FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Refactoring-assisted migration of monoliths to microservices

Breno Salles

WORKING VERSION

Mestrado em Engenharia de Software

Supervisor: Prof. Jácome Cunha

June 15, 2023

Refactoring-assisted migration of monoliths to microservices

Breno Salles

Mestrado em Engenharia de Software

June 15, 2023

Abstract

In the world of software development, the concept of microservices is gaining popularity. This architectural style has received a lot of attention in both business and academia, and converting a monolithic application into a microservice-based application has become a regular practice. Companies with limited resources struggle with migrating their already existing monolithic applications to microservices and software architects and developers frequently face challenges as a result of a lack of complete awareness of alternative migration methodologies making the refactoring process even harder. The goal of this study is to structurally analyse the state of the art in regards to the migration of monolithic applications to microservices architectural style, mainly how tools are helping architects, engineers, and developers in this migration and how automated they are. To address this challenge, a systematic literature review was conducted, resulting in the identification of one hundred and six relevant publications. These publications were organised and grouped to provide a more comprehensive understanding of the current tools available for microservice migration. Finally, we present a high-level idea for an extensible tool that may aid architects, engineers, and developers during the refactoring process by addressing gaps in understanding of various migration tools and approaches, allowing for easy comparison between multiple options.

Acknowledgments

Amet omnis quasi sit eius repellendus. Quia excepturi illum pariatur ipsa voluptas Quam eligendi vero amet rem aut fugiat optio Animi dicta nesciunt fuga esse magni id Magni labore impedit.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Microservices is an architectural style that evolved from Service Oriented Architecture (SOA). Just like SOA, microservices are an alternative to monolithic architecture. The main contrasts are that, while monolithic applications are software systems with a single, integrated codebase that includes all necessary components, and features [11], microservices tend to be separated, and loosely coupled [16]. Also while monoliths tend to be easier to develop they may scale poorly and are harder to maintain when compared to microservices [15]. Microservices are increasingly being used in the development of modern applications, particularly in the areas of cloud computing and DevOps [18]. Many organizations, including large enterprises and startups, are adopting microservices as a way to build and deploy applications more quickly and efficiently [19]. Microservices are particularly well-suited for distributed, cloud-based environments, where they can take advantage of the flexibility and scalability of the cloud [4]. This type of architecture is already being applied in multiple well-known companies, like Uber, Netflix, eBay [20], and also being followed by the rest of the herd when compared to monolith architecture [22].

Refactoring from monoliths to microservices is a heavily debated topic both in the academic world and the industry. The main takes from this debate are that refactoring is difficult and time-consuming, especially for companies with limited resources who struggle with migrating their already existing monolithic applications to microservices. To help address this, some tools were developed that help with the refactor [28, 30, 31], but in today's world, where the amount of data and information is constantly increasing, it would be ideal to have a centralised location where architects, engineers, and developers can access and utilise all the tools that are currently available as well as those that will be developed in the future. Unfortunately, at the moment, no tool that offers multiple options for migrations with different possibilities exists.

The goal of this study is to structurally analyse the state of the art in regards to the migration of monolithic applications to microservices architectural style, mainly how tools are helping architects, engineers and developers in this migration, and how automated they are. Furthermore, in the following thesis, the purpose is to develop a tool which aims to aggregate existing tools into a single platform, as well as provide the means to extend and incorporate new tools. This tool will offer a convenient and comprehensive way to access and use a variety of tools that help the refactor from monoliths to microservices as well as provide them with a perspective on several migration proposals, allowing for easily comparable and different combinations options.

To achieve this, the guidelines presented by Kitchenham and Charters [13] were followed while performing a systematic literature review. The research protocol was defined at first and then followed to ensure all results could be reproduced.

According to Kitchenham and Charters [13], research questions should be specified as they will direct the entire review methodology. The research questions formulated are as follows:

RQ1. What tools already exist that aid in the migration process of monoliths to microservices?

- *RQ1.1. How do they take the monolith as input?*
- *RQ1.2. How do they produce the microservice as output?*
- *RQ1.3. Are they bound to a specific language?*

RQ2. How can an aggregator of those tools help architects, engineers and developers in their microservice migration?

Chapter 2

Background

To give readers a foundational understanding of microservices architecture and its key features, we will provide a brief overview of microservices and contrast them with traditional monolithic applications. This will allow readers to clearly understand the differences between the two architectures.

2.1 Monoliths

Monolithic applications are software systems that are designed as a single, self-contained unit [9]. In other words, monolithic applications are composed of a single, integrated codebase that includes all of the necessary components and features for the application to run [11]. This means that all of the different parts of the application, such as the user interface, business logic, and database access are all contained within a single codebase and are not modularized or separated into distinct components that are separately deployed and executed.

Monolithic architecture is a traditional approach to software development that has been widely used for many years [9]. It is generally characterized by a strong emphasis on simplicity and ease of development. However, monolithic applications can also be more difficult to maintain and update, as changes to one part of the codebase can have unintended consequences on other parts of the system. This can make it challenging to introduce new features or make changes to the application without significant testing and debugging [11].

Despite these challenges, monolithic applications are still widely used in many contexts due to their simplicity and ease of development. They are particularly well-suited for small to medium-sized applications that do not require a high level of modularity or separation of concerns.

2.2 Microservices

Microservices is an architectural style that structures an application as a collection of loosely coupled services [16]. This means that each microservice is a self-contained unit of functionality, which communicates with other microservices through well-defined interfaces, typically using a lightweight messaging protocol such as HTTP [5].

One key benefit of this approach is that it allows for greater flexibility and scalability [15]. Because each microservice is independent and modular, it can be modified and deployed independently of the other services in the application. This can make it easier to make changes to the system, as it is not necessary to redeploy the entire application every time a change is made. In addition, the modular nature of microservices allows for easier scaling, as individual services can be scaled up or down as needed to meet changing demand [5, 15, 16].

Another advantage of microservices is that they can be developed and maintained by small, autonomous teams [3]. This can be beneficial for organizations with a large codebase or a distributed development team, as it allows for more focused development and faster deployment of changes [14].

