinternational.com/nvivo/home.

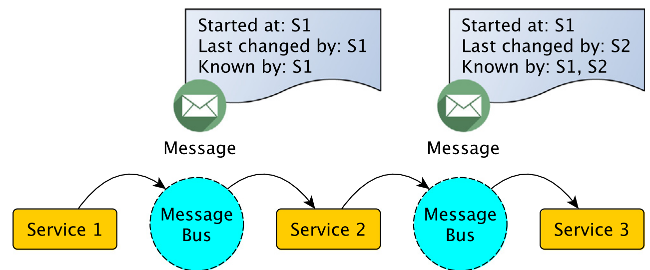


Fig. 11. A message carrying metadata that is updated every time it is consumed and forwarded by a different service.

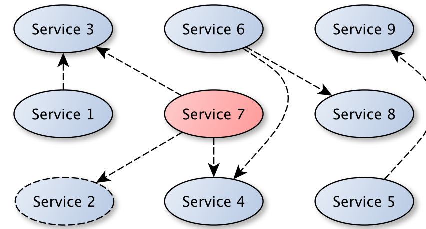


Fig. 12. In the absence of a tracking dependencies mechanism, it is impossible to know that, if Service 7 is no longer necessary, Service 2 can also be deactivated, but Service 4 cannot.

Debt 1.1: Insufficient message traceability

When messages contain insufficient metadata, developers might find it difficult to track the messages' source. Fig. 11 shows services that deliver messages through a message bus. If no traceability metadata is available, Service 2 consumes a message from the message bus, but the service does not know which service produced the message.

Interest. The primary issue is the impossibility of tracking dependencies among services. One interviewee exemplified this debt by saying that "there is a regulatory requirement to document the traceability of data to the data source, and the lack of metadata and a data dictionary make it difficult to fulfill this requirement". Fig. 12 presents a set of services and their dependencies. If Service 7 is no longer needed, Service 2 can also be deactivated because no other services use it, but Service 4 cannot be deactivated because it is used by Service 6. If it is impossible to track dependencies, it is not safe to deactivate dependent services, as in the examples mentioned above (Companies A and E). The number of services in the product will grow, and possible unused services such as Service 2 in Fig. 12 will remain deployed and consume resources (Companies A and E). Besides, it is impossible to track the messages' sources when necessary. This incurs costs related to, for example, data-tracing regulations (Company A). In such a case, a financing company that, by law, must track financial operations might find this impossible to do because no available metadata indicates the sources.

Principal. Some interviewees described the primary solution as adding service ownership metadata to the messages as the (Companies A and E). Fig. 11 gives an example of hypothetical metadata information attached to the messages. The metadata allows Service 3 to know (i) the service from which the message flow originated, (ii) the entire list of services that used that information, and (iii) the last service that changed the message. Company A went further and proposed defining such requirements with "the implementation of the canonical [data] model design pattern" to "ensure compliance with data traceability". A canonical data model is a design pattern in which there is agreement on and standardization of data definitions in different business systems (Hohpe and Woolf, 2012) to ensure that the services contain the required information.

Table 3
Architectural technical Debt identified on each company.

ID	Debt	Company A			Company B			Company C			Company D			Company E			Company F			Company G		
		D	I	P	D	I	P	D	I	P	D	I	P	D	I	P	D	I	P	D	I	P
1.	Insufficient metadata in the messages																					
1.1.	Insufficient message traceability	X	X	X										X	X	X						
1.2.	Poor dead letter queue growth management	X	X	X										X	X	X						
2.	Microservice coupling	X	X	X	X	X		X	X	X				X	X	X	X	X	X			
3.	Lack of communication standards among microservices	X	X	X																		
4.	Inadequate use of APIs																					
4.1.	Poor RESTful API design				X	X	X	X	X	X	X	X	X									
4.2.	Use of complex API calls when messaging is a simpler solution										X	X	X									
5.	Use of inadequate technologies to support the microservices architecture	X	X	X				X	X	X												
6.	Excessive diversity or heterogeneity in the technologies chosen across the system	X	X	X										X	X		X	X	X	X	X	X
7.	Manual per service handling of network failures when target services are unavailable				X	X	X	X		X												
8.	Unplanned data sharing and synchronization among services																					
8.1.	Sharing persistence or database schema							X		X	X	X	X							X	X	X
8.2.	Unplanned database synchronization							X	X	X												
9.	Use of business logic in communication among services	X	X	X																		
10.	Reusing third-party implementations																					
10.1.	Many services using different versions of the same internal shared libraries	X	X	X							X	X	X	X	X	X	X	X	X			
10.2.	External dependencies with various licenses requiring approval				X	X	X															
11.	Overwhelming amount of unnecessary settings in the services	X	X	X	X	X		X	X	X	X	X		X	X		X	X	X	X	X	X
12.	Excessive number of small products				X	X								X	X							

Debt 1.2: Poor dead letter queue growth management

A *dead letter queue* receives messages sent to a nonexistent or full queue and messages rejected for other reasons. A dead letter queue accumulates such messages to facilitate their inspection. The lack of mechanisms to control the dead letter queue's growth represents an ATD issue because the queue becomes "one place with lots of messages and no ownership".

Interest. Dead letter queues grow so quickly that inspecting the messages becomes impossible, so the queue consumes more and more resources. Cleaning up the queue without losing important information may be impossible due to the high number of dead messages. Besides, it may be difficult to predict the causes that lead such messages to be sent to the dead letter queue instead of their original destinations (Companies A and E).

Principal. The primary solution is to remove the dead letter queue, shifting responsibility for message deliveries to the services (Company A). That would help solve any issues that cause the queue to grow, such as services not caring about the accumulation of messages. If that is impossible, Companies A and E agree that there are still two possible solutions, as articulated by one of our interviewees: "You should distribute this [the dead letter queue] either by having ownership metadata on the message or having distributed queues". Therefore, adding metadata that identifies the messages' sources would make it easy to handle lost messages because it would be easier to track their creators. Splitting single dead letter queues into smaller queues managed by different teams helps "divide and conquer" the problem.

Debt 2: Microservice coupling

Microservices should be designed to be mostly independent of each other and have well-defined boundaries and interfaces. A lack of knowledge or negligence about good design practices may lead a company to accumulate ATD related to tightly coupled microservices. One of our interviewees said, "How do you decouple the dependencies is always something you need to work out". We found this debt more often in the products that use APIs.

Interest. The accumulation of this debt creates too many dependencies among teams, so delays from one team may affect the other teams' development plans by, for example, delaying deliveries. Besides, someone with access to all involved teams must handle the unnecessary coordination overhead. This was exemplified by one case in which three teams were developing "three very tight microservices", and there was a "need to tightly coordinate the work between them" (Companies A, B, C, E, and F). Another cost incurred by this debt is the cascading effect of spending time and effort updating and deploying many dependent services due to changes made to one service (Companies A, B, C, E, and F). If accumulating this debt is a common practice in a company, the number of services easily increases. Various services end up not being useful in diverse situations because they are too solution-specific (Company A). Finally, the debt may cause incidents into dependent services that are not updated as required (Companies B, C, E, and F).

Principal. Designing services to be generic and independent usually incurs a cost, which is the principal, as also mentioned by Company A. Company E reduced the accumulation of the aforementioned debt by "being API first", which requires teams to consider what the service is instead of focusing on the code. According to the interviewee who made the suggestion, "It is more about orchestrating APIs", and "what's behind an API is no longer relevant". Company C considered using internal training about developing good APIs to mitigate the problem. Company F proposed setting aside some time slots during development to continuously clean, refactor, and improve the APIs.

Debt 3: Lack of communication standards among microservices

When autonomous teams do not have proper guidelines or standard models for creating APIs or message formats (depending on how their services communicate with other services), the company may accumulate a debt in which many APIs or message formats emerge from the various teams because, as stated by an interviewee from Company D, "each message producer of messages is left to define the format of the data themselves". While this debt might not be a problem in small systems, especially because microservices allow teams to decide their standards, having many standards will incur additional costs and become an issue. This is the "Tower of Babel problem" in our previous work (de Toledo et al., 2019). We found evidence of this problem in Company A's use of messages.

Interest. Developers often exert unnecessary effort when translating messages among distinct formats to allow different services to communicate. According to our sources, this debt leads to "data duplication, lack of consistency, and unwanted complexity". The solution becomes overwhelmingly complex due to too many API or message formats. Each time one service must interact with another, the team that develops the first service must learn a new message or API format to define the proper translations (Company A).

