Architectural Technical Debt in Microservices

A case study in a large company

Saulo S. de Toledo
Dept. of Informatics
University of Oslo
Oslo, Norway
Email: saulos@ifi.uio.no

Antonio Martini
Dept. of Informatics
University of Oslo
Oslo, Norway
Email: antonima@ifi.uio.no

Agata Przybyszewska
Dept. of Computer Science
IT University of Copenhagen
Copenhagen, Denmark
Email: agpr@itu.dk

Dag I.K. Sjøberg
Dept. of Informatics
University of Oslo
Oslo, Norway
Email: dagsj@ifi.uio.no

***Abstract*—*Introduction:* Software companies aim to achieve continuous delivery to constantly provide value to their customers. A popular strategy is to use microservices architecture. However, such an architecture is also subject to debt, which obstructs the continuous delivery process and thus negatively affects the software released to the customers.**

***Objectives:* The aim of this study is to identify issues, solutions and risks related to Architecture Technical Debt in microservices.**

***Method:* We conducted an exploratory case study of a real life project with about 1000 services in a large, international company. Through qualitative analysis of documents and interviews, we investigated Architecture Technical Debt in the communication layer of a system with microservices architecture.**

***Results:* Our main contributions are a list of Architecture Technical Debt issues specific for the communication layer in a system with microservices architecture, as well as their associated negative impact (interest), a solution to repay the debt and the its cost (principal). Among the found Architecture Technical Debt issues were the existence of business logic in the communication layer and a high amount of point-to-point connections between services. The studied solution consists of the implementation of different canonical data models specific to different domains, the removal of business logic from the communication layer, and migration from services to use the communication layer correctly. We also contributed with a list of possible risks that may affect the payment of the debt, as lack of funding and inadequate prioritization.**

***Conclusion:* We found issues, solutions and risks that are specific for microservices architectures not yet encountered in the current literature. Our results may be useful for practitioners that want to avoid or repay Technical Debt in their microservices architecture.**

***Index Terms*—Technical Debt, Architecture, Microservices, Case Study**

I. Introduction

Software companies aim to achieve continuous delivery to constantly provide value to their customers [1]. With the objective of making teams able to deploy new features independently from other teams, and therefore more continuously, a strategy recently adopted by several companies is the use of a microservices architecture. Such architecture, based on loosely-coupled services, is also perceived as favorable for the evolution of the system, boosting quick replacement of sub-optimal services and the addition of new ones [2].

However, microservices technology has only recently become a popular topic, which implies a lack of empirical data on long-standing systems using such technology [3]. Such lack holds also with respect to what is defined (or considered to be) a good microservices architecture and, consequently, to what sub-optimal solutions can lead to costly Architectural Debt.

Architecture in microservices is in fact based on a number of qualities and structural features that are different from traditional systems. An example of this is the use of a collection self-contained microservices connected by a messaging system usually called communication layer. However, knowledge about what is either a virtuous or a harmful design of such architectures is still missing [3], especially for evidence collected in a systematic research fashion and in the context of well-established industrial systems.

Understanding more about the negative effect (interest) of Architectural Technical Debt (ATD), about possible solutions and their cost (principal of ATD) would be useful for organizations and development teams that adopt microservices. Consequently, our research questions are:

RQ1: What is ATD in microservices?

- **RQ1.1:** What is the negative impact (interest) generated by ATD in microservices?
- **RQ1.2:** What is a solution for the identified ATD in microservices and its associated refactoring cost (principal)?

We have investigated a large company developing financial services, in which microservices have been extensively used (about 1000 services). After the first preliminary investigation, we found that the organization had identified that the communication layer contained particularly costly ATD. In the studied case, the software of the communication layer was refactored and the services had started to be migrated to the new solution.

In addition to the previous RQs, we also studied what led to the presence of ATD and which risks were encountered by the organization when refactoring and migrating to the new solution. We therefore also aim to answer the following research question:

RQ2: What are the risks in refactoring ATD in microservices?

The remainder of this paper is organized as follows. Section II presents the main concepts on microservices, ATD and the company context. Section III presents our research methodology, data collection and analysis. Section IV Presents our findings for each RQ. Section V presents the implications

for research and industry. Section VI presents the related work. Finally, Section VII presents our conclusions.

II. Background

A. Microservices and TD

Microservices is an architectural style that decomposes a system into small and loosely coupled services [3]. According to Dragoni et al. [4], a service in the context of microservices is an independent process that interacts with their equals by means of messages, which can be synchronous or asynchronous. For asynchronous messaging, it is common to use technologies like Apache Kafka[1] and RabbitMQ[2] as a communication layer [5], or to write a piece of software on top of tools like these to act as such.