However, there are also challenges to consider when adopting a microservices architecture [6]. One challenge is the added complexity of managing a distributed system, as there may be a larger number of moving parts to monitor and troubleshoot [16]. In addition, the communication between microservices can add latency to the system, which may impact the performance of the overall application [6, 17].

Overall, microservices can be an effective way to structure an application, particularly for large, complex systems that require a high degree of flexibility and scalability [16]. However, it is important to carefully evaluate the trade-offs and consider whether the benefits of a microservices architecture are worth the added complexity [6].

2.3 Refactoring

Refactoring is the process of modifying the internal structure of an existing codebase without changing its external behaviour [2]. When migrating from a monolithic architecture to a microservices architecture, it may be necessary to refactor the existing codebase to break it into independent microservices. This can be a complex and time-consuming process, particularly for large, complex systems [15].

There are several factors to consider when refactoring an existing codebase for a microservices architecture [15]. One challenge is ensuring that the code is modular and loosely coupled so it can be developed and deployed independently as a microservice. This may require restructuring the code, introducing new abstractions and interfaces, and potentially even rewriting parts of the code.

Another challenge is preserving the application's existing functionality while making changes to the codebase. It is important to carefully plan and test the refactoring process to ensure that the application continues to work as expected after the changes are made.

Overall, refactoring an existing codebase for a microservices architecture can be a significant undertaking, and it is important to carefully evaluate the resources and time required to complete the process [15].

Chapter 3

Systematic Literature Review

A systematic literature review is a type of review that aims to identify, evaluate, and summarize the results of all studies that address a specific research question or topic [10, 12, 13]. It involves following a specific methodology to identify, analyse, and interpret all relevant evidence related to the research question being addressed. The purpose of a systematic literature review is to provide a comprehensive and up-to-date overview of the current state of knowledge on a specific research question or topic. It is a critical appraisal of the existing research and it can help identify gaps in the literature and inform future research directions [13].

As per Kitchenham and Charters guidelines [13], a systematic literature review (SLR) involves three phases: planning, conducting, and reporting. The planning phase involves establishing the review protocol based on the research questions and the need for the review. The conducting phase involves selecting primary studies and applying the criteria established in the review protocol to analyse them. Finally, the reporting phase involves the creation of the report. These guidelines were loosely followed in the development of this review.

3.1 Research Methodology

To address the research questions posed in Chapter 1, the appropriate research methods were utilised as means to properly investigate the current state of the art. To give guidance, Figure 3.1 shows a diagram of the review iterations that will be explained in the following sections.

3.1.1 Data sources

To access relevant research and information in this field, it is advisable to search several databases that specialise in scientific literature. Table 3.1 presents a list of several such databases, including the ACM Digital Library, Science Direct, IEEE Xplore, Wiley, Springer Link, Engineering Village, and Google Scholar. These databases contain a wealth of knowledge and resources, including journal articles, conference proceedings, technical reports, and more, which can be useful for staying up to date on the latest developments in this field.

Figure 3.1: Review iterations

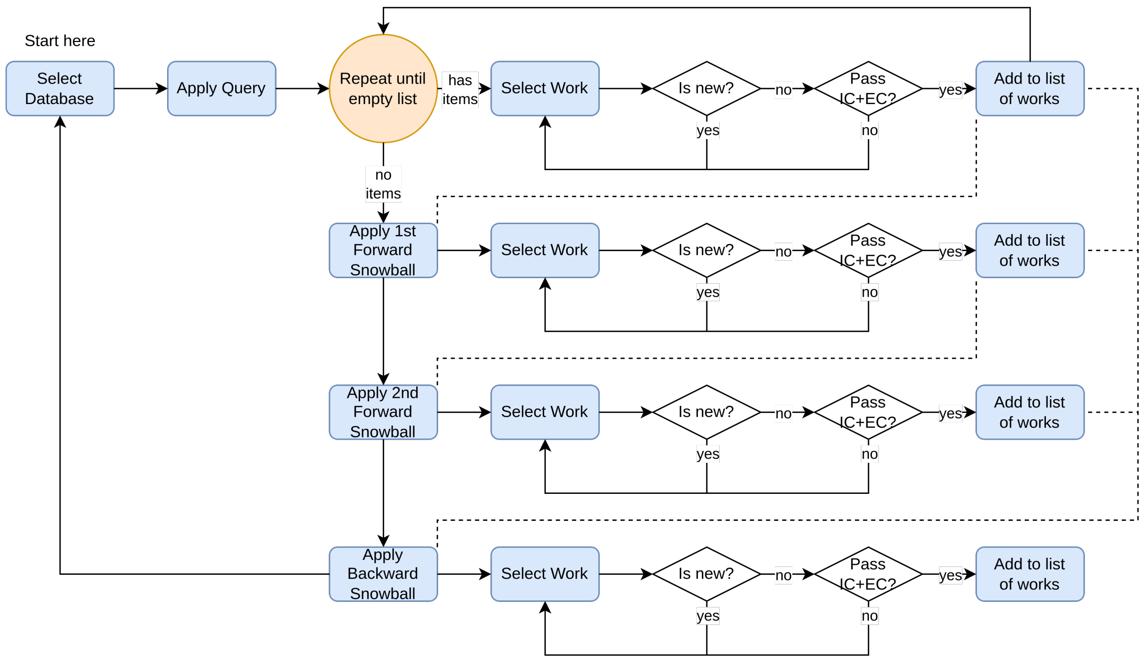


Table 3.1: Databases

ID	Search Engine	Website
DB.1	ACM Digital Library	https://dl.acm.org/
DB.2	IEEE Xplore	https://ieeexplore.ieee.org/
DB.3	Springer Link	https://link.springer.com/
DB.4	Wiley	https://onlinelibrary.wiley.com/
DB.5	Science Direct	https://www.sciencedirect.com/
DB.6	Engineering Village	https://www.engineeringvillage.com/
DB.7	Google Scholar	https://scholar.google.com/

3.1.2 Search strategy

To ensure a thorough and comprehensive search for relevant publications in this field, we will utilise a breadth-first search approach. This method involves starting with a specific query string and selecting relevant publications from a given database. We will then use a technique called snowballing to expand the search and locate additional relevant publications. Snowballing involves searching for citations and publications that are related to the initially selected publications.

