From Monolithic Systems to Microservices: A Decomposition Framework based on Process Mining

Davide Taibi[a] and Kari Systä[b]
TASE - Tampere Software Engineering Research Group, Tampere University. Tampere, Finland

Keywords: Microservices, Cloud-native, Microservice Slicing, Microservice Decomposition, Microservice Migration.

Abstract: Decomposition is one of the most complex tasks during the migration from monolithic systems to microservices, generally performed manually, based on the experience of the software architects. In this work, we propose a 6-step framework to reduce the subjectivity of the decomposition process. The framework provides software architects with a set of decomposition options, together with a set of measures to evaluate and compare their quality. The decomposition options are identified based on the independent execution traces of the system by means of the application of a process-mining tool to the log traces collected at runtime. We validated the process, in an industrial project, by comparing the proposed decomposition options with the one proposed by the software architect that manually analyzed the system. The application of our framework allowed the company to identify issues in their software that the architect did not spot manually, and to discover more suitable decomposition options that the architect did not consider. The framework could be very useful also in other companies to improve the quality of the decomposition of any monolithic system, identifying different decomposition strategies and reducing the subjectivity of the decomposition process. Moreover, researchers could extend our approach increasing the support and further automating the decomposition support.

1 INTRODUCTION

Legacy and monolithic system have become hard to maintain because of tight coupling between their internal components. Modifying a feature in one class often involves changes in several other classes, thereby increasing the needed development time and effort. The decomposition into small and independent modules is a strategy that companies may adopt to improve maintainability (Parnas, 1972) (Soldani et al., 2018). Often, at the same time, companies want to utilize benefits of microservices, such as independent development, scaling and deployment (Taibi et al., 2017d).

Microservices are relatively small and autonomous services deployed independently, with a single and clearly defined purpose (Fowler and Lewis, 2014). Because of their independent deployment, they have a lot of advantages. They can be developed in different programming languages, they can scale independently from other services, and they can be deployed on the hardware that best suits their needs. Moreover, because of their size, they are easier to maintain and more fault-tolerant since a failure of one service will not break the whole system, which could happen in a monolithic system. Since every microservice has its own context and set of code, each microservice can change its entire logic from the inside, but from the outside, it still does the same thing, reducing the need of interaction between teams (Taibi et al., 2017b) (Taibi et al., 2017c).

However, decomposing a monolithic system into independent microservices is one of the most critical and complex tasks (Taibi et al., 2017d)(Taibi et al., 2017e) and several practitioners claim the need for a tool to support them during the slicing phase in order to identify different possible slicing solutions (Taibi et al., 2017d) (Taibi et al., 2018). The decomposition is usually performed manually by software architects (Taibi et al., 2017d)(Soldani et al., 2018). Up to now, the only help that software architects can have is based on the static analysis of dependencies with tools such as Structure 101[1] while the slicing of the system commonly is delegated to the experience of the

[a] https://orcid.org/0000-0002-3210-3990
[b] https://orcid.org/0000-0001-7371-0773

[1] Structure101 Software Architecture Environment - http://www.structure101.com

Taibi, D. and Systä, K.
From Monolithic Systems to Microservices: A Decomposition Framework based on Process Mining.
DOI: 10.5220/0007755901530164
In *Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER 2019)*, pages 153-164
ISBN: 978-989-758-365-0

software architect itself. Moreover, static dependency analysis tools are not able to capture the dynamic behavior of the system and run-time dependencies like frequent method calls could have an influence to both maintainability and performance. Thus, we decided to approach the slicing based on runtime behavior instead of only considering static dependencies.

In order to ease the identification of microservices in monolithic applications, we adopted a data-driven approach for identifying microservices candidates based on process mining performed on log files collected at runtime. Our approach combines process mining techniques and dependency analysis to recommend alternative slicing solutions. Our decomposition approach can be used by software architects to support their decisions and to help them easily identify the different business processes in their applications and their dependencies, reducing the subjectivity and the risks of related to the slicing process.

We validated this work with an industrial case study performed in collaboration with an SME that we supported in the migration phase, comparing the decomposition solution proposed by the software architect with the one obtained from the application of our process-mining based approach.

The results show that process mining can be effectively used to support the decomposition of microservices and that it also supports the identification of existing architectural issues in monolithic systems. The result can be used by companies to reduce the risk of a wrong slicing solution, suggesting different slicing options to the software architects and providing additional analysis of the software asset.

This paper is structured as follows. Section 2 presents the background on processes for migrating and splitting monolithic systems into microservices. Section 3 describes our proposed approach. Section 4 reports on the industrial case study. Section 5 discusses the results, while Section 6 draws conclusions.

2 BACKGROUND AND RELATED WORK

Decomposing a system into independent subsystems is a task that has been performed for years in software engineering. Parnas (Parnas, 1972) proposed the first approach for modularizing systems in 1972. After Parnas's proposal, several works proposed different approaches (Lenarduzzi et al., 2017a). Recently, the decomposition of systems took on another dimension thanks to cloud-native systems and especially microservices. In microservices, every module is developed as an independent and self-contained service.