When the number of services in such architecture grows, it is common to find for ways to standardize the communication. One solution that has been adopted in Service Oriented Architecture (SOA) [6] projects, from which microservices architecture has been considered a subarea by several practitioners [7], is the use of canonical data models (or just canonical models). Canonical models define how to structure an organization's information by being a reference for how to represent entities and their relationships in the system [8]. As an example, a canonical model can define that a user profile in a social network for job seekers has a name, an email and optionally the company he works for; that definition, along with details like the requirement of having a valid email address, for instance, should be shared by all services that work with this data. If a particular service is part of another data domain, e.g. a newsletter service, and does not need to access (or should not know about) this data, it can use a specific canonical model for that domain.

Technical Debt (TD) was first introduced as a way to present to non-developers the dangers of shipping "not quite right code" [9]. TD is a sub-optimal design or implementation that brings benefits in the short term but increases the costs of the system in the long run, consequently affecting its evolvability and maintainability [10]. An example is the use of a database solution that does not meet all needs of the system but is easier to use immediately.

Several authors have expanded the metaphor of TD, including the definition of several kinds of TD. Architectural Technical Debt (ATD) is a kind of TD that is concerned with the system architecture [11]. Kruchten et al. [11] identify ATD as the most challenging type of TD to be uncovered.

The definition of TD [10] includes three key concepts:

- **Debt:** the presence of sub-optimal solutions. The amount of debt in a system can be calculated as the amount of sub-optimal solutions with respect to a desired solution. For example, a system that has 100 dependency violations has more debt than one that has only one violation.
- **Interest:** an extra cost that must be paid due to the existence of a debt. In other words, it is the amount that will be saved if the debt does not exist. By using the previous dependencies example, the interest is the additional cost of adding/changing code with the presence of dependencies in contrast to a better solution. For example, when several non-allowed dependencies are in place, the developers might need to change the code in several places instead of in one place only.
- **Principal:** the cost of developing the system avoiding the debt or the cost of refactoring it. In our previous example, the principal is the cost of setting up an optimal system with loosely coupled files from the beginning or, if the debt has already been accumulated, the principal is the cost of refactoring the system to remove the dependencies.

If the interest is low, accruing the debt might be beneficial. For example, a dependency that does not generate extra cost, should not necessarily be removed. On the contrary, if the interest is greater than the principal, avoiding or refactoring the debt is usually considered beneficial. Thus, a fundamental question in our investigation on TD is when to incur or repay the debt. In this case study, two other concepts also helped us to reason about what influenced the decision to fix TD and how to do it:

- **Risk:** they can affect the decision process today or can be a source of concern in the future. The fear of something go wrong may affect the probability of taking that decision;
- **Solution:** An approach for fixing technical debt or reducing the interest paid. In the example related to dependencies, a solution is a rewriting a system without the dependencies or reducing the work of the developers so they do not have to change the code and pay the interest.

B. Company Context

The company where the study was performed, is a large, multinational financial services, with a complex, heterogeneous IT system landscape.

- **Business:** The core of the financial business is about compliance and risk.
  The financial crisis has led to a series of tight regulations of the financial sector, and different governance bodies have different legislations (Basel, MiFID[3] I, MiFID II, FRTB[4]), that are a prerequisite to having a banking license. More control is exercised about realistic and timely reports on risk, exposure in different markets, currencies and clients, limits on risk, data lineage and reporting of trade activities to authorities.
  Risk computations give a significant business advantage, if they are performed as close to real time as possible, and the different computation heavy simulations are performed in as much detail as feasible. This gives a business demand on computing as close to real time as possible,

[1] http://kafka.apache.org/
[2] https://www.rabbitmq.com/
[3] Markets in Financial Instruments Directive
[4] Fundamental Review of the Trading Book

that also needs to be consistent across a data set of around 20.000.000 daily transactions.

- **Organization:** As the company we studied is multi-national, they are operating in several countries, and separate national legislation, together with super-national regulations like European Union (EU), give rise to another dimension of IT complexity.
  The organization has introduced a separation of duties, where employees either belong to Business, IT or Operations. Accountability is structured in such a way, that every application has an Application Owner in the business that is the responsible sponsor, and drives the business requirements. There is an Application provider in the IT organization, that is responsible for the application development and delivery. Finally, there is a sign-off, where the Operations team takes the accountability for operating the application/service, and is responsible for the Service Level Agreement (SLA) [12]. Most teams are structured as virtual teams, across the national borders.
  Frequent organizational changes, together with the history of mergers, have resulted in heterogeneous IT systems, lack of ownership for older artifacts, and orphaned business logic in the communication middleware.
- **Architecture:** Microservices architecture, consisting of about 1000 services, of which about 150 are critical for the business.
  The services operate using different connectors - the legacy communication layer solution, new communication solution, point to point service calls, database pumps, direct database connections, file transfers and mails, external banking networks, and mainframe gateways.
  The connectivity layers are often containing logic that performs ETL[5], and sometimes also contains business logic. Some of the logic has become orphaned, due to organizational changes, and a lack of direct link between application and connector, as these are often owned by different teams.
- **Process:** To counteract new compliance requirements and legislation from regulators, and the inability to upgrade systems when needed - due to accumulation of technical debt, the management has decided to start a large program, that will simplify some of the complexity, reduce technical debt and restore business agility.
  Focus was on data foundation, and fixing lack of accountability, by appoint Data Asset Owners, that will govern and mandate the processing of various data assets inside the organization.
  Complicated logic, and inability to test changes, has led to a complicated change process, and DevOps culture was introduced to counteract this.
  The traditional governance of architecture in the company, has been a waterfall-based model, inspired by the PRINCE2 methodology [13], with a series of approval gates - early architecture approval, solution design approval, and a run gate. After a project is in run state, there are no more controls.
  A program has been set up to refactor the problematic area, and the program has moved away from the traditional waterfall methodology. Instead, they aim to execute using agile and Scrum methodology. Later, during the program's lifetime, in order to coordinate the large number of projects, the refactoring program embraced the Scaled Agile Framework (SAFe) with its Agile Release Trains (ART) construct.