There are two types of snowballing that we will employ in this search: forward snowballing and backward snowballing. Forward snowballing involves searching for citations and publications using Google Scholar for the initially selected publications. This process can be repeated multiple times, with each iteration referred to as a level of snowballing. For this search, we will perform two levels of forward snowballing, in which we extract the references of the initially selected publications (level one) and then select the references of those references (level two).

Backward snowballing involves searching for publications that have been cited by the initially selected publications. This technique can also be repeated multiple times, but for this search, we will only perform one level of backward snowballing. This will include all previous publications found during the forward snowballing step.

By utilising both forward and backward snowballing techniques, we aim to cast a wide net and identify as many relevant publications as possible.

After completing the search for relevant publications in a given database using the specified query string, we will move on to the next database. This approach is advantageous because it allows us to efficiently locate relevant publications while minimizing the number of duplicates that are analysed. By searching multiple databases and using snowballing techniques, we can identify a large number of relevant publications and eliminate the need to analyse many of them in subsequent iterations.

3.1.3 Query definition

To identify relevant publications for this research, we will utilise a range of keywords related to the topic of microservices. These keywords will include various phrases and terms used to describe microservices. As for the practices that may help identification of microservices, keywords that help this architectural refactoring should be included, such as “migration", “refactor", “identification". It could also be useful to use “monolith" (and all its possible synonyms) to be the comparison against “microservices", although this can result in some extra publications not related to microservices but instead related to “service-oriented architecture". An expected outcome or conclusion of the publication could be included, “approach" or even “tool". The main keywords that will be used are present in Table 3.2.

Table 3.2: Keywords

Focus	microservices
Refactoring	migration, decomposition, identify, refactor, evolve, discover, transition
Target	monolith
Outcome	approach, tool

First iteration

The first iteration of the review has the main purpose of determining how many tools exist that are able to solve this research question or, at the very least, help partially with it. In order to do this, one could not be limited to tools that are documented in academic databases therefore, in addition to the databases mentioned in Table 3.1, GitHub, GitLab and even DuckDuckGo were searched for, even though they do not represent a scientific search engine.

By using some keywords mentioned in Table 3.2 the following initial trial query was created:

(“microservice" OR “micro-service") AND (“migration" OR “identification") AND (“monolithic" OR “monolith") AND (“tool")

Second iteration

In the second iteration of the review process, the focus is on locating publications that describe alternative approaches for migrating from monolithic to microservices architectures that may not have been implemented in practice. This will allow an increased understanding of the current state of the art in this area, identify any gaps or areas where further research is needed, and determine what can be improved upon. This information will be useful in guiding the development of our tool and abstraction.

As this was the beginning of a new iteration of the review process, the results of the previous “test run" were not considered when extracting literature. Only after this iteration was completed did we incorporate the previously analysed literature and exclude it from further analysis to avoid duplication of effort. This allowed us to focus on identifying new and potentially relevant publications for our purposes.

Relying on the keywords identified in Table 3.2, the following query was created:

(microservice OR micro?service*) AND (migrat* OR identif*) AND (monolith*) AND (migrat* NEAR/2 (process* OR approach*))*

In the event that the publications located through the snowballing approach are no longer adding significant value to the overall understanding of the topic, a new search query should be applied to the remaining databases. In some instances, the query produced more than two thousand results, which would have been impractical to analyse within the given timeframe. Therefore, we modified the query to focus only on the titles and abstracts of the publications. The revised query that should be used is:

(microservice OR "micro-service") AND (migrat* OR decompos* OR identif* OR refactor* OR evolv* OR extract* OR discover* OR transition*)*

3.1.4 Selection Criteria

In order to filter the publications, the title and the abstract will be analysed and should mention at least one of:

IC1. A tool that automates the process of migration of monoliths to microservices.

IC2. Identification of microservices from monolith systems.

IC3. Analysis of tools or approaches for migrating from monoliths to microservices.

In cases of ambiguous abstracts, further inspection of the publication may be done. When this happens, and if relevant publications apply, conclusions should also be taken into account.

As for more pratical approach for exclusion of publications, the criteria will be:

EC1. Publications that are not written in English or Portuguese.

EC2. For Portuguese publications, English must be the language used in the abstract.

EC3. Publication is not accessible.

3.2 Research Results

The query mentioned in the first iteration Section 3.1.3 was then applied to DB.1, DB.2 and DB.3 and after applying the selection criteria mentioned in Section 3.1.4, results were gathered and are presented in Table 3.3.

Table 3.3: Tool Search

Database	**Total number of results**	**Extracted Results**
DB.1	118	3
DB.2	4	0
DB.3	658	3

As for GitHub, GitLab and DuckDuckGo, the query would be essentially typed into their respective search engine and the results gathered as well as the query are presented in Table 3.4.

Through this search process, we can also trace the references used in these publications to determine if the tools described were based on previous work, but only implemented a specific approach. This will help us to understand the context and origins of these tools and how they fit into the broader landscape of research in this area.

As for the second iteration mentioned in Section 3.1.3, we then proceeded to apply the query and search the first database. As shown in Table 3.5, from four hundred and fifty results, only twelve passed the selection criteria defined in Section 3.1.4.

After reviewing the references of the identified papers and applying forward and backward snowballing techniques, we were able to locate additional related publications and expand the scope of our search as demonstrated in Table 3.6. This helped us to increase the number of relevant publications that we were able to consider in the next steps of the process.