2.1 The Microservice Decomposition Process

Taibi et al. (Taibi et al., 2017d) conducted a survey among 21 practitioners who adopted microservices at least two years ago in order to collect their motivation for, as well as the evolution, benefits, and issues of the adoption of microservices. Based on the results, they proposed a migration process framework composed of two processes for the redevelopment of the whole system from scratch and one process for creating new features with a microservice architecture on top of the existing system. They identified three different processes for migrating from a monolithic system to a microservices-based one. The goal of the first two processes is to support companies that need to migrate an existing monolithic system to microservices by re-implementing the system from scratch. The aim of the third approach is to implement new features only as microservices, to replace external services provided by third parties, or to develop features that need important changes and therefore can be considered as new features, thus gradually eliminating the existing system. All three of the identified processes are based on four common steps but differ in the details.

- *Analysis of the System Structure*. All processes start by analyzing dependencies mainly with the support of tools (Structure101, SchemaSpy [2], or others)
- *Definition of the New System Architecture*. Architectural guidelines or principles, and proposal of a decomposition solution into small microservices are defined. The decomposition is always done manually.
- Prioritization of feature/service development. In this step, all three processes identify and prioritize the next microservices to be implemented. Some processes prioritize microservices based on customer value; others according to components with more bugs; and yet others prioritize the development of new features as microservices, expecting that, in the long run, the new ecosystem of microservices will gradually replace each feature of the existing monolith.
- *Coding and Testing* are then carried out like any other software development project. Developers adopt the testing strategy they prefer. However, in some cases, testing of the different microservices is performed by doing unit testing at the microservices level and black-box testing at the integration level.

[2]http://schemaspy.sourceforge.net/

In this work, we focus mainly on the first two steps, supporting companies in the analysis of the system structure and in the identification of decomposition alternatives. The architectural guidelines should be defined by the company based on their internal policies.

2.2 Proposed Approaches for Identifying Microservices

Only a limited set of research works propose approaches aimed at supporting developers in decomposing their systems into an optimal set of microservices.

Abbott and Fischer (Abbott and Fisher, 2015) proposed a decomposition approach based on the ”scalability cube”, which splits an application into smaller components to achieve higher scalability. Richardson (Richardson, 2017) also mentioned this approach in his four decomposition strategies:

- ”Decompose by business capability and define services corresponding to business capabilities”;
- ”Decompose by domain-driven design subdomain”;
- ”Decompose by verb or use ‘cases’ and define services that are responsible for particular actions”;
- ”Decompose by nouns or resources by defining a service that is responsible for all operations on entities/resources of a given type”.

The first two strategies are mostly abstract patterns of human decisions (Yourdon and Constantine, 1979) while the others are based on predefined criteria. Kecskemeti et al. (Kecskemeti et al., 2016) proposed a decomposition approach based on container optimization. The goal is to increase the elasticity of large-scale applications and the possibility to obtain more flexible compositions with other services.

Arndt and Guercio suggest decomposing a monolith system using a layered architecture style, with the outcome being highly cohesive and loosely coupled services, such as representation and business services. Another possibility is to start from a monolithic system and progressively move towards a microservices-based architecture (Zimmermann, 2017) or delivering separate microservices by splitting a development team into smaller ones responsible for a limited group of microservices.

Vresk et al. (Vresk and Cavrak, 2016) defined an IoT concept and platform based on the orchestration of different IoT system components, such as devices, data sources, data processors, and storage. They recommend combining verb-based and noun-based decomposition approaches. The proposed approach hides the complexity stemming from the variation of end-device properties thanks to the application of a uniform approach for modeling both physical and logical IoT devices and services. Moreover, it can foster interoperability and extensibility using diverse communication protocols into proxy microservice components. Gysel et al. (Gysel et al., 2016) proposed a clustering algorithm approach based on 16 coupling criteria derived from literature analysis and industry experience. This approach is an extensible tool framework for service decomposition as a combination of a criteria-driven methods. It integrates graph clustering algorithms and features priority scoring and nine types of analysis and design specifications. Moreover, this approach introduces the concept of coupling criteria cards using 16 different instances grouped into four categories: Cohesiveness, Compatibility, Constraints, and Communications. The approach was evaluated by integrating two existing graph clustering algorithms, combining actions research and case study investigations, and load tests. The results showed potential benefits to the practitioners, also confirmed by user feedback.

Chen et al. (Chen et al., 2017) proposed a data-driven microservices-oriented decomposition approach based on data flow diagrams from business logic. Theyr approach could deliver more rational, objective, and easy-to-understand results thanks to objective operations and data extracted from real-world business logic. Similarly, we adopt process mining to analyze the business processes of a monolithic system.

Alwis et al. (De Alwis et al., 2018) proposed a heuristic to slice a monolithic system into microservices based on object subtypes (i.e., the lowest granularity of software based on structural properties) and functional splitting based on common execution fragments across software (i.e., the lowest granularity of software based on behavioral properties). This approach is the closer to our work. However, they analyzed the system by means of static analysis without capturing the dynamic behavior of the system and they did not propose measures to evaluate the quality of the slicing solution proposed.

Taibi et al., proposed a set of patterns and anti-patterns that should be carefully considered during the microservice decomposition (Taibi and Lenarduzzi, 2018) (Taibi et al., 2019) recommending to avoid a set of harmful practices such as cyclic dependencies and hard-coded endpoints but also to consider critical anti-patterns and code smells (Taibi et al., 2017a) that can be generated into the monolithic system.