[5]Extraction, transformation and loading: three steps to combine data from multiple sources.

III. Methodology

We conducted an exploratory case-study in a large, international company. The company is involved in a large financial group that works with financial services related assets. The studied system has a Microservice oriented architecture with several issues that were identified by the company mainly on their messaging layer. It contains more than 1000 services, a messaging layer used by most of the services, and some other communication technologies. Due to the existence of ATD, the company developed a new solution to replace the old messaging layer and the other communication technologies, unifying the data transmission terminology and solving several related issues. Within the case, we were able to identify ATD issues through the evaluation of documents and face-to-face interviews. With the migration in process, we can also identify both the interest (by investigating the costs especially on areas that did not migrate to the new system yet) and the principal (identified by the cost of migration itself and the areas in the company that finished their migration to a new solution free of those ATD). The need for fixing issues in the communication layer made us to focus on this part of the system. We examined some documentation on the business case related to the Technical Debt and interviewed some practitioners and we used qualitative techniques to analyze them. Our main goal was to understand the problems in the communication layer pointed out by the company, finding their causes and costs, while looking to the case as a whole.

A. Data collection

The case we study is a large refactoring initiative, that works on replacing the aged middleware, together with its burden of embedded business logic, with a modern architecture, based on the reactive manifesto. Both communication layer alternatives have been in production in parallel for a long time, making for a perfect lab, where architecture changes, and change to technical debt can be studied.

We acquired data from the following sources:

- Business case documents created by the enterprise architects contained information about:
  - the ATD present in the analyzed system. Based on this information, we created a first list of ATD issues, which was later refined in subsequent steps. (Section IV);

  - the negative impact created by the ATD, used to infer the interest of ATD;
  - the technologies used in the current implementation and those that have been proposed to mitigate the problems;
  - a proposed approach to refactor the system to remove the ATD;
  - the risks regarding the removal of the ATD, also used to define its principal;
  - the risks regarding not removing the ATD, also used to infer the interest of the ATD.
- A meeting with two enterprise architects and one product owner. All of them worked in both the old and the new solution. We collected the following information:
  - an overview of the entire microservices architecture;
  - an overview about the organization and the relationship with the ATD case;
  - an explanation about the motivations and the challenges of refactoring;
  - a list of suitable interviewees to collect more data.

These two sources indicated that the major issues were affecting the communication layer. We then decided to interview those subjects mentioned in the meeting with the following goals:

- confirm and get more details about the risks and impacts described in the document;
- confirm our understanding and get more details about the project, its context and its architecture, from different perspectives;
- confirm the existence of the ATD pointed out in those sources, including their impact and costs, as well as getting more instances of ATD, interest and principal from different people involved in the project;
- get information about the solution in progress and its costs;

We conducted semi-structured face-to-face interviews with ten subjects:

- **three product owners/managers:** The questions concerned the company context and the challenges in the project seen from a management point of view (for handling the current solution or to change to a new one without the debt);
- **two architects:** The questions concerned issues in the software architecture and the proposed solutions, including risks;
- **five developers:** The questions concerned daily issues faced by the development team and the way they tried to fix the issues.

Each interview was audio recorded. The relevant parts were transcribed and then coded. During the interviews, our questions focused in the specific role of the respondents (e.g. we asked questions about the organization for managers, and questions about architectural decisions to architects). The interview guide is presented in Table I. Some questions, such as the number eight, might seem to be suggestive: for example, "what are the problems with the messaging layer?" might lead the interviewee to assume that problems are already present in the messaging layer. However, this was done by design since the questions were based on facts obtained from the previous document analysis. We wanted to confirm if such problems exist across the interviewees, elicit more details and identify other information not mentioned in the document analysis.

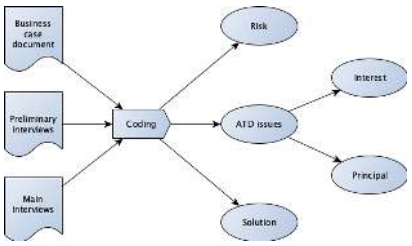

Fig. 1. Data analysis overview

B. Data analysis

We coded the interviews to search for evidence that would answer our research questions. The best way to do it is by classifying the data into the previously introduced categories (i.e. ATD issues, interest, principal, organizational context, risk and solution). Thus, we used a deductive approach on top of open coding [14].