Principal. According to Company A, "this problem is typically solved in organizations by using the canonical [data] model design pattern". The implementation must be properly policed, or this model may also become complex and costly.

Debt 4: Inadequate use of APIs

Poor API design (i.e., failing to properly plan the API interface, error codes, etc.) may be the easiest way to have working code initially, but this has negative effects. In some cases, APIs are used in situations where other solutions, such as messaging, would be preferable. Such situations constitute the debt of inadequate API use. We present details below on each of those situations.

Debt 4.1: Poor RESTful API design

When using RESTful APIs, several conventions address their readability and use. For example, the REST Uniform Interface standardizes implementing *create*, *retrieve*, *update*, and *delete* operations in a resource. HTTP also includes a list of status codes[2] that should be used in the API's responses. In our study, some developers tried designing RESTful APIs without following the proper conventions because they "were still focusing on the functional part of the job", resulting in a poor design (Companies B, C, and D). An example of this problem was using operations that should have performed a resource update but instead retrieved a collection of items (Companies B, C, and D).

Interest. Poor API design causes several issues: (i) the API is difficult to use because its results may not follow expected conventions (Companies C and D); (ii) APIs are not stable, as one interviewee emphasized: "Should I make any changes, I need to make a new revision of the API" (Companies C and D); (iii) the API's instability requires the creation of new API versions and the need to maintain the old and deprecated versions (Company B); (iv) such changes also make it difficult to maintain backward compatibility in new versions of the API (Companies C and D); as a final result, (v) intentional and unintentional breaking changes[3] are common (Companies C and D).

[2] https://tools.ietf.org/html/rfc2616#section-6.1.1.

[3] A breaking change is a change in one part of a software (e.g., in a microservice's API) that potentially causes incompatibility with other components, causing failure. Examples of breaking changes for an API are changes in the response codes (e.g., *200 OK* to *201 Created* in HTTP) and renaming the location of the resource (e.g., renaming */user* to */users*, so previous clients are not able to find the related resource anymore).

Table 4
Catalog of architectural technical Debts, interest and principal.

ID	Architectural Debt	Consequences (Interest)	Solutions (Principal)
1.	Insufficient metadata in the messages		
1.1.	Insufficient message traceability	**A, E:** impossibility to identify and deactivate services that are not necessary anymore **A:** impossibility to track the source of messages, incurring costs with, for example, data tracing regulations	**A, E:** add services ownership metadata to the messages, allowing identification of their source **A:** implementation of a Canonical Data Model that ensure compliance
1.2.	Poor dead letter queue growth management	**A, E:** impossibility to identify the source of messages and determine the causes of the message loss in the dead letter queue	**A:** removal of the dead letter queue and to move the responsibility of the message deliveries to the endpoints **A, E:** add metadata to identify the source of the messages **A, E:** splitting the dead letter queue into smaller queues, managed by different teams
2.	Microservice coupling	**A:** increasing amount of unnecessary services **A, B, C, E, F:** too many dependencies among teams, creating coordination overhead; cascading changes in service consumers when producers are updated; eventual breaking on services **B, C, E, F:** eventual incidents in not updated services	**A:** use some time to design generic and independent services **C:** internal training about API development **E:** use of an API-first approach while designing services **F:** considering slot for continuous API improvement during development
3.	Lack of communication standards among microservices	**A:** cost with translations among services; overwhelming amount of message formats for developers	**A:** ensure standardization with a Canonical Data Model
4.	Inadequate use of APIs		
4.1.	Poor RESTful API design	**C, D:** APIs are not stable, with frequent breaking changes, hard to use and frequently not backwards compatible **B:** instability demands the creation and maintenance of multiple API versions	**B, C:** additional effort to stabilize the API and avoid changes in the future **C:** management of API versions; tracking of internal and external consumers; definition of clear deprecation strategy **D:** definition of a standard for the APIs
4.2.	Use of complex API calls when messaging is a simpler solution	**D:** additional coupling among services; tests are inherently complex	**D:** redesign of services using a messaging approach
5.	Use of inadequate technologies to support the microservices architecture	**A:** big latency in the services communication; need of a dedicated team to maintain the third-party tool **C:** impossibility to provide some functionalities	**A, C:** proper planning about the technology and migration as soon as possible
6.	Excessive diversity or heterogeneity in the technologies chosen across the system	**A, E, F:** some services cannot communicate each other **E, F:** developers cannot easily migrate to other teams **A, F:** resistance to change technologies later **G:** developer velocity slows down, need to maintain distinct tools and additional source code repositories	**A, F:** limiting the set of technologies used by the teams **G:** use of language specific mono-repositories and incentive their use for related projects: related software written in the same programming language are more likely to use the same tooling
7.	Manual per service handling of network failures when target services are unavailable	**B:** extra cost on maintaining additional complexity in the architecture	**B, C:** use of third-party products (e.g., circuit breakers) that provide such mechanisms **B:** use of a service mesh
8.	Unplanned data sharing and synchronization among services		
8.1.	Sharing persistence or database schema	**D, G:** potential breaking on services **D:** complex database schema and difficulty to track services using the data	**C, D:** having separated databases for each service **C:** creation of distinct database schemes for each service inside the same database **G:** wrapping of the database within a service, preventing direct access
8.2.	Unplanned database synchronization	**C:** synchronization issues may be visible to users	**C:** the solution is context dependent, depending on the problem, a shared database might be needed, or a more complex transaction mechanism must be implemented
9.	Use of business logic in communication among services	**A:** unnecessary cost to maintain business logic in the communication layer	**A:** moving such business logic to the services, keeping the communication layer as thin as possible

(*continued on next page*)

Principal. Our interviewees suggested that APIs be standardized (Company D), versioned (Company C), and kept as stable as possible so that changes in the services do not affect their APIs (Companies B and C). Our interviewees also suggested using an explicit deprecation strategy. In other words, after the deprecation announcement, a previously defined period of support should not be extended (Company C). It is also useful to track internal and external consumers. Hence, it is possible to contact

Table 4 (*continued*).

ID	Architectural Debt	Consequences (Interest)	Solutions (Principal)
10.	Reusing third-party implementations		
10.1.	Many services using different versions of the same internal shared libraries	**A, E:** costs to plan and update all related services; several services may not be updated and multiple versions of the library should be maintained **A:** cost of issues generated by refusal to adopt by early adopters **D:** cost to handle breaking changes **F:** dependency between service and library developer teams	**A, D, E, F:** reduction in the use of shared libraries **D:** replication of simple code, creation of services to perform complex code functionalities
10.2.	External dependencies with various licenses requiring approval	**B:** delays while waiting for approval of new libraries	**B:** investing in a process to evaluate and approve external dependencies as fast as possible
11.	Overwhelming amount of unnecessary settings in the services	**A, B, C, E, F:** complex environment **D, G:** unexpected issues after deploy	**A, G:** creation of repository for configuration settings **A, F:** reducing of the amount of configuration settings on services **G:** requirement of peer approval before accepting changes on settings **C:** creation of a configuration server to automate deploy of configuration settings
12.	Excessive number of small products	**B, E:** governance on multiple projects instead of one (if a monolith)	Our study reveals no solution to this problem

them directly and request migration to a new and more stable version of the API (Company C).

Debt 4.2: Use of complex API calls when messaging is a simpler solution

There are situations in which an asynchronous communication approach is particularly appropriate, such as executing long-running jobs. In other situations, such as when one service needs an immediate response from another after updating a user's address, synchronous communication is a better choice. A REST call is synchronous, whereas the messaging approach is asynchronous. Using REST when messaging is more appropriate constitutes another architectural debt because there are costs associated with using an improper solution. We found a specific instance of this problem in Company D, which described "rather complex service calls back and forth where messaging would have been a much better solution and allowed for much better testing".

Interest. Instead of preparing services to respond to events (e.g., when a message arrives), API endpoints were created for each instance of communication among the services (Company D). Such a situation increases coupling among the services due to "a relatively complex handshake between two different services". In other words, all the involved services depend directly on each other's API endpoints instead of simply triggering an event (message). Besides, the services are harder to test due to the aforementioned complexity. Thus, the costs of maintaining such services increase.

Principal. According to Company D, the primary solution is "moving completely, for these particular cases, to a message passing" approach. The messages should be generic enough to be used by all the involved services without complex processing.