3 THE DECOMPOSITION FRAMEWORK

Applications built from microservices should be as decoupled and as cohesive as possible (Fowler and Lewis, 2014). In the case of loosely coupled services, changes to one service should not require changes to other services. Therefore, the developers of microservices can change and deploy their microservices independently. As reported by Sam Newman (Newman, 2015), "a loosely coupled service knows as little as it needs to about the services with which it collaborates.". Therefore, developers should limit the number of different types of calls from one service to another.

Cohesion is the degree to which the elements of a certain class belong together. It is a measure of how strongly related each piece of functionality of a software module is (Fenton and Bieman, 2014). High cohesion makes the reasoning easy and limits the dependencies (Kramer and Kaindl, 2004). Low coupling is commonly correlated with high cohesion (Kramer and Kaindl, 2004) (Jabangwe et al., 2015). In microservices-based systems, low cohesion is achieved by grouping common business processes together, so that, if developers need to change the behavior, they need to change only a single microservice (Newman, 2015). Practitioners commonly analyze dependencies with tools such as Structure 101. However, while dependency analysis tools can support the identification of static dependencies, they do not enable the identification of the full execution path. Our approach combines process mining techniques and dependency analysis to recommend alternative slicing solutions. In the next sub-sections, we report the underlying assumptions of our approach and the different steps that compose the decomposition process.

3.1 Assumptions

The approach requires the availability of an extended version of a log trace collected at runtime. For each user operation performed from the user interface (e.g., clicking on a button), or from any entry point of a system (e.g., APIs or command line), all the activities must be traced from the log files. Information about each class and method that is traversed for the execution of the operation must be included. The complete execution must be traced completely from the entry point (a click on a submission form or the beginning of an operation) to the access to the database (if any) and to the results returned to the client. An example of data reported in the log file is shown in Table 2. In this step, we instrument the software to produce a log. The log trace must include events that indicate entrance and exit of functions as well as database accesses. Each entry in the log also has a timestamp and a session ID, to distinguish between the usage of the same class or method from different users. The log trace could be collected by instrumenting the source code with Aspect Oriented Programming, by adding the log trace into each method or with existing applications such as the Elastic APM [3] or similar, or adopting an approach similar to the one applied in (Suonsyrjä, 2015). In case the data collection is not yet in place, we recommend to use Elastic APM, since it allows to easily instrument the code with a minor effort. For some languages (e.g. Java and Node.js) the instrumentation requires the addition of one line of code to the application configuration, specifying the type of log trace required and the logging server URL.

3.2 The Decomposition Process

Once the log files are created, companies can start the decomposition following our 6-step process (Figure 1).

Step 1: Execution Path Analysis
In the first step, we identify the most frequently used execution paths with a process-mining tool. In our case, DISCO [4] was used to graphically represent the business processes by mining the log files. The same result can be obtained by any other alternative process-mining tool. The result is a graphical representation of the processes, reporting each class and database table used in the business processes, with a set of arrows connecting each class based on the log traces. The result of this first step produces a figure similar to the one presented in Figure 2, that allows to understand:

- Runtime execution paths of the system. Paths never used, even if possible, are not represented in the figure.
- Dependencies between the classes of the system. The arrows represent the dependencies between methods and classes. External dependencies to libraries or web-services are also represented.
- The frequency of usage of each path. Process mining tools present the most used processes with thicker arrows
- Branches and Circular Dependencies. The graphical representation allows easy discovery of circular dependencies or branches (e.g., conditional

[3]The Elastic APM Libraries. https://www.elastic.co/solutions/apm

[4]https://fluxicon.com/disco/

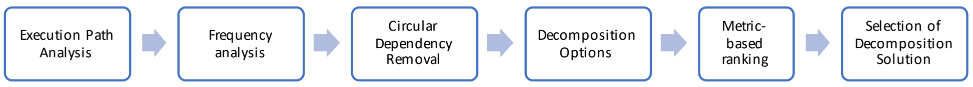

Figure 1: The Decomposition Process.

Table 1: Frequency analysis of each execution path.

Path	Freq.
A.a(); A.b(), B.b(), C.c(), DB.query, Table A, Table B, ...	1000
A.b(); A.c(), B.a(), C.c(), DB.query, Table A, Table B, ...	150

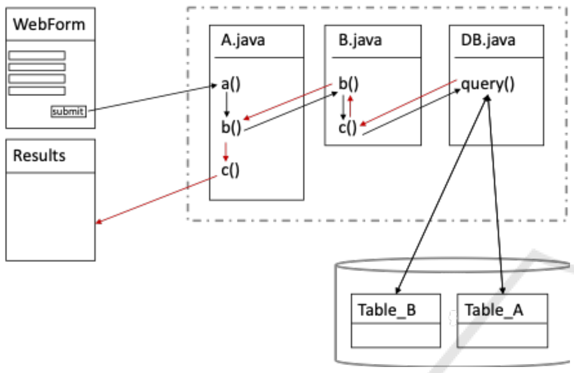

Figure 2: Simplified Process Example.

statement that led to different path based on the input provided), in case they exist.

The complete chain of arrows forms a candidate of a process. Figure 2. represents a simplified example of one business process representing the data reported in Table 2.