Figure 1 presents an overview of the data analysis. We started by analyzing the business case document. We manually classified the information therein according to the aforementioned categories and identified topics that were unclear. Those categories were used as codes. We then conducted preliminary interviews to acquire more information about the project and validate our understanding of the document. Finally, we planned and conducted the main interviews. We partially transcribed them and coded both (audio and transcriptions) by using a Qualitative Data Analysis (QDA) tool called NVivo, which supports tracking of the links between codes and the original data in the interviews and auxiliary documents. The last step assisted us to find, confirm and rectify our understanding about the data.

Some examples of coding:

- **ATD issues:** *"There is business logic inside the messaging layer code"* (quote by an architect);
- **Interest:** *"a lot of effort is needed to investigate why that happened"* (quote by a developer);
- **Principal:** *"For each new IT development team migrated to the new technology, there will be an associated start-up cost equivalent to the man hours used within the team to get accustomed to the new tools and ways of working"* (quote from the business case document);
- **Risk:** *"the risk of not migrating is that the message and data landscape becomes more complex"* (quote by an architect);
- **Solution:** *"the solution is a middleware system that (...) has been deployed to production"* (adapted from the business case document).

TABLE I
INTERVIEW QUESTIONS

ID	Question	Related to	Target group
1	Tell us about the organization, its divisions and how the current problems affect them.	• Debt • Interest	• PO/Managers
2	Tell us about the organization, its divisions and how migrating to a new solution is affecting them.	• Debt • Principal	• PO/Managers
3	What are the risks of the migration not being properly funded? What is the probability of that happening?	• Interest • Risk	• PO/Managers
4	What incidents related to ATD happened recently? What are their causes and impact?	• Debt • Interest	• PO/Managers • Architects
5	What are the challenges of migrating to a new solution? What are the costs of the migration? What are the costs of not migrating?	• Interest • Principal	• PO/Managers • Architects
6	What is the probability/cost of the transition to a new solution never finish?	• Interest • Risk	• PO/Managers • Architects
7	What is the effort required to migrate to the new solution?	• Principal	• PO/Managers • Architects • Developers
8	What are the problems with the messaging layer? What effort is required to keep working on it?	• Debt • Interest	• Architects • Developers
9	We found some issues regarding data transformation in the messaging layer on previous sources of data. Why is that important, how that works in the old and in the new solution, and what is the cost/impact of that?	• Debt • Interest • Principal	• Architects • Developers
10	What is the importance of a canonical model in the project?	• Principal	• Architects • Developers
11	Can you give one or more examples of issues that (and why): • reduce development speed? • cause more bugs? • have a negative impact on other system qualities? • impact many developers? • will become worse in the future?	• Debt • Interest	• Developers
12	What is the importance of a canonical model in the project?	• Principal	• Developers
13	What are the risks of keeping working with the old solution? Why?	• Interest • Risk	• Developers
14	What are the risks of migrating to a new solution? Why?	• Principal • Risk	• Developers

The procedure leads us to identify the data presented in the Section IV.

IV. RESULTS

This section discusses the results of each of the research questions. Table II gives a summary.

A. What is ATD in Microservice? (RQ1)

The results we present below focus on the communication layer between services. As shown in Table II, we identified the following issues:

- *Too many point-to-point connections among services:* Most of the services communicate through a messaging layer, but this layer is not used correctly. Instead of exposing data that could be used by any consumer through the communication layer, every new service that arrives in the system is designed to communicate only with its immediate consumers, creating a point-to-point connection unique for these endpoints. Thus, it is extremely complex to visualize and predict the connections between services. As a result, there is a high volume of connections between the services in the system;
- *Business logic inside the communication layer:* There are filters specific to some endpoints that contains business logic in order to transform data between endpoints. However, that creates a dependency between the services and the communication layer code. If developers change the data sent by a service, the communication layer can encounter a scheme that it does not recognize and behave unexpectedly. Service developers should not change the format before it has been agreed with the communication layer team. Also, the latter must know how every service (from several hundreds) work to proper update the business logic when required;
- *No standard communication model (Tower of Babel problem):* There is no standard model for communication among services. Each team creates its own communication mechanism, which is messy in an environment with hundreds of services. We call it *Tower of Babel problem* in analogy to the history on the Tower of Babel, where speakers could not understand each other due to the different languages;
- *Weak source code and knowledge management for different services:* Microservices encourage practitioners to work with small, independent services. In an environment with several services, some of these teams may be disbanded or allocated to work on other services. In this case study, some knowledge about legacy services and also