Debt 5: Use of inadequate technologies to support the microservices architecture

Technology choices may positively or negatively affect software architecture. Technologies used in microservices are different from those used in other architectural styles (for example, those ones used for service discovery and circuit breaking, as well as others discussed in Section 2.1). There are certainly technological similarities with other SOA approaches, but however, there are also differences (e.g., ESBs should not be used in microservices, as discussed in Section 2.1). For example, different cloud providers support different sets of tools and technologies, such as operating systems and storage software; some architectural choices simply do not work with them or face limitations. One interviewee said, "The technological base that we built a platform upon was not the best choice for what we wanted to offer". Therefore, selecting an inadequate set of technologies, such as a Platform-as-a-Service or Infrastructure-as-a-Service provider that does not support the technology required for the software architecture incurs a debt.

Interest. This debt's interest is context-dependent. In our findings, choosing the wrong technology as the message bus responsible for transferring messages among services caused considerable latency in such communication, with the consequent costs of requiring a team to maintain the third-party tool instead of working on other priorities (Company A). Company C could not provide certain new features because the previously selected platform did not provide enough sufficiently managed services. The company had to deal with the costs of implementing the required functionalities without the availability of proper technologies.

Principal. Companies A and C reported that the primary solution should be planning the architecture before selecting the technologies because it is harder to change later on. Since that was impossible, they migrated to more appropriate technologies as soon as possible to prevent the interest from growing.

Debt 6: Excessive diversity or heterogeneity in the technologies chosen across the system

Selecting programming languages and related technologies are architectural choices that must be made in any project. Microservices give developers the freedom to choose different tools, programming languages, communication technologies, API standards, messaging technologies, and other technologies for each service. Although this is an advantage because some languages, frameworks, and technologies are more appropriate to specific tasks, such freedom can also lead to the interest described below. Such freedom may lead to debt, causing the company to have an excessively diversified environment.

Interest. We found the following set of issues in four companies: (i) Services that use one technology cannot communicate with services that use another, such as cases with REST APIs on one side and messaging technologies on the other (Companies A, E, and F). When services that use distinct technologies must

communicate, a workaround must be developed. (ii) Individuals from one team cannot easily migrate to another because the necessary skills are quite different (Companies E and F). (iii) It is difficult to change the technology choices later because the services run for a long time and the teams become accustomed to their own choices (Companies A and F). Finally, (iv) developer performance decreases when teams must maintain distinct tools with potentially divergent setups (e.g., languages or environments) and when there is a need to maintain additional tooling source code repositories (Company G).

Principal. The primary solution reported by the companies was limiting the set of technologies available to the teams, especially those technologies used by multiple services and teams (Companies A and F). When talking about limiting the set of available technologies, one of the interviewed practitioners said, "I don't want too much alignment on that, but we need to keep the complexity under control and work to minimize the complexity of all parts". Besides, using programming language-specific monolithic repositories[4] allows related software (e.g., different microservices for distinct payment methods such as cash and credit cards) written in the same programming language to share tools from the same repository. Such repositories encourage using the same deployment tools, which may prevent a surge in different setups and support the merging of services because they use the same technologies (Company G).

Debt 7: Manual per service handling of network failures when target services are unavailable

When distinct microservices communicate synchronously (Section 2.1.1), they usually contact each other through the network. Unfortunately, several issues can occur during such communication. The communication channel may be overloaded, or the target service may be unavailable for many reasons, such as a lack of resources, crashes, or timeouts. One well-known approach is to create a mechanism within the service to retry contacting the target a fixed number of times. However, this approach has proven to be a debt because "every single developer has to think about how to handle that case themselves", leading to the costs described below.

Interest. Our interviewees mentioned that implementing the retry mechanisms manually on each service increased the code's complexity because developers must decide how the service handles the situation: "Should it retry, give up, or send a signal error? What should it do?" This approach increases the services complexity (and related maintenance cost) and increases the architecture's overall complexity (Company B).

Principal. The companies that reported this problem suggested reducing the complexity by using third-party products that provide features with retry mechanisms, such as circuit breakers (Section 2.1.3) (Companies B and C). One interviewee from Company B declared, "That is why we introduce the service mesh: to simplify that [the complexity created by developers when they must think about handling the retries]". Service meshes are introduced in Section 2.1.4 and usually contain circuit breaking mechanisms. Still, they should be used carefully and only when needed because they may cause an overhead, especially if the team does not have experience with them.

[4] This is also known as monorepos; it is a single source code repository for storing many projects.

Debt 8: Unplanned data sharing and synchronization among services

Microservices may have their own databases, but they can also share or synchronize data with other services. In such situations, the company may incur the debt of not planning data sharing or synchronization among services, leading to various costs. This debt could be interpreted as a way of causing coupling among services (Debt 2). Still, we consider it another because fixing it does not necessarily solve the coupling issue, nor do the fixes we present for solving Debt 2 solve the databases' issues.

Debt 8.1: Sharing persistence or database schema

Sharing the same persistent storage or database schema with multiple microservices is a critical architectural debt that can easily lead to high costs. For example, one company recognized the following situation: "We still have a common database today that is a technical debt that we are aware of, and we will have to get rid of this common database".

Interest. A service may require changes in the database schema or the data stored in the database. Such modifications may potentially break other services that use the same schema: "You easily break other services by changing the structure without even noticing it or noticing it too late" (Company D and G). If using the same database schema for different services is common in the company, an unknown number of services may use the same database schema. Therefore, it is difficult for a development team to know whether other services use the schema due to the lack of tracking for such information. This increases the odds of breaking other services (Company D). In such cases, the database design is complex and contains information from multiple services (Company D).

Principal. The ideal solution is to use separate databases for each service: "We are trying to split up that common database so that each service is responsible for its own database" (Companies C and D). It is also possible to wrap the database in a service, exposing it through an API instead of requiring direct access (Company G). Although the last solution does not solve the database complexity problem (i.e., the database still contains data from multiple services, which is more complex than having separate databases), it does reduce direct database schema manipulation. If wrapping the databases in this way is impossible for any reason (e.g., the services are business-critical and the migration cannot be done at once), our interviewees suggested using different database schemas for each service to enable the services to be changed independently (Company C).

Debt 8.2: Unplanned database synchronization

Microservices increase the likelihood of having distributed databases, but they may require synchronization. However, one interviewee said, "A big challenge is data consistency in use cases involving multiple services and multiple databases, which must be somehow consistent or aligned to fulfill the use case successfully". In such a situation, the company may incur the debt of improperly planned synchronization.

Interest. The software composed of multiple microservices will remain inconsistent until all related databases are updated. This can lead to bugs. An interviewee from Company C said, "There were cases in which we had features that required us to basically align three databases to show the right information on a [user's] dashboard". Because we cannot present the real use case due to confidentiality restrictions, we explain the problem using a fictitious example: an online bookstore has 10 copies of a particular book. The store is developed using microservices and contains a microservice for purchases and a microservice for managing its inventory. Two users purchase the same book simultaneously, one for a single copy and the other for all 10

copies. Both users pay for their orders simultaneously, but the one who is buying all 10 copies finishes first. When the user buying a single copy finishes the payment, there are no books left because there is no mechanism to synchronize the inventory management and purchase services in this example. Company C encountered a similar problem. The costs vary depending on the business criticality of the affected product features.

Principal. Company C reported that distinct solutions could be considered depending on feature criticality. In some cases, a database must be shared by the services (see Debt 8.1), or a complex transaction mechanism must be planned, but no approach to implementing such a transaction was reported. We found no similar problems and solutions in other companies.

Debt 9: Use of business logic in communication among services

Microservices encourage the use of dumb pipes (i.e., simple message routers) for communication. The communication layer used by microservices should not include business logic. However, in projects like one developed by Company A, "the data transported changed within the communication channel itself". The changes are made by the services communication channel using business logic. Using business logic in the services communication layer constitutes an architectural debt because such a logic is not supposed to exist.

Interest. Maintaining additional business logic apart from the services is costly, as any changes to the services may also require changes to the communication layer where the business logic is located. Besides, "each time a new system is on-boarded, you need to set up the communication flow, requiring the communication channel team to provide the flow and possibly set up some business logic". In other words, an external team—the communication channel maintainers—must understand details about how the related services work to implement the business logic (Company A).