Step 2: Frequency Analysis of the Execution Paths

The thickness of the arrows created by the DISCO tool indicates the frequency of the calls between classes. This makes it possible to clearly understand which execution path are used most frequently and which classes are rarely or never used during runtime. The output of this step is a table representing all the

Table 2: Example of Log Traces (Timestamps are shortened for reasons of space).

Start Time	End Time	Sess.ID	Class	Method
00:00	00:36	S1	Form.jsp	btnClick()
01:00	01:39	S1	A.java	a()
01:40	01:45	S1	A.java	b()
01:45	01:55	S1	B.java	b()
01:56	02:05	S1	B.java	c()
02:05	02:13	S1	DB.java	query()
02:14	02:21	S1	DB	TABLE A
02:22	03:28	S1	DB	TABLE B
02:29	02:36	S1	B.java	c()
02:36	02:45	S1	vB.java	b()
02:46	02:55	S1	A.java	b()
02:56	03:03	S1	A.java	c()
03:04	03:16	S1	Results.jsp	render()

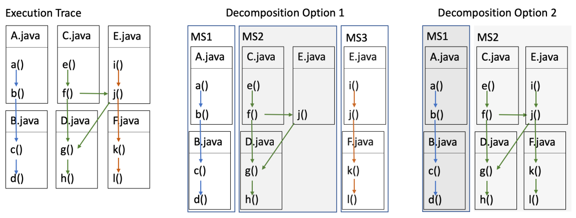

Figure 3: Simplified Process Example.

different execution paths with the frequency of their usage.

Step 3: Removal of Circular Dependencies

In this step, we first find circular dependencies by analyzing the execution paths reported in the table generated in the first Step (e.g. Table 2). This can be done with a simple algorithm to discover cycles in the execution paths. In the case of circular dependencies, software architects should discuss with the development team how to break these cycles. One example of the patterns that can be applied to break the cycles is Inversion of Control (Martin, 2003). However, every cyclic dependency could need a different breaking solution that must be analyzed carefully. The result is a refined version of the execution path table (see Table 2 as example).

Step 4: Identification of Decomposition Options

Starting with the execution paths without cyclic dependencies obtained from Step 3, we identify different decomposition alternatives by visually inspecting the generated graphs. The candidate processes may have common sub-paths, i.e., the processes may merge or split. Thus, different decomposition solutions are possible. This process could also be automated by developing an algorithm that provides all different decompositions based on the paths with fewer intersections. However, in this case, we rely on the expert-based decomposition. As highlighted in Figure 3, the decomposition options need to deal with the duplication of some classes or methods. As example, the execution traces reported in Figure 3 show that both the green and the orange execution traces use j(). Therefore, software architects could propose two decomposition alternatives. The first option includes the creation of three microservices where class E.java() is duplicated in microservice MS2 and MS3. The second option includes the creation of two microservices, merging MS2 and MS3. Both options have pros and cons, but the decision of merging two execution traces or splitting into different microservices must be discussed with the team. If two microservices candidates for the splitting have different purposes, it is reasonable to consider the splitting. If they are doing the same thing, then it would be better to merge them into one single microservice.

Step 5: Metric-based Ranking of the Decomposition Options

In this step, we identify three measures to help software architects to assess the quality of the decomposition options identified in Step 4: Coupling, Number of classes per microservices, Number of classes that need to be duplicated.

Coupling

The decomposition to microservices should minimize coupling and maximize cohesion. Coupling and cohesion can be calculated with different approaches. While coupling can be obtained from our log traces, for all the cohesion measures we also need to know about the access to the local variables of each class, which makes it impossible to calculate them from the data reported in the log traces. However, coupling is commonly considered as inversely proportional to cohesion (Jabangwe et al., 2015). Therefore, a system with low coupling will have a high likelihood of having high cohesion (Jabangwe et al., 2015). We define the Coupling Between Microservice (CBM) extending the well-known Coupling Between Object (CBO) metric proposed by Chidamber and Kemerer (Chidamber and Kemerer, 1994). CBO represents the number of classes coupled with a given class (efferent couplings and afferent couplings). This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions.

We calculate the relative CBM for each microservice as follows:

$$\text{CBM}_{MS_j} = \frac{\text{Number of external Links}}{\text{Number of Classes in the Microservice}}$$

where "Number Of External Links" represents the number of calls to external services used in each class of microservice. An external service linked several times by different classes of the same microservice is only counted once. External services could be other microservices, external APIs, etc.

CBM is calculated for each microservice independently and presented in a table for the next step.

Number of Classes per Microservice

This measure helps to understand how big the microservice identified is and to identify if there are microservices that are too big compared to others. the number of classes should be minimized since the smaller the number of classes the more independent its development can be. Considering the example reported in Figure 3, the decomposition option 1 has 7 classes while option 2 has six classes.

Number of Classes that Need To Be Duplicated

In some cases, several classes will be in common between two execution traces. As example, the method j in Class E (Figure 3) is used by two execution traces. In the example depicted in Figure 3, decomposition option 1 has one class that needs to be duplicated, while option 2 requires no classes to be duplicated.

This measure helps to reason about the different slicing options, considering not only the size of the microservices but also the number of duplications, that will be then reflected in the development of the microservices. Duplicated classes should be avoided since the duplication adds to the size of the system and its maintenance.