TABLE II
BRIEF SUMMARY OF OUR FINDINGS

	Debt (D)	Description of Interest (I)	Description of Principal (P)
1	Too many point-to-point connections among services.	Extra effort when evolving and maintaining the system due to the high amount of connections among services, increasing the system management and operations effort.	Rewrite the communication layer and migrate the services to use it properly.
2	Business logic implemented in the communication layer.	Communication layer developers must know and work on unnecessary details related to each client, decreasing business agility.	Remove the business logic form the communication layer and updated the services where required.
3	There is no approach to standardise the communication model among services (*Tower of Babel* problem).	Too many different models to deal with, decreasing business agility and increasing the cost of ownership for each service.	Define a canonical model per domain and rewrite all clients to use it.
4	Weak source code and knowledge management for different services.	Missing source code and lost knowledge due to miss of documentation and related artifacts, demanding code rewriting, reverse engineering or some workaround, increasing the total cost of ownership of the project.	Centralize the source code and documentation for all services in a common management system.
5	Unnecessary presence of different middleware technologies in the communication among services.	Extra effort to handle different technologies that have the same objective of sending messages among services, decreasing business agility.	Provide a common middleware that can be used by all services and rewrite those to use it.

some source code was lost due to changes in the teams or due to key people no longer working in the organization. Each team used its own source code management tool and shared the knowledge only internally within the team. Some information was lost a few months after the teams are disbanded;

- *Different middleware technologies:* Seven different middleware technologies (including the communication layer that was being rewritten) were used to exchange messages among services.

B. What is the negative impact (interest) generated by ATD in microservices? (RQ1.1)

We summarize each ATD issue and explain the related interest found in our investigation:

- *Interest on D1 (high number of point-to-point connections among services):*
  - *High cost for evolving and maintaining the system:* Several interviewees mentioned the complexity for handling the high number of direct connections between the services. The connections are not easy to track, as well as investigating the data exchanged between them;
  - *Coupling between services:* Once the services are directly dependent of each other format, they are coupled. That affecting the principle of lose coupling between services in a microservices architecture, in which each service should have none or a minimal impact on other services. Therefore, there is an extra work of changing dependent services instead of updating only one.
- *Interest on D2 (existence of business logic inside the communication layer):*
  - *Necessity of handling an overwhelming amount of details about services in the communication layer:* Developers in the communication layer must know details from the clients in order to maintain the business logic. Every time a new client arises, there is a possibility of having new business logic. It is impossible for the developers to know the details for all different services; thus, they need to put some extra effort to understand such a complex system;
  - *Unnecessary dependency between development teams:* When a change in the data sent by a service is required, the service developers should contact the communication layer team before doing any changes to prevent the system from behaving unexpectedly. A conversation then takes place through formal mail messages between the teams. One of our interviewees mentioned that the work they do in an entire week could take minutes if they did not have this bureaucracy in the process. In such a way, the extra-cost is the time wasted by both teams while waiting for the agreement to be bound;
  - *Unnecessary implementation of transformations and filters:* The communication layer developers must implement and maintain transformations and filters that should not be present in that layer. This interest grows over time due to the existence of an increasing number of extra codes. As the system becomes more complex, the more difficult it is to refactor it.
- *Interest on D3 (no approach to standardize the communication model among services):*
  - *There are too many different data formats to deal with:* Teams developing new services decide about the format of the data to be used by them. If there is no guideline to follow, it is likely to have a new format different from existing ones. As a result, some data transformation and filtering must take place in order to allow these services to communicate. Thus,

there is an extra cost to implement and maintain the code that performs this job.

- *Interest on D4 (weak source code and knowledge management for different services):*
  - *Effort for reverse engineering software with missing source code:* While updating the communication layer, some changes may need to be done in one of these projects with missing source code to support new requirements. When that happens, extra effort is required to one (or more) of these: (a) rewrite the old code, (b) do reverse engineering in order to try an alternative solution or (c) creating some sort of workaround.
  - *Effort to understand software whose knowledge and documentation is missing:* When documentation or any other source of knowledge about a piece of software is not easily accessible, it may be hard for the team that is working on it (especially if it is a new team that did not work in the original software) to understand the reasons of some inputs or outputs in that software. Indeed, that happened in this case and interviewees reported the loss of several hours trying to figure out that information.
- *Interest on D5 (unnecessary presence of different middleware technologies in the communication among services):*
  - *Need for maintaining different middleware solutions with the same purpose:* The existence of different technologies for solving the same problem only makes the architecture harder to handle, requiring effort to understand different technologies every time they are used by a new client.