In the case of Science Direct, as presented in Table 3.7, two queries were done. The main reason for this is that Science Direct is limited to 7 *OR* conditions, therefore it was necessary to

Table 3.4: Search Engine Tool Search

Search Engine	**Query**	**Total number of results**	**Extracted Results**
GitHub	`https://github.com/search?q=monolith+to+microservice`	745	4
GitLab	`https://gitlab.com/search?search=monolith%20to%20microservice`	0	0
DuckDuckGo	`https://duckduckgo.com/?q=monolith+to+microservices+tool`	Uncountable	2

Table 3.5: DB.1 Results

Total number of results	**Extracted New Results**
450	12

Table 3.6: DB.1 Snowballing Results

1st Forward	**2nd Forward**	**Backward**
23	2	45

split it into two queries where it does not affect the general condition. In the specific case, the *"evolv"* keyword was moved into a separate query. Also, Science Direct automatically accepts truncations without using the *"*"* char. The two queries are:

1. *(microservice OR "micro-service") AND (migrat OR decompos OR identif OR refactor OR extract OR discover OR transition)*

2. *(microservice OR "micro-service") AND (evolv)*

After iterating over the results and reviewing the references of the newly found publications, we did not identify any additional publications that were worth including in the final list. This

Table 3.7: Remaining DB Results

Database	**Total number of results**	**Extracted New Results**
DB.5 - 1st	21	1
DB.5 - 2nd	0	0
DB.4	9	3
DB.6	114	12
DB.3	20	0
DB.2	0	0

marked the end of our general search for relevant publications. We were able to find one hundred and six relevant publications.

3.3 Publications Grouping

Given the large number of publications that were identified as potential candidates for further analysis, it was necessary to further reduce the list to a more manageable size. To accomplish this, we employed a categorization approach in order to better organize and prioritize the publications for later selection. This allowed us to select and analyse the most relevant publications for our purposes. Through this process, we arrived at three main categories that were derived from RQ1 into which we could place each publication. This will be especially relevant when creating the new tool, by enhancing the possibility of integrating various tools that employ different approaches, in order to provide the developer with multiple perspectives, which may facilitate the ability to make comparisons and informed decisions.

- The **approach** used for identifying microservices from monoliths, Table 3.8.
  - *Data flow*.
  - *Dependency analysis*.
  - *Execution log*.
  - *etc*.
- The current **status** of the publication, Table 3.9.
  - It only explains the *method* at a high level.
  - Has implementation details with the *algorithm* on how to identify.
  - Already has a working *tool*.
- The **language** in which that it targets, Table 3.10.

- *Java*.
- *Cpp*.
- *C*.
- *Language Agnostic*.
- *etc*.

The publications that were selected are grouped in Tables 3.8, 3.9 and 3.10. It is important to note that the papers analysed in this study were classified into multiple categories, as opposed to a singular classification. To facilitate a more comprehensive understanding, the classified papers can be viewed on the online spreadsheet [1].

Table 3.8: Approach grouping

Approach	**Amount**
Data flow	8
Control flow	7
Dynamic analysis	11
Semantic analysis	4
Problem frames	1
Model based	10
Static analysis	13
Dependency analysis	15
Multi objective	1
Feature analysis	7
Data analysis	6
REST	4
Graph based	2
Domain analysis	10
Neural analysis	2
Layer	1
Business analysis	3
Strangler pattern	1
Code change history	1
Contributor based	1
Logs analysis	1
Transactional contexts	1
Execution flow	1
Unknown	2

[1] `https://bit.ly/publication-grouping`

Table 3.9: Status grouping

Status	Amount
Method	48
Algorithm	7
Tool	25
Unknown	1

Table 3.10: Language grouping

Language	Amount
Agnostic	59
Java	16
Ruby	1
Python	3
Unknown	4

3.4 Selected Work

Having evaluated most of the literature in regard to tools that help with the migration of monoliths to microservices, we need to select those that are most relevant for the purpose of this thesis. Given our focus on tools and their implementation, we will prioritise works that have already developed a tool and made it available for a free inspection and use. Therefore, if a publication does not provide a link to the tool or instructions for self-hosting or deploying it, it is not worth further consideration. This will help us to focus our efforts on publications that provide practical and useful information about tools and their implementation.

3.5 Publication Analysis

In the following sections, we will provide an analysis of the data collected during the knowledge extraction process from the selected publications. This analysis will allow us to address the primary research question (RQ1) and its sub-questions.

In the following sections, we will provide an analysis of the data collected during the knowledge extraction process from the selected publications. This analysis will allow us to address the primary research question (RQ1) and its sub-questions.

3.5.1 Monolith as an input for the tool

The first aspect to be analysed is the input requirements for the tool. Despite the growing interest in microservice migration using automated tools, the field is still in its infancy, and the existing solutions tend to address specific issues rather than being versatile. As a result, the inputs for these tools are often rigid and not easily adjustable. For example, raw source code and OpenAPI specification were possible ways tools use for identifying microservices from monoliths and will be further discussed.

Source Code

One potential method for providing input to a tool is by utilising source code directly. Our research revealed that eighteen of the contributions analysed use source code as input for their tools with multiple using Spring Boot or other equivalent frameworks to help in understanding the overall code structure. One reason for the use of frameworks is that they provide building blocks for developers, meaning the core functionality of the framework is already in place and developers simply fill in the gaps allowing for the framework to apply inversion of control [8]. Since the behaviour of the framework is kept intact, the tool can then safely analyse the overall code and even apply the refactoring.

For instance, Freitas et al. [30] tool, MicroRefact [2], utilises Java source code to extract structural information by relying on the Abstract Syntax Tree. This information is used to generate a list of candidate microservices. They then leverage Spring Boot decorators, particularly those utilising the Java Persistence API (JPA), to infer the entities of the database and their relationships. This process then results in the output of working Java code for each identified microservice.

OpenAPI

In a microservices architecture, one of the common solutions for communication between different microservices is through HTTP calls. Therefore, it is reasonable to assume that identifying microservices within a monolithic application could be done by examining their REST endpoints since they will be exposed through HTTP protocols. This inspection of REST endpoints can also serve as a guide for decomposing the monolithic application into smaller, independent services. In fact, when the programming language was not a determining factor, OpenAPI was commonly used as the standard for distinguishing microservices from monolithic systems.

The tool proposed by Al-Debagy and Martinek [24] utilises the OpenAPI specification file to identify microservices within a monolithic application. The tool begins by extracting the operation names from the OpenAPI file, which are then input into the Affinity Propagation Algorithm [7]. This algorithm calculates the number of microservices by analyzing the messages exchanged between data points. Afterwards, clustering is performed by utilising the Silhouette coefficient [21] which results in the identification and grouping of similar microservices, helping in the decomposition of the monolithic system.