Step 6: Selection of the Decomposition Solution

This is the final step where, based on the different decomposition alternatives identified in Step 4, and on the measures collected in Step 5, software architects can decide which solution to adopt by merging or splitting existing processes. Software architects could consider the recommendation provided by our decomposition process and discuss with the team which solution is most suitable for them, considering the organizational structure. Our process does not recommend the best decomposition solution, but provides a reasoning framework on the possible decomposition options.

4 VALIDATION: INDUSTRIAL CASE STUDY

In this section, we validate the decomposition framework proposed in Section III. With this study, we aim to understand whether our approach can support developers in easily identifying different slicing options. For this purpose, we performed an industrial case study to compare the ad-hoc decomposition solution proposed by the software architect with the solutions proposed by our approach.

According to our expectations, we formulated the goal of the case study as:
Analyze the proposed decomposition solutions
for the purpose of evaluating and comparing
with respect to the perceived slicing easiness and usefulness
in the context of the migration of a monolithic system to microservices
from the point of view of software architects.

We formulated the questions of the case study as follows and further derived questions and metrics from them:

RQ1. Q1: Does adopting the proposed decomposition framework **ease** the identification of different microservices?

RQ2. Q2: What do the developers, software architects, and project manager think about the **applicability** of this approach?

RQ3. Q3: Are the developers willing to **use** the approach in the future?

We answered our questions by surveying the project manager and the software architect who first manually applied the decomposition process as usual and then evaluate the decomposition options proposed by our framework. The measures identified for the questions were derived from the Technology Acceptance Model (Venkatesh, 2000). All questions were evaluated based on a 5-point ordinal Likert scale with the following options: 1 = strongly disagree, 2 = disagree, 3 = neither agree nor disagree, 4 = agree, 5 = strongly agree.

Q1 – Perceived Ease of Use: Here we aim to compare the perceived easiness of our approach with that of the experience-based (ad-hoc) approach. We adopted the Technology Acceptance Model (Venkatesh, 2000) to collect measures about the ease of use of our approach, identifying the following metrics:

- The process-mining approach would be easy for me to use during the decomposition phase.
- It would be easy for me to become skillful at using the process-mining approach to decompose a monolithic system.

Q2 - Applicability: What do the participants think about the applicability of our approach? To answer this question, we collected the time overhead needed to perform the process-mining approach. Perceived usefulness: measures the degree to which the participants considered the approach useful for making project decisions. The evaluated criteria were:

- I am sure that I was able to better decompose the system with this approach.
- I was able to find alternative decomposition strategies.
- I was able to better decompose the system, but the time required with the new approach is too much compared to its benefits.
- I was able to better decompose the system, but the effort needed to trace the information on the log file is too much compared to the benefits of the approach.
- The approach helped me to understand existing architectural issues in the existing monolithic system

Perceived Understandability: measures the effort needed by the subject to understand the approach built or whether the participants will need to exert little effort to understand the relationship with the system concepts.

- It was easy for me to understand how the approach works. Perceived easiness: measures the degree to which the subject believed that he or she was able to make project decisions easier than without the approach.
- It was easy for me to identify decomposition options with the support of this approach.
- I was able to identify decomposition options with less effort compared to the ad-hoc manual decomposition.
- I was able to identify decomposition options more accurately.

Self-efficacy by Applying the Technique: The perceived ability to decompose a monolithic system into microservices by means of our proposed approach.

- It was easy for me to keep an overview of the project and of the different decomposition options.
- The approach helped me to increase the quality of the decompositions.

Q3 - Willingness to Use our Approach in the Future: With this question, we aim to understand whether the company would be willing to use our system in the future. We collected this measure with the following question:

- I will adopt this approach in the future.

Table 3 report the list of questions and the results of this study.

4.1 Study Context

The approach was applied in an SME in Milan (Italy). The company develops a document management system for bookkeeping, for Italian tax accountants. The goal of the system is to make it possible to manage the whole bookkeeping process, including management of the digital invoices, sending the invoice to the Ministry of Economic Development, and fulfilling all the legal requirements, which usually change every year.

The system is developed by two teams of 4 developers, plus two part-time developers following the moonlight scrum process (Taibi et al., 2013), the software architect and a project manager. currently being used by more than 2000 tax accountants, who need to store more than 20M invoices per year. The system has been developed for more than 12 years and is now composed of more than 1000 Java classes.

Table 3: The Questionnaire adopted in this study - Results.

Questions		**Metrics**	**Project Manager**	**Software Architect**
Q1 – Perceived ease of use		The proposed approach would be easy for me to use in the decomposition phase.	4	5
		It would be easy for me to become skillful at using the process-mining approach to decompose a monolithic system.	3	4
Q2 - Applicability	*Perceived usefulness*	I am sure that I was able to better decompose the system with this approach.	5	5
		I was able to find alternative decomposition strategies.	5	5
		I was able to better decompose the system but the time required with the new approach is too much compared to its benefits.	4	4
		I was able to better decompose the system but the effort needed to trace the information on the log file is too much compared to the benefits of the approach.	2	1
		The approach helped me to understand existing architectural issues in the monolithic system.	4	5
	Perceived understandability	It was easy for me to understand how the approach works.	4	4
	Perceived easiness	It was easy for me to identify decompositions options with the support of this approach.	3	4
		I was able to identify decomposition options with less effort compared to the ad-hoc manual decomposition.	5	4
		I was able to identify decomposition options more accurately.	5	5
	Self-efficacy	It was easy for me to keep an overview of the project and of the different decomposition options.	4	4
		The approach helped me to increase the quality of the decompositions.	5	4
Q3 - Willingness to use our approach		I will adopt this approach in the future.	3	3

The Italian government usually updates the bookkeeping process between December and January of every year, which involves not only changing the tax rate but also modifying the process of storing the invoices. However, tax declarations can be made starting in March/April of each year. Therefore, in the best case, the company has between two to four months to adapt their software to the new rules in order to enable tax accountants to work with the updated regulations from March/April.