Principal. Our interviewees suggested moving all business logic to the services themselves, thus allowing the communication layer to act as a dumb pipe. The costs involved are related to implementing such logic on each service: each team must understand and implement the changes independently (Company A).

Debt 10: Reusing third-party implementations

Reusing code can reduce resources while programming. In software development, reuse may occur by developing libraries used in various microservices. We call them *shared libraries* in the context of microservices because various services usually share them. On the other hand, reusable code—such as frameworks, language extensions, and libraries—may also be developed by external parties. We call codes from external parties *external dependencies*. For a single microservice, shared libraries and external dependencies can both be considered third-party implementations. We found evidence that using such third-party implementations may be an architectural debt and that the interest differs depending on whether they are shared libraries or external dependencies.

Debt 10.1: Many services using different versions of the same internal shared libraries

Companies may develop their own libraries to reuse code. Such libraries can act as black boxes for complex operations, and other reasons exist for using such libraries. One company states, "You could use REST, but if you want to be efficient, you want a native binding because it is faster". However, these libraries may constitute architectural debt if many services use such libraries and cause the interest described next.

Interest. Several negative impacts must be considered: (i) New releases of the libraries may require updates on every service using them. A roadmap must be established to handle such changes (Companies A and E). (ii) Several versions of the libraries must be maintained, because replacing old versions in all running services might be impossible for reasons like development priorities: "Sometimes the clients are business-critical and, in their roadmap, upgrading to a new version of a library it is not the top priority" (Companies A and E). (iii) Early adopters may refuse to implement new versions, especially if breaking changes exist (Company A). (iv) If library use cases are frequently unknown, breaks may occur due to unexpected situations. Such breaks lead to fixes that may lead to breaking changes when libraries are updated (Company D). (v) The service's developers using the library depend explicitly on the team that is developing the library, so delays in releasing library versions with some required functionality will likely affect the service's developer team (Company F).

Principal. All the companies that mentioned the problem agree that they should avoid and discourage using shared libraries as much as possible (Companies A, D, E, and F). Company D suggested that a complex shared code should be transformed into services and that more straightforward codes should be duplicated by the different teams. Several practitioners suggested considering exceptions only when no better alternative exists "to keep the amount of shared libraries as minimal as possible" (Companies A, D, E, and F).

Debt 10.2: External dependencies with various licenses requiring approval

External dependencies are any libraries, frameworks, or similar software developed by external parties. We found evidence that their use might lead to architectural debt when the types of licenses allowed to be used by the company are strictly limited. Many products depend on some externally developed software; not accumulating this debt is almost impossible, but steep interest can be avoided.

Interest. All third-party codes' licensing limitations must be documented: "We need to document whether they are exportable in order to be able to perhaps include them in the main application and send them to trial, or even in order to be able to run them in a public cloud because that is also an export from one country to another". Eventually, some dependencies must be replaced due to non-compliance with regulations. Licenses may limit business models (e.g., it may not be possible to sell the product or service) and even prevent the software's distribution to some countries. Due to the high risk of regulation issues, approving such dependencies may be time-consuming and cause delays (Company B).

Principal. No current approach exists to handle this issue other than investing in a process to evaluate and approve such dependencies as fast as possible. Company B suggests that teams not use external dependencies whose licenses were not approved in advance to avoid this issue.

Debt 11: Overwhelming amount of unnecessary settings in the services

Microservices can be reused and deployed in various settings by tweaking parameters. One company explained the situation: "Microservices tend to expose some configuration settings that can basically be overridden. So, you can set a default value, and then whoever is using or deploying your microservice can override it at deployment time. When we add many microservices together and aggregate them into big settings trees, handling these

kinds of parameters becomes very overwhelming". For instance, allowing parameters to be overridden in different environments adds some control over resource usage. However, the company may incur debt caused by many unnecessary configuration settings among the services, which leads to higher maintenance costs.

Interest. Managing many parameters takes time and leads to overhead while deploying products (Companies A, B, C, E, and F). Such debt also increases the likelihood of unexpected issues caused by wrong setting values: "It was a very frequent problem that deployments would go wrong because somebody changed something in deployment scripts, and it would take a while to figure out why it was going wrong". The greater the number of values, the greater the likelihood of mistakes. More value combinations will also need to be tested. Mistakes require time to fix, and more tests will require more development time (Companies D and G).

Principal. Some interviewees suggested creating a repository to keep the configuration settings, usually managed by a control version system (Companies A and G). Besides, peer approval for every change in the settings repository before production helps prevent issues (Company G). Interviewees also suggested reducing the number of configurable settings on each service to the minimum necessary (Companies A and F). A configuration server to automatically apply changes from the settings repository to the deployments also simplifies managing the settings (Company C).

Debt 12: excessive number of small products

By definition, a microservice is a small product with a single capability and its own life cycle from development to deployment. It includes documentation and any governance required for software products. For example, one company reported that "each of the microservices effectively became a very small product, and we have a full process for handling products". This may lead to debt via an excessive number of small products.

Interest. Each microservice must be governed separately—which requires overhead with dedicated management, development, deployment, and maintenance. This does not exist with monoliths because they require only one deployment (Companies B and E).

Principal. Our study reveals no solution to this problem. The companies reported that they "haven't really found a good way to change our standard ways of working and documenting to handle that yet", and "these kinds of aspects are a bit difficult when you have very small units of software flowing all over the place".

5. Discussion

This section discusses our ATDs in different contexts, how they relate overall, how to avoid them, and the limitations of our study.

5.1. Debts in different contexts

We discuss the various ATDs in relation to the company application domain, the project stage, and other specific context factors. We aim to help practitioners understand, adapt, and apply our results to their specific contexts.

Difficulties with message traceability (Debt 1.1) affect financial systems more than other product types. Poor management of dead letter queues (Debt 1.2) was more common in companies that migrated from older approaches. Problems with coupling (Debt 2) affect most companies in this study; only Company G did not report critical coupling issues among services, but we have only one interviewee from this company.

The lack of communication standards among services (Debt 3) occurred more frequently in applications with many services; smaller products seem to avoid this problem (Table 1 shows each project's number of microservices). However, this conclusion requires more investigation because Companies F and G, which have many services, did not report it as a problem.

The inadequate use of APIs (Debt 4) is a debt that requires attention every time a new API is created or updated. Not all the companies reported it, but it affected most companies using APIs in our investigation (some other companies used primarily messaging approaches or a balance between APIs and messaging). Our interviewees did not discuss techniques to design good APIs. We focused on reporting our findings from the interviews. However, we believe that the API design plays an important role in fixing this debt. See, for example, the work of Mosqueira-Rey et al. (2018), in which they present a systematic approach for developing usable APIs. There might be a relation between this debt and Debt 3 (lack of communication standards among microservices), although it is not apparent from our data. Inadequate technology use (Debt 5) is usually harder to fix because, for example, changing an entire platform is hard. Many parts of a project may depend on chosen technologies, so changing them may require an entirely new project. Therefore, Debt 5 requires more attention at the project's beginning, when technologies are being chosen.

One solution proposed for solving the overwhelming diversity of technologies (Debt 6) is limiting the number of technologies used. This might contradict the definition of microservices because they are supposed to be independently deployable units that give practitioners the freedom to choose the technologies to use. However, the diversity should be reasonably limited in practice because it may lead to various problems, such as difficulties migrating developers to other teams in a company and a decrease in a team's performance because of the need to maintain potentially divergent setups. Additionally, despite monolithic repositories being suggested as a solution for some cases of this debt, there is a risk that using them for many microservices might lead to overloaded repositories. Therefore, a good practice may be using monolithic repositories with smaller subsets of microservices and only when such microservices share the same technology stack.

Manually handling retries while trying to communicate a temporarily unavailable service (Debt 7) may also require attention. This debt was only found while using APIs because the messaging technologies do not require direct access to another service—only to the message bus.

Issues with shared data (Debt 8) were rare, but their effects were some of the most dangerous ones (e.g., breaking other services). They require attention, especially while designing new services. Note that Debts 8.1 (Sharing persistence or database schema) and 8.2 (Unplanned database synchronization) are highly interconnected: Debt 8.2 might result from splitting a database, for example, from solving Debt 8.1. On the other hand, sharing a database among services, which is a solution proposed by some of the interviewees for Debt 8.2, may incur Debt 8.1. The last case can be avoided by using different database schemas.

Business logic in the communication layer (Debt 9) is dangerous, especially for legacy systems, and companies developing new products seem to be aware of this issue and are avoiding it successfully.