MsDecomposer [3] [41] uses a similar approach to identify microservices within a monolithic application. The first step is to calculate the similarity of candidate topics and response messages among the APIs. Then, it constructs a graph that represents the similarity between different APIs,

[2] `https://github.com/FranciscoFreitas45/MicroRefact`
[3] `https://github.com/HduDBSI/MsDecomposer`

where the APIs are represented as nodes and the similarity score is the weight. Finally, it applies a graph-based clustering algorithm on the constructed graph, which helps to identify the candidate microservices.

This highlights that OpenAPI is a language-agnostic method for identifying the architecture of software systems.

Other

Besides the input types that have been previously mentioned, there are other types that may not be able to create a new category but are still relevant to the microservices identification process. For example, using a specific system model as input [40] or utilising the history of changes to understand in addition to common artefacts like classes and methods [39].

3.5.2 Microservices as an output for the tool

For microservices identification, it is important to understand how existing tools output their identified microservices in order to cater to the needs of users. The ideal outcome would be a fully functional code ready to be deployed, as it makes the migration process smoother, ensuring that the resulting microservices have all the necessary components and reducing the effort needed for manual migration. From the work that was analysed, tools that output a list of candidates and tools that output source code are more relevant to take into consideration and will be further discussed.

Candidates List

A prevalent approach for identifying microservices in a monolithic system is by producing a candidate list. This approach is used in most of the publications and tools found, and it is a way of providing an organized and structured output of the microservices that can be derived from the monolithic application, in order to facilitate the migration process. The candidate list is commonly used as a guide or checklist for architects, engineers, and developers to assist in the actual partitioning of the application. It typically includes but is not limited to, data entities, interfaces, methods, and other relevant system components that are used as references to guide the migration process.

The tool proposed by Al-Debagy and Martinek [24] uses OpenAPI as input to identify microservices within a monolithic application. One of the limitations of their output is that it only provides a list of candidates, which may not include information about the relationships and interactions between each microservice. This can be an inconvenience when assigning different development teams or groups with the task of applying the migration, as they may not have enough information to understand how the microservices interact and depend on each other, making it harder to assign and split the workload accordingly.

There are other tools available that output a more informative result for microservices identification. Even though they still output candidate lists, where no migration is done yet, these tools provide more detailed information about the relationships and connections between microservices, some of them even including visual feedback such as clusters and call context tree diagrams [31–33, 35]. This additional information can be very beneficial for architects and developers, as it makes it easier for them to understand the connections and dependencies between microservices, and make informed decisions about how to proceed with the migration process.

Source Code

Generating the final output of the migration process as source code that is ready to be deployed would be the ideal outcome for most cases, as it would lessen the efforts needed to migrate, but it is not yet commonly used in current literature. Out of the tools that were analysed, only two of them employ this method. One of them is the work of Freitas et al. [30], and another is Mono2Micro [31–33, 35] that in an update explained in a YouTube video [4] they now output Java code.

3.5.3 Tool target language

The majority of works utilised Java as their primary programming language for input [23, 25, 27, 29, 31–33, 36, 38, 43]. This may be attributed to the language's strict syntax rules, which facilitate the examination of source code during the inspection process. Additionally, some of them utilised the Spring Boot framework in conjunction with Java [28, 30, 37–39, 42], which further enforces structure through the utilisation of decorators. In contrast, those who employed programming languages other than Java, such as Python, utilised corresponding frameworks like Django, to compensate for the language's more lenient syntax constraints [26, 34].

[4] https://youtu.be/NXno1fUoC8U

Chapter 4

Tool Design

During our analysis described in Section 3.5, it was impossible to locate any relevant works that could address the second research question (RQ2). As a result, it will be necessary to tackle this question and strive to answer it.

As of the time of writing this paper, the design of the proposed tool is not yet finalised. We have only developed initial ideas about how we will approach the problem and have created a sketch of the proposed tool architecture, which is shown in Figure 4.1.

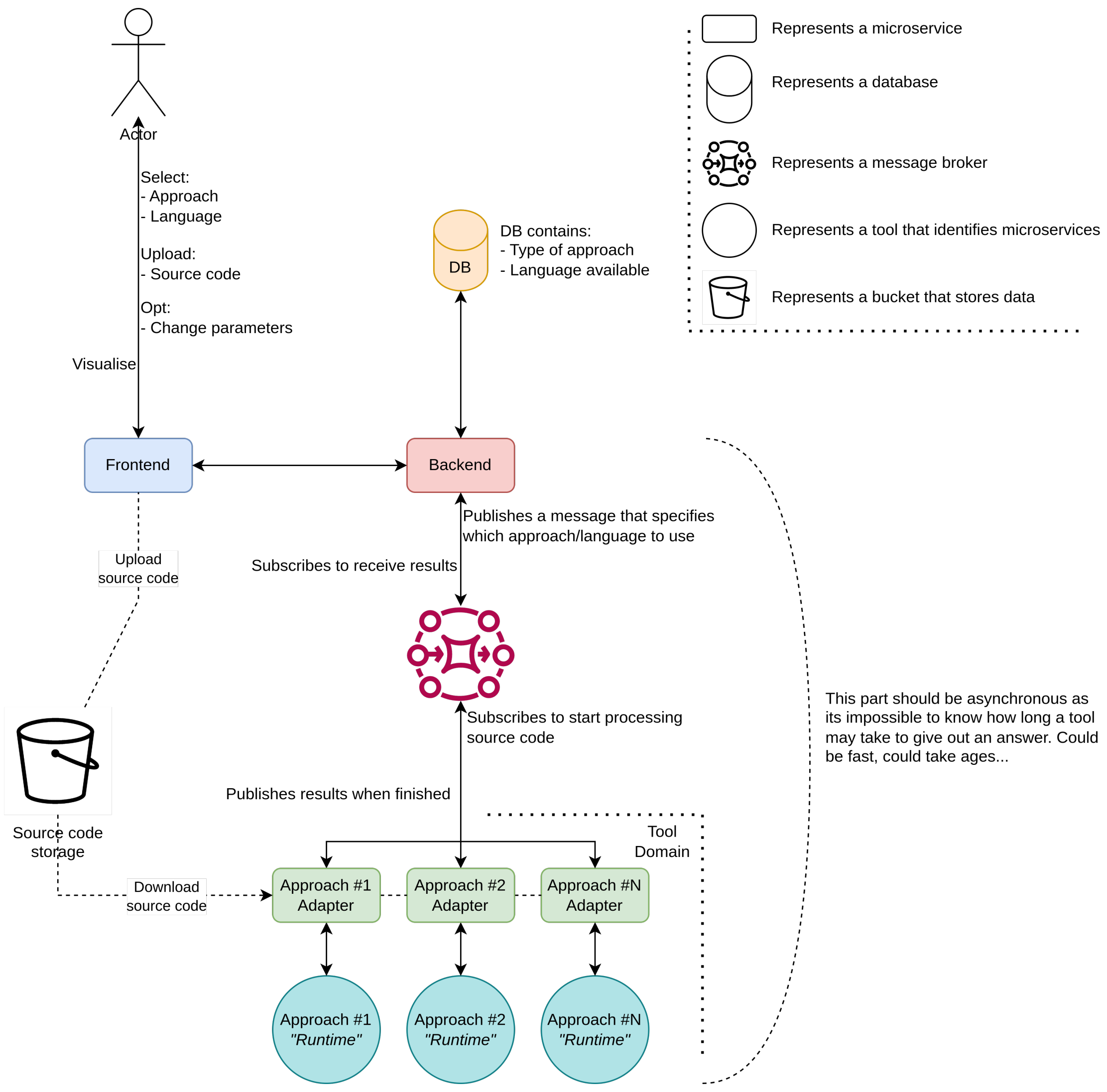
The proposed tool architecture consists of three distinct components. The frontend is responsible for the interface between the tool logic and the user. The tool domain contains adapters for each individual tool and the tool runtime, which receives the monolithic input and generates the candidate microservices. The backend serves as a bridge between the frontend, the database of available tools, and the tool domain.