Up to now, the company used to hire a consultancy company to help them during these three months of fast-paced work. However, since the system is growing year after year, they decided to migrate to microservice to facilitate maintenance of the system (Saarimäki et al., 2019) and to distribute the work to independent groups, reducing communication needs and supporting fast delivery (Taibi et al., 2017c)(Taibi et al., 2017b).

4.2 Study Execution

We performed this study in three steps:

Step 1. Q1: The software architect manually identified the different microservices and a decomposition strategy.

Step 2. Q2: We applied our 6-steps process to identify different decomposition solutions and then we compared them to the solution proposed by the software architect.

Step 3. Q3: The software architect and the project manager provided feedback on the usefulness of our approach.

4.3 Case Study Results

With the support of the development team, the software architect manually identified a set of microservices. He first drew the dependency graph with Structure 101.

Then we applied our approach to mine their log files. The company already logged all the operations of their systems with Log4J [5], tracing all the information reported in Table 2, together with other information such as the user involved in the process, the ID of the customer with which the tax accountant is currently working, and other information related to the current invoices.

From this information, we identified 39 different business processes with DISCO. For confidentiality reasons, we can only disclose a high-level and anonymized portion of the system. Figure 4 depicts an example of two business processes (save invoice and view invoice). We then calculated the frequency of each process. DISCO can automatically draw thicker arcs between processes, thereby simplifying the approach.

Of 39 processes, three processes had been used only three times during one year of logging, 17 pro-

[5]https://logging.apache.org/log4j

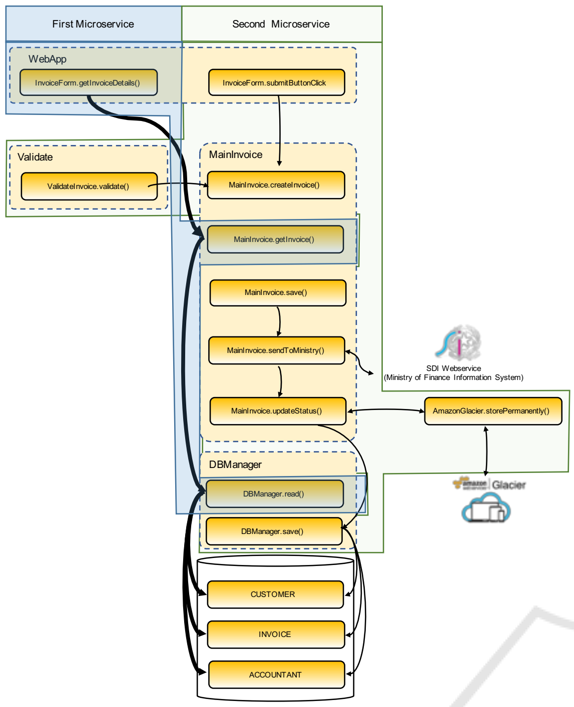
First Microservice
Second Microservice
WebApp
InvoiceForm.getInvoiceDetails()
InvoiceForm.submitButtonClick
Validate
ValidateInvoice.validate()
MainInvoice
MainInvoice.createInvoice()
MainInvoice.getInvoice()
MainInvoice.save()
MainInvoice.sendToMinistry()
SDI Webservice
(Ministry of Finance Information System)
MainInvoice.updateStatus()
AmazonGlacier.storePermanently()
Glacier
DBManager
DBManager.read()
DBManager.save()
CUSTOMER
INVOICE
ACCOUNTANT

Figure 4: The Proposed Slicing Options – (Simplified example).

cesses less than 1000 times, and 12 processes between 1000 and 10K times; five processes had been used more than 60B times. This distribution was initially expected for most of the processes. The most frequently used processes were due to the submission of the invoices from each tax accountant.

Based on the previous step, the software architect, together with the project manager, identified three slicing options, taking care to avoid circular dependencies between services, especially in the case of the three classes that suffered from an existing circular dependency. Therefore, we proceeded with the calculation of cohesion and coupling of the different slicing options. The CBM was counted from the outgoing calls reported in the output of DISCO tool.

The first solution was composed of 25 microservices (19 classes duplicated) with an average CBM of 0.16; the second solution was composed of 27 microservices (21 classes duplicated) with an average CBM of 0.16; while the third solution was composed of 21 microservices (14 classes duplicated) with an average CBM of 0.22. Table 4 shows the measures collected for five microservices of each decomposition solution.

The first initial result is the list of classes, methods, and database tables used in the different processes together with their frequency of usage. Moreover, the solution proposed by the software architect had higher coupling and was more complex than the ones identified with the process-mining approach. The analysis of log files also shows that some processes were used much more than expected while one process traverses an unexpected set of classes performing an incorrect process. Moreover, they also discovered three unexpected circular dependencies and the presence of two harmful code smells (Taibi et al., 2017a).