Misusing shared libraries (Debt 10.1) may generate high costs. Therefore, we argue that they should be used carefully and only when needed. Our interviewees did not report the same issues for shared libraries while discussing external dependencies, such as frameworks. This study does not identify the reasons for differences among shared libraries and external dependencies. Issues

with external dependencies (Debt 10.2) seem to relate to licensing and are restricted to companies with multiple deploys of the same software in various regions worldwide.

A common problem for every project using microservices involves many configurable settings in microservices (Debt 11). All the companies reported that the services' deployment is complex and error-prone because of the number of settings to define.

The excessive number of products (Debt 12) is hard to avoid. It may relate to the services' granularity or other unknown factors. It might simply be a drawback of using microservices. Thus, this problem must be investigated further.

Many of the debts described here may be found in other architectural styles. For example, poor RESTful API design (Debt 4.1) may be found in monoliths. However, they may be different in microservices because each service is a separate application, which adds to the costs. A monolith, for example, may expose a single API, whereas the same functionality might require the existence of multiple APIs in separate applications, with multiple points of failure instead of a single one. Also, microservices use several additional technologies not applied in other architectural styles. We introduced a general description of those technologies in Section 2.1. Many of those technologies were originally developed or adapted to support SOA, but others were created specifically to support microservices, as discussed in Section 2.1.

Other approaches that have been proposed in gray literature can help manage ATD. For example, regarding Debt 8.1, Newman (2019) suggests using database views and creating a wrapping service. Newman (2019) also discusses other patterns that we could not match with our findings, including migration patterns, user interface composition, and database synchronization. Although such patterns might have been used, they were not mentioned by our respondents.

5.2. Common interests and principals among the debts

Understanding how debts are related is essential to better plan and analyze the consequences of refactoring. Fig. 13 illustrates the debts, their interests and their principals. It shows which interests and principals are common among the debts. It was built by grouping the interests described in Table 4 into higher-level categories. Then we connected all the debts to these high-level interest categories to show which debts generated similar interests. For example, the extra work for handling cascading changes and a higher number of unnecessary services from Debt 2 and the overwhelming amount of message formats for developers to handle from Debt 3 were both categorized into "development overhead". Some interviewees explicitly stated some relationships; the researchers inferred others through the cross-case analysis.

Finally, we repeated the same procedure for the principals, but we found only two principals shared between debts: the metadata standardization and the design of generic services.

Development overhead is caused by nine out of the twelve debts we reported (or eleven of sixteen, considering the sub-debts). Thus, it is not possible to reduce such development overhead without investing in repaying many different debts. The developers are the ones most affected by such overhead.

Potential breaks within the services are caused by four debts (Debts 2, 4, 8, and 10). Investing in paying those debts might reduce the probability of having to deal with instability and cascading failures.

Dependencies among teams are caused by three debts (Debts 2, 4, and 10). Such dependencies contribute to delays in the project as well as productivity loss because the developers on one team must wait for another team to start their work. Similarly, team velocity reduction is caused by three debts (Debts 4, 6, and 10). It may be advisable to pay extra attention to Debts 6 and 10 because they are responsible for many different interests.

Fig. 13 also shows other specific interests and principals that are related to only one or two debts.

We expect a mapping like the one shown in Fig. 13 to help practitioners to focus on the debts they want to manage according to the most costly issues they perceive in their projects.

5.3. Microservices coupling

We have found evidence that four debts (or six, considering the sub-debts) were indirectly increasing the probability of microservices coupling (Debt 2) and its consequences. Debt 3 (Lack of communication standards among microservices) leads to an excessive number of APIs or message formats, increasing the likelihood of having formats designed as coupled by default. As a result, the involved services cannot communicate with ones other than those originally developed for such interaction.

The main consequence of Debt 4 (Inadequate use of APIs) is having a set of services whose boundaries are not well defined, leading to the creation of functionalities in the wrong services. Thus, some services that should work independently must work together (i.e., they are coupled because some functionality required by one of them is in another service).

Debt 8 (Unplanned data sharing and synchronization among services) may cause an indirect coupling through the database. Changes in the database triggered by some change or additional functionality may eventually cause breaks in the other services. The services that share the database depend on each other's database structure.

Finally, Debt 9 (Use of business logic in communication among services) might lead to coupling because the logic is usually coupled to the services it is designed for.

In our study, coupling seems to be a common issue to be handled by microservices developers. It was mentioned several times by developers not only as a debt itself as described in Debt 2 but also as an indirect consequence of other debts. Investing in repaying the four debts (in addition to Debt 2) above might reduce the probability of having coupling among services.

5.4. Approaches for avoiding ATDs in microservices

Some techniques described in the non-peer-reviewed literature may help handle some ATDs reported in our research. Two techniques that may connect to two ATDs reported by the companies in this work are Domain-Driven Design (DDD) (Evans, 2004; Garcia-Molina and Salem, 1987).

One of the most significant challenges of using microservices relates to boundaries between them. The wrong boundaries may increase coupling among microservices (Debt 2.1). DDD is a powerful approach for defining microservices boundaries, but using DDD off the shelf may be hard in practice. It deserves more research, however.

To remove Debt 8.1, some of the studied companies chose to separate databases. None of the interviewees shared details about approaches they might have been using to avoid inconsistencies in the data. However, three approaches are: using the *eventual consistency* model (optimistic replication) (Vogels, 2008), *sagas* (Garcia-Molina and Salem, 1987), and *two-phase commits* (Reddy and Kitsuregawa, 1998).

Vogels (2008) explains that a system using an eventual consistency model does not guarantee that subsequent accesses will return the last updated value right away, but eventually all accesses to the data will return the last updated value. The delay in the consistency is a trade-off versus high availability in distributed systems.

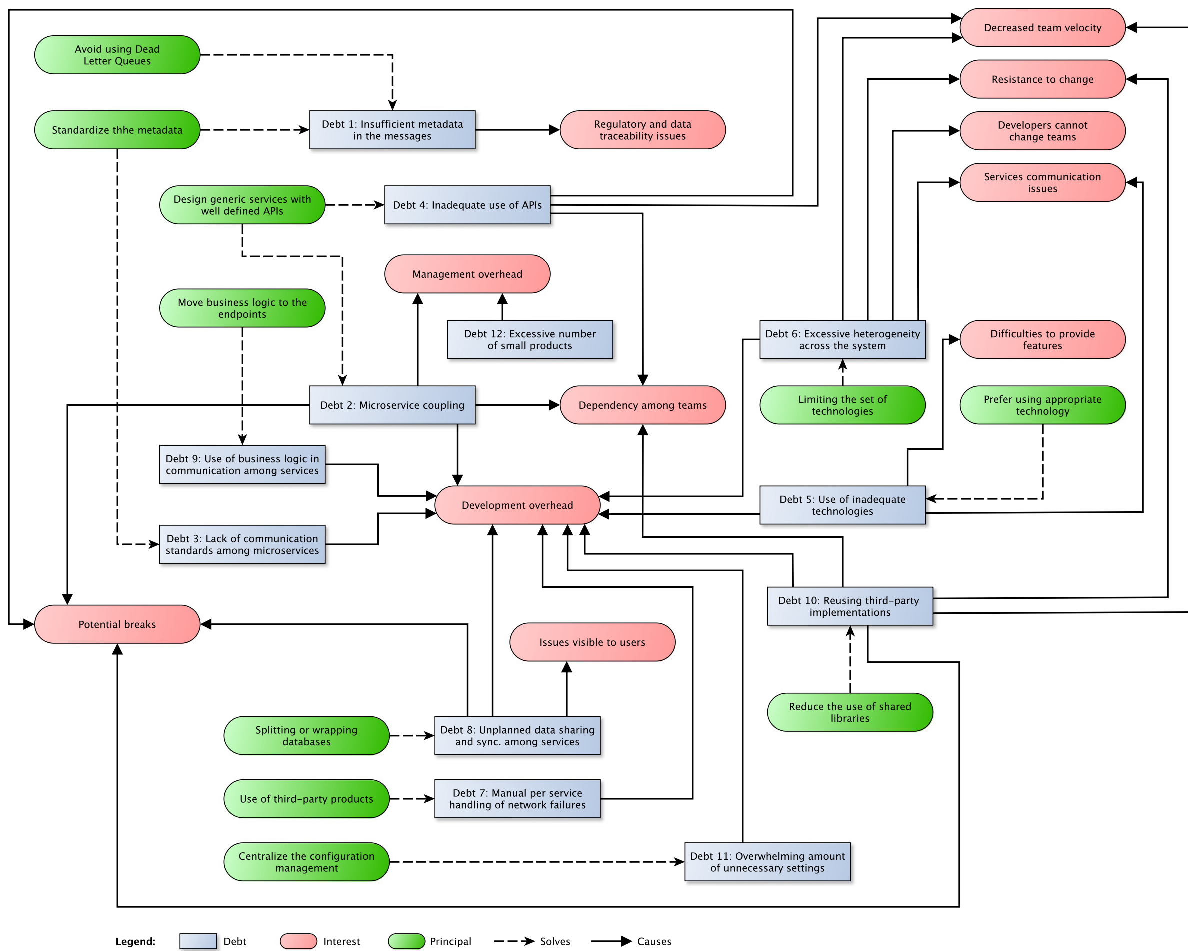


Fig. 13. Common interests and principals among the debts.