The main role of the backend component in the proposed tool architecture is to initiate jobs and notify the frontend when a given job has been completed. A job consists of the process of identifying microservices from a given monolithic input, which is then completed when the output containing the identified microservices is produced. To avoid potential bottlenecks when multiple jobs are run concurrently, the backend does not perform these tasks directly but rather delegates them to the individual tools, therefore acting like a bridge.

In the tool domain, the purpose of the adapter is to provide a consistent interface for interacting with the tool runtime, regardless of the specific input and output formats that it uses. This is important because different tools may accept different inputs and produce different outputs, such as JSON or raw code. By using an adapter to translate between these formats, it becomes easier to process the inputs and outputs of the tool and integrate them with the overall tool logic. The tool runtime receives the monolithic input and generates the candidate microservices, while the adapter serves as a "middleman" between the tool runtime and the other components of the architecture.

One of the first decisions that we have considered is which target platform the tool should support. There are three major platforms that are relevant for this purpose: macOS, Windows, and Linux. Determining which of these platforms to focus on would require extensive surveying to ensure that the tool will be used by the intended audience. While market share data suggests

Figure 4.1: Candidate tool architecture


Represents a microservice
Represents a database
Represents a message broker
Represents a tool that identifies microservices
Represents a bucket that stores data
Actor
Select:
- Approach
- Language
Upload:
- Source code
Opt:
- Change parameters
Visualise
DB
DB contains:
- Type of approach
- Language available
Frontend
Backend
Upload source code
Publishes a message that specifies which approach/language to use
Subscribes to receive results
Subscribes to start processing source code
Publishes results when finished
This part should be asynchronous as its impossible to know how long a tool may take to give out an answer. Could be fast, could take ages...
Source code storage
Tool Domain
Download source code
Approach #1 Adapter
Approach #2 Adapter
Approach #N Adapter
Approach #1 "Runtime"
Approach #2 "Runtime"
Approach #N "Runtime"

that Windows is the most widely used desktop operating system, followed by macOS and Linux [1], this may not necessarily reflect the operating systems used by the architects, engineers, and developers who are responsible for migration. Given the limited time frame, it is infeasible to conduct a thorough survey to determine the preferred operating system of this target audience. As a result, we have decided to adopt a more cross-platform solution. After evaluating the options, we have determined that the browser is the most suitable platform for this purpose.

Ideally, each component of the proposed tool architecture would be deployed as a microservice in a Docker container to facilitate scalability and cross-platform deployment. This is especially important for the tool runtime component, as it is uncertain at this point whether the tools will take advantage of multithreading. By using Docker, we can create a new container for each job, which would allow us to handle multiple jobs concurrently and avoid the need for users to wait for their jobs to complete. Of course, it is possible that certain tool limitations may prevent us from implementing this approach, but it is our goal to use microservices and Docker to the greatest extent possible to ensure the flexibility and scalability of the tool.

Given that the tasks performed by the tool runtime component are asynchronous, the communication between the tool runtime and the backend cannot be based on synchronous HTTP request-response cycles. Instead, we will use a message queue to connect the two components, with the use of message brokers. The backend will send a message to initiate a job, which will be picked up by the appropriate adapter and executed by the tool runtime. Once the job is completed, the tool runtime will send another message to the backend to indicate that it is finished. This approach allows us to process multiple jobs concurrently and avoid potential bottlenecks in the communication between the tool runtime and the backend.

4.1 Requirements

Software requirements, also known as system specifications, refer to the explicit delineations of the functionalities a given system should offer, the scope of services it should provide, and the operational constraints it must adhere to. There are two types of requirements: Functional and Non-Functional, where the first focuses on what a system is supposed to do and the latter on how a system is supposed to be. We used the MoSCoW method to prioritise each requirement, and it is an acronym from the first letter of each prioritisation category: Must have, Should have, Could have, Will not have.

We delimited eighteen functional and eight non-functional requirements as an initial Minimum Value Product (MVP), presented in Table 4.1 and Table 4.2, respectively. It is essential to highlight that no requirement fits the *"Will not have"* category because requirement elicitation was not conducted with a user but rather a brainstorming between the entities responsible for this work.