One of the most important decision drivers that lead to the selection of one of the three identified solutions, was the type of decomposition. One of the selected decomposition options proposed to slice the system based on the need of creating more shared libraries. Shared libraries commonly increase the complexity of the migration, and increase the maintenance complexity. The other reason was related to the developer's knowledge and the code ownership. The selected solution allowed to split the systems reducing the need of re-organizing the teams and re-allocating the responsibility of the code. The developers preferred to migrate the code they have written in the past, instead of migrating the code written by other developers.

5 DISCUSSION

In this work we proposed a decomposition process to slice monolithic systems into microservices based on their runtime behavior.

The main benefit of analyzing runtime information is the availability of the data on the usage of each component, together with the dynamic analysis of dependencies. We identified several dead methods and classes that were never used at runtime and we also spotted some cyclic dependencies. The static analysis of dependencies would have spotted the circular dependencies but not all the dead code. Moreover, thanks to the information obtained from the frequency of usage of each method, we also better understood which feature is used more, we prioritized the development and the slicing of the monolithic system differently. Without the information on the frequency of usage of methods, we could have created a microservice that would have done most of the computational tasks.

We are aware about possible threats to validity. We tried to reduce them by applying a common process mining tool (DISCO), that has been developed for several years and has been adopted by several companies and universities. However, the tool could have identified some processes incorrectly. Moreover, we are aware about the complexity related to the data collection, since to adopt our process, com-

Table 4: Decomposition metrics for the decomposition options.

MS	Decomposition Solutions											
	Solution 1				Solution 2				Solution 3			
	CBM	#Links	#Classes	#Dupl. Classes	CBM	#Links	#Classes	#Dupl. Classes	CBM	#Links	#Classes	#Dupl. Classes
MS1	0.08	6	75	2	0.26	7	27	0	0.13	5	39	2
MS2	0.29	9	31	4	0.33	11	33	2	0.26	7	27	0
MS3	0.08	2	25	0	0.06	2	33	1	0.33	10	31	4
MS4	0.16	7	43	2	0.08	4	50	3	0.17	7	41	3
MS5	0.17	5	30	0	0.18	10	56	0	0.14	4	28	0

panies need to instrument their code to collect the log files at the method level. About the generalizability of the results, the validation case study was based on an analysis of the processes of one company. The project manager and the software architect had a limited experience decomposing systems into microservices but the authors of this paper have more than four years of experience in supporting companies in decomposing systems into microservices and closely followed them during the migration.

Companies could benefit from our lessons learned, by applying this process to decompose their monolithic system, but also monitoring the runtime behaviors or existing microservices to continuously understand possible issues. However, despite this approach being very beneficial in our company, the results could have a different impact on other companies. Researchers can benefit from this approach and extend it further. New optimization metrics could be defined, and in theory, it would be possible to propose an automated decomposition approach that would identify the slices by maximizing the metrics identified. Genetic algorithms could be a possible solution for this idea.

6 CONCLUSION

The decomposition of monolithic systems into microservices is a very complex and error-prone task, commonly performed manually by the software architect.

In our work, we demonstrated the usefulness of existing process-mining approaches for decomposing monolithic systems based on business processes identified from the process-mining approach.

Our goal is not to create a tool to support the automated slicing, but to provide an extra support to software architect, to help them in identifying different slicing options reducing the subjectivity.

We first proposed a simple process-mining approach to identify business processes in an existing monolithic solution based on three steps. In the first step, a process-mining tool (DISCO or similar) is used to identify the business processes. In the second step, processes with common execution paths are clustered and a set of microservices is proposed based on business processes with similar behavior, paying attention to not include circular dependencies. In the third step, we propose a set of metrics to evaluate the decomposition quality.

We validated our approach in an industrial case study. The software architect of the SME together with the project manager identified a decomposition solution and asked our consultancy to assess it and to identify other possible decomposition options. This enabled us to compare our process-mining approach with the decomposition solution they proposed.

As a result, we found that our process simplified the identification of alternative decomposition solutions, and provided a set of measures for evaluating the quality of the decomposition. Moreover, our process-mining approach keeps track of the classes and methods traversed by each process, which does not only help to identify business processes but also makes it possible to discover possible issues in the processes, such as unexpected behavior or unexpected circular dependencies.

In case log data is available, or in case it is possible to collect logs, we highly recommend that companies planning to migrate to microservices should use it, considering the very low effort needed to identify alternative solutions with our approach (less than two working days) and the possible benefits that can be achieved.

Future works include the development of a tool to facilitate the identification of the process, the automatic calculation of the metrics, and identification of other measures for evaluating the quality of the decomposition. We are also planning to further empirically validate this approach with other companies and to include dynamic measures for evaluatinfg the quality of the system at runtime (Lenarduzzi et al., 2017b) (Tosi et al., 2012). In the future, we are also planning to adopt mining software repositories techniques to identify the areas that changed simultaneously in the past, to help developers to understand pieces of code connected to each other.

Another possible future work is to include identification of partial migration, i.e., migration of a limited set of processes from a monolithic system. Finally, we are also considering to extend this work by proposing not only different decomposition options but also a set of patterns for connecting microservices based on existing common microservices patterns (Newman, 2015) (Taibi et al., 2018) and anti-patterns (Taibi and Lenarduzzi, 2018)(Taibi et al., 2019).