Garcia-Molina and Salem (1987) proposed sagas in 1987. Today, the technique is adapted to microservices. Sagas involve implementing transactions among the services via a sequence of local transactions in several services (a saga). If one transaction fails, a set of compensating transactions may restore the databases' previous state.

Two-phase commits are discussed in distributed database research (e.g., Reddy and Kitsuregawa, 1998). This technique consists of preparing the commit (Phase 1) and storing it permanently (Phase 2) after an agreement from all involved parties. Newman (2019) argues against the use of two-phase commits mainly because of the existence of a window of inconsistency that might lead to problems. When the data must be in two different places, Newman (2019) suggests using sagas instead. Regarding solutions for ATD, both approaches require further investigation.

5.5. Microservice architecture maturity

Microservice architecture is still maturing. We notice that practitioners' different understandings about what comprises a microservice lead to different practical decisions. On the other hand, SOA is a more mature concept supported by several standards, such as the family of WS-* standards for web services. Microservices are a way of implementing SOA, but there are no standards such as the WS-* to guide microservices' implementation, making practitioners less supported with best practices.

Large companies tend to have heterogeneous and experienced staff members with distinct backgrounds. Such a variation among staff members may generate different opinions about solving the same problems. Because emerging architecture's definitions and related technologies are still maturing, the different opinions increase practitioners' likelihood of struggling with ATD in such environments. Consequently, knowing about frequent and costly ATD in microservices is important.

5.6. Limitations

The sample size of respondents from companies is limited, particularly from Companies F and G, with only one interviewee each (see Table 2). This might create bias because we could not triangulate the data via additional perspectives from other participants from the same company (source triangulation).

Furthermore, the fewer people we interviewed in a company, the fewer smells may have been found. Note that not finding a particular debt does not mean the debt does not exist—only that we were unable to find it.

Also, we selected our interviewees through convenience sampling. Thus, the debts we found may not be representative of the respective companies.

Returning to the interviewees multiple times may make them biased during the process, either in favor of the study or in favor of (or against) their system. Still, the number of return interviews with the same interviewees was low, and we primarily asked about problems other than those discussed before.

Questions 7, 9, 10, and 11 emerged during our interviews and were directly related to the debts identified in the preceding interviews, so they may introduce some bias to our results. We

mitigated this issue by asking those questions at the end of the interview after asking open questions about their existing debt. By doing so, we avoided anchoring the subjects' answers to specific debts. Also, interviewees may be uncomfortable stating their thoughts for several reasons and may not tell the whole truth. We mitigated the problem via source triangulation. In cases with only one person (Companies F and G), it was impossible to achieve source triangulation.

The purpose of a multiple-case study is to investigate a set of cases in depth, not to generalize findings statistically. Yin (2018) states, "rather than thinking about your case(s) as a sample, you should think of your case study as the opportunity to shed empirical light on some theoretical concepts or principles". Practitioners may judge to what extent our findings apply to their particular context based on similarities and differences between their company and the investigated companies. A survey does not provide a rich context-related insight but enables reporting results statistically. A natural step further in our research is to conduct a survey where we collect opinions from more people and companies regarding the identified ATDs, interests, and principals.

6. Related work

We identified a set of ATDs in microservices that could hinder the adoption of microservices. A deeper study about barriers (and drivers) for adopting microservices comes from Knoche and Hasselbring (2019), who identify issues related to compliance, regulations, and licenses as barriers. We identified that licensing and regulations might become an ATD (and a barrier) for microservice architecture.

This article extends our previous work (de Toledo et al., 2019) by investigating six additional companies. The current results confirm the found debts in our previous single case study apart from business logic in the communication among services. The current study resulted in cross-company insights and, specifically, an update of our catalog of ATDs. Moreover, the originally-proposed debts in de Toledo et al. (2019) were changed as follows:

- "Too many point-to-point connections among services" is better explained as a coupling issue (Debt 2) and is also related to Debt 3;
- "Business logic implemented in the communication layer" is now described in Debt 9;
- "There is no approach to standardize the communication model among services" is described in Debt 3;
- "Weak source code and knowledge management for different services" is removed from our catalog because we found that it is better classified as a distinct, non-architectural type of debt;
- "Unnecessary presence of different middleware technologies in the communication among services" is merged into Debt 6, which is a more general debt description.

Taibi and Lenarduzzi (2018) interviewed 72 developers and built a catalog of bad smells on microservices. More recently, Taibi et al. (2020) provided a taxonomy of architectural and organizational anti-patterns in microservices and their possible solutions. There is a close relationship among ATDs, architectural smells, and architectural anti-patterns, but they are different concepts; not all bad smells and anti-patterns are ATD: a code duplication may be considered a bad smell, but not TD if there is no interest. Still, some of the ATDs we have identified can be considered anti-patterns. However, describing them as ATDs enables us to evaluate them in terms of interest and principal. For example, some smells identified by Taibi and Lenarduzzi (2018) overlap with ours: (i) shared persistence, which overlaps with Debt 8.1; (ii) too many standards, which overlaps with Debt 6; (iii) shared libraries, which overlaps with Debt 10.1; and (iv) Microservice Greedy, which overlaps with Debt 12. However, no solution is offered other than careful consideration of services to create. They present the same problems and solutions we found in our interviews, reinforcing the importance of these problems. No other overlaps exist. Our study also presents more details about the debt and its interest and principal.

Hasselbring and Steinacker (2017) argue that transforming internal libraries into open-source software may reduce issues with shared libraries (i.e., it may solve Debt 10.1). Despite insufficient evidence to confirm their suggestion, we believe more in-depth studies could confirm or refute such findings.

Bogner et al. (2019b) performed a qualitative study with 10 companies via 17 interviews to explore evolvability assurance processes for 14 microservice-based systems. Many of the issues reported were related to ATD. Our work differs in the research focus: they investigated evolvability assurance processes and came across ATD issues, but we systematically investigated the debt, interest, and principal in microservices' architecture. Their work partially overlaps with the following ATDs we found in our study: (i) technological heterogeneity, in which they discuss that their participants are divided about the use of several different technologies in microservices, which relates to Debt 6; (ii) interservice dependencies and the ripple effect, which refers to Debt 2; (iii) breaking API changes, which is an interest of Debt 4.1; and (iv) distributed code repositories, in which they argue that it may complicate the access to the source-code and relates to Debt 6 as well.

In summary, some related studies overlap with ours in a limited way, but none is as extensive and comprehensive as ours concerning ATD in microservices.

7. Conclusions and future work

During software development, it is vitally important to manage ATD to avoid extra costs in the long term. We provided a cross-company analysis to create a catalog of ATDs in microservices, their consequences (interest), and their solutions (principal). Moreover, we created a map of relationships among ATDs, their interest and principal. Such a map may support practitioners in identifying and avoiding ATDs and planning refactorings to remove them.

Regarding RQ1, we found ATDs that included business logic among services, shared databases, lack of data-traceability mechanisms, poorly designed APIs, and shared libraries. As for RQ2, we observed that such debts caused substantial interest, such as unexpected breaks due to changes in the database schema or other dependencies, unnecessary API complexity, coupling among services, and dependencies among teams. Finally, for RQ3, we identified how companies handle such ATDs and the ATD costs.

Future work includes running a survey to increase our results' generalizability and collect additional information on repayment prioritization. Furthermore, based on the insights reported in this article, we propose a new study that investigates metrics for measuring debt, principal, and interest in microservice architecture to quantify costs and benefits and support prioritization and decision-making.