C. What is a solution for the identified ATD in microservices and its associated refactoring cost (principal) (RQ1.2)

The list presented below is the result of the company's experience by fixing these issues, and as such they indeed represent real case solutions:

- *Principal on D1 (high number of point-to-point connections among services):*
  - *Rewrite the communication layer:* Removing the point-to-point connections requires redesign on the way the services communicate. That, of course, needs a communication layer capable of handling this new way of exchanging messages. Thus, the new layer should be redesigned according to such scheme;
  - Migrate the services to use the new architecture: With a new communication layer in place, every service should migrate to use it.
- *Principal on D2 (existence of business logic inside the communication layer):*
  - *Remove the business logic inside the communication layer:* It is important to simplify the layer by removing all that logic, keeping the responsibility of knowing how to communicate to the services themselves. That, of course, only moves the problem to another place, but that is reduced by solving D3 as we discuss later;
  - *Move the business logic to the services:* There should not be transformation code inside the communication layer if that is required only by the services. They should have the responsibility of handling the data they want. Once this adds complexity to the services, a better approach should be to combine this solution with the solution to D3, as we discuss later.
- *Principal on D3 (no approach to standardize the communication model among services):*
  - *Define a canonical model per domain:* A way to solve the need of transformation between several services is the standardization of the communication. That can be done by defining a canonical model. However, in a big system, the meaning of entities (e.g. user, company etc.) vary according to the context they are applied (e.g. the data required in a social media describing a user and the company he/she works for is different from the data required for a user subscribed in a newsletter service). In this context, a single canonical model can be confusing, because it will merge information from several domains. That can (i) cause an overhead of details in the model, requiring developers to put fields to identify a complete user profile in a single newsletter service that only needs their emails, (ii) expose the structure of data that should not be shared (a newsletter services does not need to know that some service store the user address, for instance), and (iii) different domains may interpret the same attribute in a different way ("type" in a newsletter system can store the type of newsletters that will be received by the subscriber, but it can means the type of user in a profile, as administrator or visitor). In addition, the same field may contain different kinds of data depending on the context. Thus, a good solution is to have several canonical models for the different domains;
  - *Update the services to use the newly defined canonical models:* After having the canonical models in place, the next step is to update the services to use it. That will also help to solve the problem pointed out on the principal for D2, where we discussed about the need of having business logic due to the existence of non-standardized data among the services.
- *Principal on D4 (weak source code and knowledge management for different services):*
  - *Centralize the source code and documentation for all services in a common management system:* The solution for this problem goes through keeping a centralized management system whereby all the historic data can be accessed, including documentation, knowledge reports and source code. A change of

policy must be in place in the company to ensure the teams will put data in this place. After that, no additional cost should be required, once the teams will only change the place where they will deposit the information.

- *Principal on D5 (unnecessary presence of different middleware technologies in the communication among services):*
  - *Provide a common middleware that can be used by all services:* The communication layer is a middleware technology. The standardization of the middleware used by the services represents a consensus between practitioners, leading to a more robust and reliable solution, while they do not need to worry with different technologies for doing the same job;
  - *Rewrite services to use the same middleware:* The services should be updated to be compatible with the new communication layer.

From the list above, we identify the need of some rewrite or refactoring in the communication layer and in the services. As it is a complex system, that work must be properly advised, and that cannot be done without paying the following extra principal costs:

- *Define and execute a governance plan to handle the migration:* The migration of services in the entire company requires a coordinated effort that should be driven by the company. There are extra costs to (i) define the plan, (ii) define full-time employees (project managers) to execute the migration on each project, and (iii) supervise the migration in a company level in accordance to the plan;
- *Maintain the system working with different solutions during the migration period:* Migrations in a system running hundreds of services should happen step by step. In the meanwhile, the total environment is more complex than before due to the existence of all past technologies plus the new one. There is a temporary increase in the cost for maintaining the complex environment until the migration is finished;
- *New requirements should be down prioritized:* Once the refactor or rewrite of the system and the communications layer is a priority, the teams need to reduce the time spent on new functionalities to work with the priorities. The cost for this issue is the possible amount of money lost by not having the new requirements;
- *Service developers must learn new technologies:* Teams that are used to some technologies will be forced to migrate to a new solution for which they do not have the proper technical knowledge. It is acceptable to consider that they will work slower in the beginning than if they were working with the technologies in which they have full experience.

D. What are the risks in refactoring ATD in microservices? (RQ2)

The company identified certain risks before the migration to help prevent unexpected problems.

- *Communication layer more complex than before if canonical models are not properly supervised:* If the company does not inspect the creation and use of canonical models, teams can try to add information that should not be there, or that violates their definition if the company defined them before. That can lead to complexity in the communication layer. The company knows this issue and is properly policing the creation of these models;
- *Not sufficient funding:* If there are no funds to finish the migration, the company can make the environment even worse, by having all the old technologies plus a new one to deal with. They mitigated this risk by realizing a previous analysis of the costs of the migration;
- *Migration possibly halted before finished:* If the management decides about halting the migration before it finishes, the same problem as the previous point will happen, having an environment more complex than before. The company mitigated this risk by preparing a plan and previously discussing with their managers;
- *Migration possibly not prioritized:* The change for the new solution must be done by all services. Only then will the migration be finished and the whole environment will be using the same solution. To mitigate this issue, the company proposed the entire change in agreement with the several sectors in the company.

V. Discussion

A. Implications for Research

We contribute to the state of the art by presenting the relationship between ATD and microservices, not previously explored. We believe this study will stimulate more investigations into the relationship among ATD, microservices and SOA. We focus especially on the costly ATD that can be accumulated in the communication layer of a microservices architecture. Together with a list of ATD issues, we also identify their principal and interest. Although this list cannot be considered complete, we compiled a novel body of knowledge on ATD in microservices. More research is needed regarding other aspects of microservices, such as the ATD related to the allocation of cross-cutting concerns in microservices, their deployment, etc.