Table 4.1: Functional Requirements

Requirement	**Priority**
Web app allows for user selection of an approach.	Must have
Web app allows for user selection of a language.	Must have
Web app allows for user parameter tuning.	Must have
Web app allows for an user visualisation of the identified microservices.	Must have
Web app allows for user code upload (either source code or specification).	Must have
Web app allows comparrison between multiple identifications.	Must have
Web app allows for user session.	Should have
Web app allows for download of output.	Could have
Web app allows for upload of previous outpus.	Could have
Backend does not halt when an identification is running.	Must have
Backend starts the identification as soon as the input is uploaded.	Must have
Backend must serve the current existing approaches/languages/parameters to the user.	Must have
Backend must signal the user when the identification is completed.	Must have
Backend must signal the tool domain to start processing a code.	Must have
Tool domain has at least one approach.	Must have
Tool domain with each approach running separatly.	Should have
Tool domain with extensible adapter for other approaches.	Must have
Tool domain signals the backend when it finishes processing.	Must have

Table 4.2: Non-Functional Requirements

Requirement	**Priority**
Web app must be deployed individually.	Must have
Web app must be deployed using Docker.	Must have
Web app could be split into multiple microservices.	Could have
Backend must be deployed individually.	Must have
Backend must be deployed using Docker.	Must have
Backend could be split into multiple microservices.	Could have
Tool domain should be deployed individually.	Should have
Each tool implementation deployed individually.	Could have

Chapter 5

Solution Development

The system's backend is implemented using Node.js, an open-source JavaScript runtime environment. Despite JavaScript's inherent limitations, such as its dynamic nature, employing Node.js allowed us to rapidly prototype a solution, a crucial requirement in this work. To mitigate the drawbacks associated with JavaScript, TypeScript was chosen as the primary programming language, benefiting from its static typing and enhanced tooling capabilities.

5.1 Technologies

Node.js

Node.js is an open-source runtime environment that enables the execution of JavaScript code outside of the web browser. Initially developed in 2009 by Ryan Dahl as a response to the criticisms he had expressed towards the widely used Apache HTTP Server.

In contrast to traditional execution models, where code is executed sequentially and relies on thread mechanisms to prevent blocking, Node.js adopts a distinct approach. It capitalises on JavaScript's event loop paradigm to manage asynchronous I/O operations. This event-driven architecture enables Node.js to handle concurrent requests efficiently, maximising performance.

Moreover, the Node.js ecosystem offers modules via NPM that address various common challenges encountered in software development. These modules help with functionalities such as file and network access, manipulation of binary data, cryptography, and other general-purpose tasks.

Node.js has become widely adopted as a versatile tooling platform for developing a wide range of applications, even when those applications themselves do not run on Node.js. Notably, frameworks such as React, Vue, and Angular utilise Node.js for tooling purposes, specifically for compiling their JavaScript framework code into browser-compatible JavaScript.

In the context of this work, Node.js is used for deploying both the backend and the underlying tool system, while is also used as a build tool for the frontend.

TypeScript

TypeScript, an open-source language developed and maintained by Microsoft, is considered a superset of JavaScript that introduces static typing to the language. The motivation behind TypeScript originates from the shortcomings experienced when developing large-scale applications with JavaScript. While JavaScript offers speed and ease of use, as projects grow in size and complexity, maintaining and updating it becomes increasingly challenging.

TypeScript brings some benefits, namely:

- Compilation: TypeScript code must be compiled to JavaScript before execution. This compilation step allows developers to identify errors during the process. If there is an invalid code, the compilation will fail, providing an opportunity to catch and rectify errors before runtime.
- Strong and Static Typing: TypeScript introduces a type system that builds upon JavaScript. While JavaScript permits variables to have any type, TypeScript enforces static typing, requiring developers to explicitly declare the type of variables, functions, methods, and many more. This approach promotes code clarity and helps prevent type-related errors by ensuring that variables are assigned appropriate types and used correctly throughout the codebase.

For this work, TypeScript is used transversally across all implementations. Anytime JavaScript is required, TypeScript is used instead.

NestJS

Kamil Myśliwiec created NestJS to facilitate the development of efficient and scalable backend applications using Node.js. This framework supports JavaScript and TypeScript languages, allowing developers to choose the language that best suits their project requirements. One of the distinguishing features of NestJS is its ability to use various programming paradigms, namely Functional Programming (FP), Object-Oriented Programming (OOP), and Functional Reactive Programming (FRP). By incorporating elements from these paradigms, NestJS provides developers with a toolkit to build applications using a combination of functional and object-oriented concepts that allow them to leverage the strengths of each approach, fostering code modularity, maintainability, and reusability. NestJS is an opinionated framework that draws inspiration from Angular and is similar to the Spring framework for Java in the Node.js ecosystem. Like Spring, NestJS offers comprehensive documentation outlining solutions to common backend challenges. It accomplishes this by providing adapters and integrations with popular existing solutions. With the speed, ease of use and integrations it provides, NestJS is used for both the backend and the underlying tool implementation.

Redis

Redis is an in-memory multi-model database famous for its sub-millisecond latency. It was created in 2009 by Salvatore Sanfilippo based on the idea that a cache can also be a durable data store. Around this time, apps like Twitter were growing exponentially and needed a way to deliver data to their end users faster than a relation database could handle. Redis, which means Remote Dictionary Server, was adopted by some of the most trafficked sites in the world, because it changed the database game by creating a system in which data is always modified or read from the main computer memory as opposed to the much slower disk, but at the same time, it stores its data on the disk so it can be reconstructed as needed. Every data point in the database is a key followed by a value that can be any of its many data structures, like lists, sets, streams, json, an others. It can be used as a distributed key-value store, cache, and message broker. This latter use case is exactly how Redis is used in the context of this application, by serving as the communication mechanism between the backend and the underlying tools. In the case of the implementation of the underlying tool, Redis is used as an internal event queue.

Astro, React and Solid

One of the most difficult decisions of a frontend developer is choosing between the UI framework, because once you jump into a ship, its really hard to get out. Astro addresses this by being framework agnostic, it allows the creation of componenets in popular frameworks, like React, Svelte, Vue, Solid and others, meaning one is not bound to technologies of each framework, bringing components from each ecossystem to speed up development.