Table 5
Interview guide.

ID	Question
1	Tell us about the organization and its divisions.
2	Describe the project, its duration, its size, its technologies, its goals and your role on it.
3	Talk more about the parts of the project that use microservices or another service-oriented architecture.
4	What challenges regarding the architecture have you faced recently? What were their causes and impacts? Did you manage to avoid any of them? How?
5	Are you migrating from an old solution? What challenges did you face during the migration? What were the costs of the migration? What were the costs of not migrating?
6	Did you have challenges regarding the communication among services? Did you have business logic outside the services in their communication? How did you manage to handle it?
7	Did you use any standard for the APIs/message format? Did you have any issues due to your choice of using/not using such standards? How did you manage to solve it?
8	How did you manage your source code and documentation?
9	Did you have any issues regarding third-party licenses? What were the costs and how did you manage to solve it?
10	Did you have any issues regarding shared libraries? What were the costs and how did you manage to solve it?
11	Did you have any issues regarding data storing? What were the costs and how did you manage to solve it?
12	Could you mention other situations with issues that (including why and how you managed to solve it): • reduce development speed? • cause more bugs? • have a negative impact on other system qualities? • impact many developers? • will become worse in the future?
13	Do you have any additional issues we did not covered before?

CRediT authorship contribution statement

Saulo S. de Toledo: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing - original draft, Writing - review & editing, Visualization. **Antonio Martini:** Conceptualization, Methodology, Validation, Formal analysis, Investigation, Writing - review & editing, Supervision, Project administration. **Dag I.K. Sjøberg:** Conceptualization, Methodology, Validation, Formal analysis, Writing - review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We are grateful to all the interviewees of this study. We thank the anonymous referees for their comments.

Appendix. Interview guide

The interview guide is presented in Table 5.

References

Avgeriou, P., Kruchten, P., Ozkaya, I., Seaman, C., 2016. Managing technical debt in software engineering (dagstuhl seminar 16162). In: Avgeriou, P., Kruchten, P., Ozkaya, I., Seaman, C. (Eds.), Dagstuhl Rep. 6 (4), 110–138. http://dx.doi.org/10.4230/DagRep.6.4.110, URL: http://drops.dagstuhl.de/opus/volltexte/2016/6693.

Besker, T., Martini, A., Bosch, J., 2017. The pricey bill of technical debt: When and by whom will it be paid?. In: Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017. IEEE, pp. 13–23. http://dx.doi.org/10.1109/ICSME.2017.42, URL: http://ieeexplore.ieee.org/document/8094405/.

Besker, T., Martini, A., Bosch, J., 2018a. Managing architectural technical debt: A unified model and systematic literature review. J. Syst. Softw. 135, 1–16. http://dx.doi.org/10.1016/j.jss.2017.09.025, URL: https://www.sciencedirect.com/science/article/pii/S0164121217302121.

Besker, T., Martini, A., Edirisooriya Lokuge, R., Blincoe, K., Bosch, J., 2018b. Embracing technical debt, from a startup company perspective. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 415–425. http://dx.doi.org/10.1109/ICSME.2018.00051, URL: https://ieeexplore.ieee.org/document/8530048/.

Bogner, J., Boceck, T., Popp, M., Tschechlov, D., Wagner, S., Zimmermann, A., 2019a. Towards a collaborative repository for the documentation of service-based antipatterns and bad smells. In: Proceedings - 2019 IEEE International Conference on Software Architecture - Companion, ICSA-C 2019. Institute of Electrical and Electronics Engineers Inc., pp. 95–101. http://dx.doi.org/10.1109/ICSA-C.2019.00025, URL: https://ieeexplore.ieee.org/document/8712355.

Bogner, J., Fritzsch, J., Wagner, S., Zimmermann, A., 2019b. Assuring the evolvability of microservices: Insights into industry practices and challenges. In: IEEE International Conference on Software Maintenance and Evolution (ICSME). Cleveland, Ohio, USA, pp. 546–556. http://dx.doi.org/10.1109/ICSME.2019.00089, URL: https://ieeexplore.ieee.org/document/8919247.

Corbin, J.M., Strauss, A.L., 2015. Basics of qualitative research: techniques and procedures for developing grounded theory, fourth ed. SAGE.

De Silva, L., Balasubramaniam, D., 2012. Controlling software architecture erosion: A survey. J. Syst. Softw. 85 (1), 132–151. http://dx.doi.org/10.1016/j.jss.2011.07.036, URL: https://www.sciencedirect.com/science/article/pii/S0164121211002044.

Di Francesco, P., Lago, P., Malavolta, I., 2019. Architecting with microservices: A systematic mapping study. J. Syst. Softw. 150, 77–97. http://dx.doi.org/10.1016/j.jss.2019.01.001, URL: https://www.sciencedirect.com/science/article/pii/S0164121219300019.

Di Francesco, P., Malavolta, I., Lago, P., 2017. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In: 2017 IEEE International Conference on Software Architecture (ICSA). pp. 21–30. http://dx.doi.org/10.1109/ICSA.2017.24, URL: https://ieeexplore.ieee.org/document/7930195/.

Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L., 2017. Microservices: Yesterday, today, and tomorrow. In: Present and Ulterior Software Engineering. Springer International Publishing, Cham, pp. 195–216. http://dx.doi.org/10.1007/978-3-319-67425-4_12, (Chapter 12).

Ernst, N.A., Bellomo, S., Ozkaya, I., Nord, R.L., Gorton, I., 2015. Measure it? Manage it? Ignore it? Software practitioners and technical debt. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015. ACM Press, New York, New York, USA, pp. 50–60. http://dx.doi.org/10.1145/2786805.2786848, URL: http://dl.acm.org/citation.cfm?doid=2786805.2786848.

Evans, E., 2004. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, p. 529.

Fowler, M., 2015. Microservice trade-offs. URL: https://martinfowler.com/articles/microservice-trade-offs.html.

Furda, A., Fidge, C., Zimmermann, O., Kelly, W., Barros, A., 2018. Migrating enterprise legacy source code to microservices: On multitenancy, statefulness, and data consistency. IEEE Softw. 35 (3), 63–72. http://dx.doi.org/10.1109/MS.2017.440134612, URL: https://ieeexplore.ieee.org/document/8186442.

Garcia-Molina, H., Salem, K., 1987. Sagas. ACM SIGMOD Rec. 16 (3), 249–259. http://dx.doi.org/10.1145/38714.38742, URL: https://dl.acm.org/doi/10.1145/38714.38742.

Hasselbring, W., Steinacker, G., 2017. Microservice architectures for scalability, agility and reliability in e-commerce. In: Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings. Institute of Electrical and Electronics Engineers Inc., pp. 243–246. http://dx.doi.org/10.1109/ICSAW.2017.11, URL: http://ieeexplore.ieee.org/document/7958496.

Hohpe, G., Woolf, B., 2012. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, first ed. Addison-Wesley.

Knoche, H., Hasselbring, W., 2019. Drivers and barriers for microservice adoption - a survey among professionals in Germany. Enterpr. Model. Inf. Syst. Archit. (EMISAJ) 14, 1–35. http://dx.doi.org/10.18417/emisa.14.1, URL: https://www.emisa-journal.org/emisa/article/view/164.

Kruchten, P., Nord, R.L., Ozkaya, I., 2012. Technical debt: From metaphor to theory and practice. IEEE Softw. 29 (6), 18–21. http://dx.doi.org/10.1109/MS.2012.167, URL: https://ieeexplore.ieee.org/document/6336722.

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

Li, W., Lemieux, Y., Gao, J., Zhao, Z., Han, Y., 2019. Service mesh: Challenges, state of the art, and future research opportunities. In: Proceedings - 13th IEEE International Conference on Service-Oriented System Engineering, SOSE 2019, 10th International Workshop on Joint Cloud Computing, JCC 2019 and 2019 IEEE International Workshop on Cloud Computing in Robotic Systems, CCRS 2019. Institute of Electrical and Electronics Engineers Inc., pp. 122–127. http://dx.doi.org/10.1109/SOSE.2019.00026, URL: https://ieeexplore.ieee.org/document/8705911.