Furthermore, we provided a set of clearly defined ATD issues for which is possible to define metrics in order to measure debt, principal and interest. For example, how to measure the number of point-to-point connections, the cost of having too many standards in the communication among services, and the cost of removing the business logic from the communication layer. We are in the process of further studying the case in order to compile and validate such measures.

In addition, we present a list of risks that should be taken in consideration by researchers to develop a risk assessment model to be used when taking decisions about TD refactoring.

B. Implications for Industry

The results presented here can be useful to help practitioners when developing systems with a microservices architecture. These insights can be used to raise awareness of possible

ATD issues related to microservices. We present a list of ATD issues together with their cost of refactoring and the associated extra negative impact, summarized in quick reference table complemented with a detailed explanation. These findings can help developers and architects to avoid the accumulation of ATD issues before they become too costly. Additionally, they can help giving targets for refactoring and possible concrete solutions that have been fruitful in other large organizations. For example, practitioners might want to avoid too many point-to-point connections, the existence of business logic in the communication layer, or plan refactoring accordingly. As a microservices system grows, the number of services affected by the ATD present in the communication layer also grows. This means that both principal and interest might grow linearly (or even worse), making it more expensive both the presence and the repayment of the debt. As such, being aware of these issues in early stages is important for the health of the organization. Likewise, the solution proposed in the current context may be reused and studied as a possible approach in other companies.

We also presented a list of risks that are being considered by the company during the migration process. These findings can help managers to handle similar initiatives, avoiding situations that can lead to unfinished migrations and consequent increase of complexity in the system.

It is important to mention that we also found other type of TD in our investigation as, for example, testing TD. Such issues have not been reported here because they either they do not represent ATD, or they are not related specifically to microservices, the core of our investigation.

C. Threats to Validity

When mapping the interview data to the categories we used for coding, there was a challenge that different interviewees did not necessarily agree on the solution to a problem. To mitigate this threat, we triangulated the information from multiple sources of evidence, including people and documents provided by the company, to confirm the validity of the data. We only considered in our results data that could be attested by most of the sources.

A possible threat is that the issues we thought affected the principal and interest of the ATD did not really do it. We mitigated this threat in the creation of the questionnaire for the interviews, making sure we asked about the negative impact and their related costs for multiple interviewees.

This study was is conducted in a single organization. To improve the external validity of the findings, studies in other organizations are needed. In addition, we have covered only the problems related to the communication layer, and not related to other aspects of microservices, but we wanted to collect a rich amount of details (debt, principal, interest etc.) with a higher level of triangulation among the data sources.

To help ensure reliability, three researchers were present during the interviews (observer triangulation), and we investigated several sources of evidence (source triangulation). All results were checked by multiple researchers.

VI. Related work

Most research on ATD is recent. The area is challenging and many more studies are needed [11] [15]. Kazman et al. [16] conducted a case study in a software development organization looking for what they call the roots of architectural problems. They did not look at microservices and we need more research to understand if such approach can work with in that context.

In a more recent study, Martini et al. [15] conducted a case study in six large international software companies to investigate the interest generated by ATD issues. They present a taxonomy of the most dangerous issues and coin the definition of contagious debt, a chain of events responsible for increasing the TD in the system. While they investigate ATD in general, we focus on the specific context of microservices.

Another work from Martini et al. [17] was conducted in a large software company. ADT was automatically identified through architectural smells. They looked at the cost/benefit of the modularization of a component. We should try that methodology in the context of microservices hereafter.

In our study, we were unable to identify architecture smells specific to microservices due to insufficient tool support.

Microservices architecture is being widely adopted especially by those who are facing problems with previous software monoliths due to their increasingly complexity, like the case we investigated in this work. An example is the work of Bucchiarone et al. [18], which presents an experience report from a migration of a monolith to microservices, also in the financial services domain. They focus in the migration process, presenting what they did and their reasons to proceed. We, by contrast, investigate ATD in an existing microservices architecture, a further step.

A systematic mapping study by Francesco et al. [19] found that most research on microservices architecture consists of solution proposals and their validation, and that there are several gaps on industry- and practitioners-oriented research as, for instance, the lack of evaluation research on microservices, testability and security. In addition, they did not find studies that investigate the presence of ATD on microservices. We, therefore, provide such new evidence.

Taibi et al. [20] presented a catalog of bad smells on microservices. Despite there is a relationship between ATD and architectural smells, they are not the same concept. Smells can be used to identify the amount of debt, but not its interest and principal. Thus, our work is differs because they only define what are the bad smells in the microservices context, and we present ATD and discuss their principal and interest. In our investigation, we found a problem similar to what they call too many standards in the communication layer: in our case we have no defined standards in the communication, leading to too many different terminologies being used, while only one should be enough. We called that the “Tower of Babel” problem. We did not find other issues presented by them in our case, but the list of ATD issues in our work lead to data they were not aware. So, our studies complement each other.