REFERENCES

Abbott, M. L. and Fisher, M. T. (2015). *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. Addison-Wesley Professional, 2nd edition.

Chen, R., Li, S., and Li, Z. (2017). From monolith to microservices: A dataflow-driven approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 466–475.

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493.

De Alwis, A. A. C., Barros, A., Polyvyanyy, A., and Fidge, C. (2018). Function-splitting heuristics for discovery of microservices in enterprise systems. In Pahl, C., Vukovic, M., Yin, J., and Yu, Q., editors, *Service-Oriented Computing*, pages 37–53, Cham. Springer International Publishing.

Fenton, N. and Bieman, J. (2014). *Software Metrics: A Rigorous and Practical Approach, Third Edition*. 3rd edition.

Fowler, M. and Lewis, J. (2014). Microservices.

Gysel, M., Kölbener, L., Giersche, W., and Zimmermann, O. (2016). Service cutter: A systematic approach to service decomposition. In *European Conference, ESOCC 20162016*, pages 185–200.

Jabangwe, R., Börstler, J., Smite, D., and Wohlin, C. (2015). Empirical evidence on the link between object-oriented measures and external quality attributes: A systematic literature review. *Empirical Softw. Engg.*, 20(3):640–693.

Kecskemeti, G., Marosi, A. C., and Kertesz, A. (2016). The entice approach to decompose monolithic services into microservices. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, pages 591–596.

Kramer, S. and Kaindl, H. (2004). Coupling and cohesion metrics for knowledge-based systems using frames and rules. *ACM Transaction on Software Engineering Methodologies*, 13(3):332–358.

Lenarduzzi, V., Sillitti, A., and Taibi, D. (2017a). Analyzing forty years of software maintenance models. In *39th International Conference on Software Engineering Companion*, ICSE-C '17. IEEE Press.

Lenarduzzi, V., Stan, C., Taibi, D., and Venters, G. (2017b). A dynamical quality model to continuously monitor software maintenance. In *11th European Conference on Information Systems Management (ECISM)*.

Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Newman, S. (2015). *Building Microservices*. O'Reilly Media, Inc., 1st edition.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058.

Richardson, C. (2017). Pattern: Microservice architecture.

Saarimäki, N., Lomio, F., Lenarduzzi, V., and Taibi, D. (2019). Does Migrate a Monolithic System to Microservices Decreases the Technical Debt? *arXiv e-prints*, page arXiv:1902.06282.

Soldani, J., Tamburri, D. A., and Heuvel, W.-J. V. D. (2018). The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215 – 232.

Suonsyrjä, S. (2015). Designing an unobtrusive analytics framework for monitoring java applications. In *International Workshop on Software Measurement (IWSM)*, pages 160–175.

Taibi, D., Diebold, P., and Lampasona, C. (2013). Moonlighting scrum: An agile method for distributed teams with part-time developers working during non-overlapping hours. In *ICSEA - International Conference on Software Engineering and Advances*. IARIA XPS Press.

Taibi, D., Janes, A., and Lenarduzzi, V. (2017a). How developers perceive smells in source code: A replicated study. *Information & Software Technology*, 92:223–235.

Taibi, D. and Lenarduzzi, V. (2018). On the definition of microservice bad smells. *IEEE Software*, 35(3):56–62.

Taibi, D., Lenarduzzi, V., Ahmad, M. O., and Liukkunen, K. (2017b). Comparing communication effort within the scrum, scrum with kanban, xp, and banana development processes. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, EASE'17, pages 258–263.

Taibi, D., Lenarduzzi, V., Janes, A., Liukkunen, K., and Ahmad, M. O. (2017c). Comparing requirements decomposition within the scrum, scrum with kanban, xp, and banana development processes. In *Agile Processes in Software Engineering and Extreme Programming*, pages 68–83.

Taibi, D., Lenarduzzi, V., and Pahl, C. (2017d). Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, 4(5):22–32.

Taibi, D., Lenarduzzi, V., and Pahl, C. (2018). Architectural patterns for microservices: a systematic mapping study. *8th International Conference on Cloud Computing and Services Science (CLOSER2018)*.

Taibi, D., Lenarduzzi, V., and Pahl, C. (2019). Microservices architectural, code and organizational anti-patterns. *Springer (in press)*.

Taibi, D., Lenarduzzi, V., Pahl, C., and Janes, A. (2017e). Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages. In *XP Workshops*, pages 23:1–23:5. ACM.

Tosi, D., Lavazza, L., Morasca, S., and Taibi, D. (2012). On the definition of dynamic software measures. In *ESEM*, pages 39–48. ACM.

Venkatesh, V. (2000). Determinants of perceived ease of use: Integrating control, intrinsic motivation, and emotion into the technology acceptance model. *Information Systems Research*, 11(4):342–365.

Vresk, T. and Cavrak, I. (2016). Architecture of an interoperable iot platform based on microservices. In *MIPRO*, pages 1196–1201. IEEE.

Yourdon, E. and Constantine, L. L. (1979). *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition.

Zimmermann, O. (2017). Microservices tenets. *Computer Science - Research and Development*, 32(3):301–310.