Li, Z., Liang, P., Avgeriou, P., 2014. Architectural debt management in value-oriented architecting. In: Economics-Driven Software Architecture. pp. 183–204. http://dx.doi.org/10.1016/B978-0-12-410464-8.00009-X, URL: https://www.sciencedirect.com/science/article/pii/B978012410464800009X.

Marquez, G., Astudillo, H., 2018. Actual use of architectural patterns in microservices-based open source projects. In: Proceedings - Asia-Pacific Software Engineering Conference, APSEC, Vol. 2018-Decem. IEEE Computer Society, pp. 31–40. http://dx.doi.org/10.1109/APSEC.2018.00017, URL: https://ieeexplore.ieee.org/document/8719492.

Márquez, G., Villegas, M.M., Astudillo, H., 2018. A pattern language for scalable microservices-based systems. In: ACM International Conference Proceeding Series. pp. 1–7. http://dx.doi.org/10.1145/3241403.3241429, URL: https://dl.acm.org/doi/10.1145/3241403.3241429.

Martini, A., Bosch, J., 2016. An empirically developed method to aid decisions on architectural technical debt refactoring: Anacondebt. In: Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16. ACM Press, New York, New York, USA, pp. 31–40. http://dx.doi.org/10.1145/2889160.2889224, URL: https://dl.acm.org/doi/10.1145/2889160.2889224.

Martini, A., Bosch, J., Chaudron, M., 2015. Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study. In: Information and Software Technology, Vol. 67. Elsevier, pp. 237–253. http://dx.doi.org/10.1016/j.infsof.2015.07.005, URL: https://www.sciencedirect.com/science/article/pii/S0950584915001287.

Martini, A., Fontana, F.A., Biaggi, A., Roveda, R., 2018. Identifying and Prioritizing Architectural Debt Through Architectural Smells: A Case Study in a Large Software Company. In: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Madrid, Spain, pp. 320–335. http://dx.doi.org/10.1007/978-3-030-00761-4_21, URL: https://link.springer.com/chapter/10.1007/978-3-030-00761-4_21.

Mo, R., Cai, Y., Kazman, R., Xiao, L., Feng, Q., 2019. Architecture anti-patterns: Automatically detectable violations of design principles. IEEE Trans. Softw. Eng. 1. http://dx.doi.org/10.1109/tse.2019.2910856, URL: https://ieeexplore.ieee.org/document/8691586.

Montesi, F., Weber, J., 2016. Circuit breakers, discovery, and API gateways in microservices. CoRR abs/1609.0 arXiv:1609.05830v2 URL: http://arxiv.org/abs/1609.05830v2.

Mosqueira-Rey, E., Alonso-Ríos, D., Moret-Bonillo, V., Fernández-Varela, I., Álvarez-Estévez, D., 2018. A systematic approach to API usability: Taxonomy-derived criteria and a case study. Inf. Softw. Technol. 97, 46–63. http://dx.doi.org/10.1016/j.infsof.2017.12.010, URL: https://www.sciencedirect.com/science/article/pii/S0950584917302471.

Newman, S., 2017. Building Microservices: Designing Fine-Grained Systems, first ed. O'Reilly Media, Inc..

Newman, S., 2019. Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith, first ed. O'Reilly Media, Inc., p. 221.

Niblett, P., Graham, S., 2005. Events and service-oriented architecture: The OASIS web services notification specifications. IBM Syst. J. 44 (4), 869–886. http://dx.doi.org/10.1147/sj.444.0869, URL: https://ieeexplore.ieee.org/document/5386704.

Rademacher, F., Sachweh, S., Zundorf, A., 2017. Differences between model-driven development of service-oriented and microservice architecture. In: Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings. Institute of Electrical and Electronics Engineers Inc., pp. 38–45. http://dx.doi.org/10.1109/ICSAW.2017.32, URL: https://ieeexplore.ieee.org/document/7958454.

Reddy, P.K., Kitsuregawa, M., 1998. Reducing the blocking in two-phase commit protocol employing backup sites. In: Proceedings - 3rd IFCIS International Conference on Cooperative Information Systems, CoopIS 1998, Vol. 1998-Augus. Institute of Electrical and Electronics Engineers Inc., pp. 406–415. http://dx.doi.org/10.1109/COOPIS.1998.706315, URL: https://ieeexplore.ieee.org/document/706315.

Richards, M., 2016. In: Barber, N., Roumeliotis, R. (Eds.), Microservices vs. Service-Oriented Architecture, first ed. O'Reilly Media, Inc., p. 45.

Runeson, P., Höst, M., Rainer, A., Regnell, B., 2012. Case Study Research in Software Engineering: Guidelines and Examples, first ed. Wiley Publishing.

Schmid, K., 2013. A formal approach to technical debt decision making. In: QoSA 2013 - Proceedings of the 9th International ACM Sigsoft Conference on the Quality of Software Architectures. ACM Press, New York, New York, USA, pp. 153–162. http://dx.doi.org/10.1145/2465478.2465492, URL: https://dl.acm.org/doi/10.1145/2465478.2465492.

Taibi, D., Lenarduzzi, V., 2018. On the definition of microservice bad smells. IEEE Softw. 35 (3), 56–62. http://dx.doi.org/10.1109/MS.2018.2141031, URL: https://ieeexplore.ieee.org/document/8354414.

Taibi, D., Lenarduzzi, V., Pahl, C., 2020. Microservices anti-patterns: A taxonomy. In: Bucchiarone, A., Dragoni, N., Dustdar, S., Lago, P., Mazzara, M., Rivera, V., Sadovykh, A. (Eds.), Microservices: Science and Engineering. Springer International Publishing, Cham, pp. 111–128. http://dx.doi.org/10.1007/978-3-030-31646-4_5, URL: https://link.springer.com/chapter/10.1007/978-3-030-31646-4_5.

Thönes, J., 2015. Microservices. IEEE Softw. 32 (1), 116. http://dx.doi.org/10.1109/MS.2015.11, URL: https://ieeexplore.ieee.org/document/7030212.

de Toledo, S.S., Martini, A., Przybyszewska, A., Sjoberg, D.I., 2019. Architectural technical debt in microservices: A case study in a large company. In: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt). IEEE, Montreal, Quebec - CA, pp. 78–87. http://dx.doi.org/10.1109/techdebt.2019.00026, URL: https://ieeexplore.ieee.org/document/8786035/.

Van Gurp, J., Bosch, J., 2002. Design erosion: Problems and causes. J. Syst. Softw. 61 (2), 105–119. http://dx.doi.org/10.1016/S0164-1212(01)00152-2, URL: https://www.sciencedirect.com/science/article/pii/S0164121201001522.

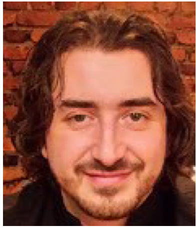
Verdecchia, R., Malavolta, I., Lago, P., 2018. Architectural technical debt identification: The research landscape. In: Proceedings - International Conference on Software Engineering. pp. 11–20. http://dx.doi.org/10.1145/3194164.3194176, URL: https://dl.acm.org/doi/10.1145/3194164.3194176.

Vogels, W., 2008. Eventually consistent. Queue 6 (6), 14–19. http://dx.doi.org/10.1145/1466443.1466448, URL: https://dl.acm.org/doi/10.1145/1466443.1466448.

Yin, R.K., 2018. Case Study Research and Applications: Design and Methods, sixth ed. Sage Publications, Inc.

Zimmermann, O., 2017. Microservices tenets: Agile approach to service development and deployment. Comput. Sci. - Res. Dev. 32 (3–4), 301–310. http://dx.doi.org/10.1007/s00450-016-0337-0, URL: http://link.springer.com/10.1007/s00450-016-0337-0.

Saulo Soares de Toledo is a Ph.D. candidate at the University of Oslo, Oslo, Norway. His research interests include technical debt, microservices, and service-oriented architecture. de Toledo received his master's degree in computer science from the Federal University of Campina Grande, Brazil.



Antonio Martini is an associate professor at the University of Oslo, Oslo, Norway, and a part-time researcher at the Chalmers University of Technology, Gothenburg, Sweden. His research interests include technical debt, software architecture, technical leadership, and agile software development. Martini received his Ph.D. in software engineering from Chalmers University of Technology, Sweden.

Dag I.K. Sjøberg is a professor at the University of Oslo. His research interests are the software lifecycle, including agile and lean development processes, and empirical research methods in software engineering. Sjøberg received his Ph.D. in computing science from the University of Glasgow, Scotland, UK.