VII. Conclusions and Future Work

We investigated what is ATD in microservices architectures with focus on principal and interest through case study in a large financial services company. We focused on the communication layer because a large set of refactoring were initiated in that area due to the high interest paid. We evaluated the system before and after the refactoring enabling us to identify the interest and principal associated with ATD.

We found the ATD issues related to microservices as follows: the existence of too many point-to-point connections among services, the presence of business logic in the communication layer, the lack of standards in the communication among services, weak source code and knowledge management and unnecessarily many different technologies used by the service developers in the communication among services. These issues caused substantial interest (of ATD), such as need of rewriting the code, extra effort to handle different technologies, and coupling between services. Some of the principal found were creation of canonical models according to the domain of the data, definition of a single middleware layer and removal of business logic from the communication layer. We also found risks that can lead to an unfinished refactoring process, and presented information that may help practitioners to mitigate them.

Our findings may help developers and architects avoid ATD issues in advance, before the cost of fixing them increases, and identify areas of the code that may need refactoring. Our results contribute to an increased understanding of the relationship between ATD and microservices.

Our future work includes analyzing additional cases, investigating metrics to measure debt, principal and interest in microservices architecture, and quantify costs and benefits related in this context. We also plan to investigate the existence of ATD in other areas than the communication layer.

References

[1] A. Martini, T. Besker, and J. Bosch, "Technical Debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations," *Science of Computer Programming*, vol. 163, pp. 42–61, oct 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642318301035

[2] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri, and T. Lynn, "Microservices migration patterns," *Software: Practice and Experience*, vol. 48, no. 11, pp. 2019–2042, 2018. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2608

[3] H. Vural, M. Koyuncu, and S. Guney, "A Systematic Literature Review on Microservices," in *Computational Science and Its Applications – ICCSA 2017*, O. Gervasi, B. Murgante, S. Misra, G. Borruso, C. M. Torre, A. M. A. C. Rocha, D. Taniar, B. O. Apduhan, E. Stankova, and A. Cuzzocrea, Eds. Cham: Springer International Publishing, 2017, pp. 203–217. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-62407-5_14

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

[5] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 1st ed. O'Reilly Media, Inc., 2017.

[6] M. P. Papazoglou, "Service-oriented computing: concepts, characteristics and directions," in *Proceedings - 4th International Conference on Web Information Systems Engineering, WISE 2003*. IEEE Comput. Soc, 2003, pp. 3–12. [Online]. Available: http://ieeexplore.ieee.org/document/1254461/

[7] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, "Microservices in Practice, Part 1: Reality Check and Service Design," pp. 91–98, jan 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7819415/

[8] G. Hohpe and B. WOOLF, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, ser. The Addison-Wesley Signature Series. Prentice Hall, 2004. [Online]. Available: http://books.google.com.au/books?id=dH9zp14-1KYC

[9] W. Cunningham, "The WyCash portfolio management system," *SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1992. [Online]. Available: http://doi.acm.org/10.1145/157710.157715

[10] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)," *Dagstuhl Reports*, vol. 6, no. 4, pp. 110–138, 2016. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6693

[11] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Software*, vol. 29, no. 6, pp. 18–21, 2012.

[12] PMI, Ed., *A Guide to the Project Management Body of Knowledge (PMBOK Guide)*, 5th ed. Newtown Square, PA: Project Management Institute, 2013.

[13] AXELOS, *Managing Successful Projects with PRINCE2*. Stationery Office, 2017.

[14] U. Flick, *An Introduction to Qualitative Research*. SAGE Publications, 2009.

[15] A. Martini and J. Bosch, "On the interest of architectural technical debt: Uncovering the contagious debt phenomenon," *Journal of Software: Evolution and Process*, vol. 29, no. 10, pp. 1–18, 2017.

[16] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka, "A Case Study in Locating the Architectural Roots of Technical Debt," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, may 2015, pp. 179–188. [Online]. Available: http://ieeexplore.ieee.org/document/7202962/

[17] A. Martini, F. A. Fontana, A. Biaggi, and R. Roveda, "Identifying and Prioritizing Architectural Debt through Architectural Smells: a Case Study in a Large Software Company," in *12th European Conference on Software Architecture, ECSA*, Madrid, Spain, 2018.

[18] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, "From Monolithic to Microservices: An Experience Report from the Banking Domain," *IEEE Software*, vol. 35, no. 3, pp. 50–55, may 2018.

[19] P. D. Francesco, I. Malavolta, and P. Lago, "Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption," in *2017 IEEE International Conference on Software Architecture (ICSA)*, apr 2017, pp. 21–30. [Online]. Available: https://ieeexplore.ieee.org/document/7930195/

[20] D. Taibi and V. Lenarduzzi, "On the Definition of Microservice Bad Smells," *IEEE Software*, vol. 35, no. 3, pp. 56–62, may 2018.