Docker

Docker is a tool that can package software into containers that run reliably in any environment. One way of packaging an application is with a Virtual Machine (VM), where the hardware is simulated then installed with the required Operating System (OS) and dependencies. This enables the hability of running multiple applications on the same infrastructure, however, because each VM is running its own OS, they tend to be bulky and slow. A Docker container is conceptually similar to a VM, but instead of virtualising hardware, containers only virtualise the OS, which results in faster and more efficient applications. This difference is presented in Figure 5.1.

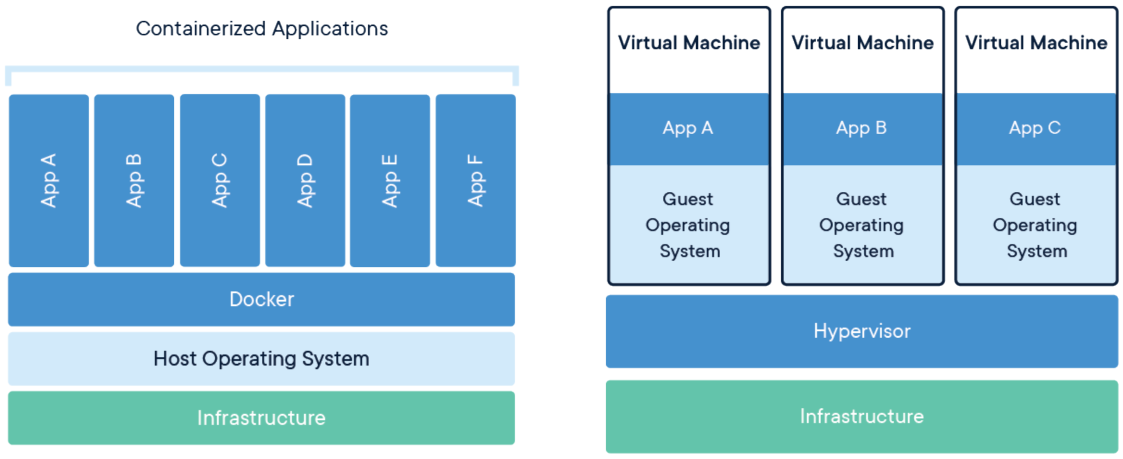
5.2 Database

For any kind of structure data, Strucured Query Language (SQL) is normally the best way of representing it, and we can immediatly understand that there are five core entities in our system:

1. Tool - which represents the underlying tool used in the decomposition.
2. Language - which represents which languages can be targeted for decomposition.
3. Result - which represents if a given job of decomposition was successful or not.
4. Decomposition - which represents the decomposition itself and all data required to represent it.

Figure 5.1: Container and Virtual Machines

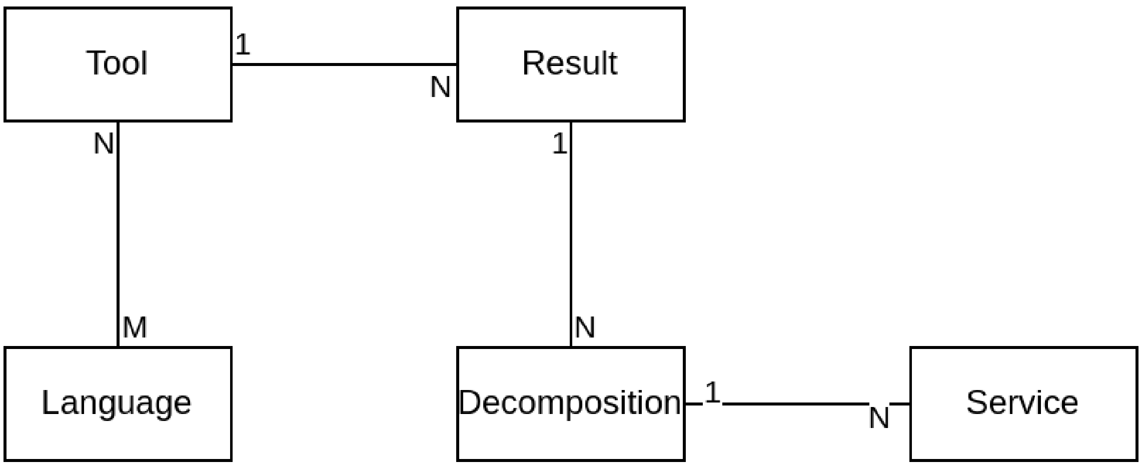
Source: https://www.docker.com/resources/what-container/



5. Service - which represents each microservice.

The relationships between them is represented in Figure 5.2.

Figure 5.2: Database Model



5.3 Backend

As previously mentioned in Section 5.1, the entirity of the backend was built relying on Node.js runtime, the TypeScript language and the NestJS framework. Following the specification presented in Chapter 4, each part of the infrastructure, that is, the backend itself and the tool adapter is a monolith that communicate between them via message broker using Redis.

5.3.1 API Design

The backend implements the REST architecture style and uses JSON as its data transport. The endpoints available are depicted in Table 5.1 and achieves the level two of Richard Maturity Model https://martinfowler.com/articles/richardsonMaturityModel.html

Table 5.1: REST Endpoints

Endpoint/Method	**GET**	**POST**
/users	N/A	Creates a user
/tools	Retrieves all tools	N/A
/results	Retrieves all results	Creates a result
/results/:id	Retrieves a result that matches ":id"	N/A
/decompositions	Retrieves all decompositions	N/A
/decompositions/:id	Retrieves a decomposition that matches ":id"	N/A
/decompositions/:id/export	Exports a decomposition that matches ":id"	N/A

5.3.2 Authentication and Authorization

The authentication and authorization mechanism serve as a way of enabling the user to come back with its user identification and retrieve previous decompositions, while being also separating its own findings and knowledge from others, providing a better user experience when coming back to the application. Authentication per se does not exist. A user can be created by simply making a *POST* request to the */users* endpoint in order to retrieve a new user and no extra fields are required. As for authorization, the paths */results* and subpaths as well as *decompositions* and subpaths require a user id to be accessible, making sure users content is not mixed together. To provide a better user experience, a client of this API, for example the frontend of this solution as described in Section 5.4, will need to save the user id locally and provide a seamless integration upon the user coming back to the application. The implementation details of the frontend are detailed in Section 5.4.4.

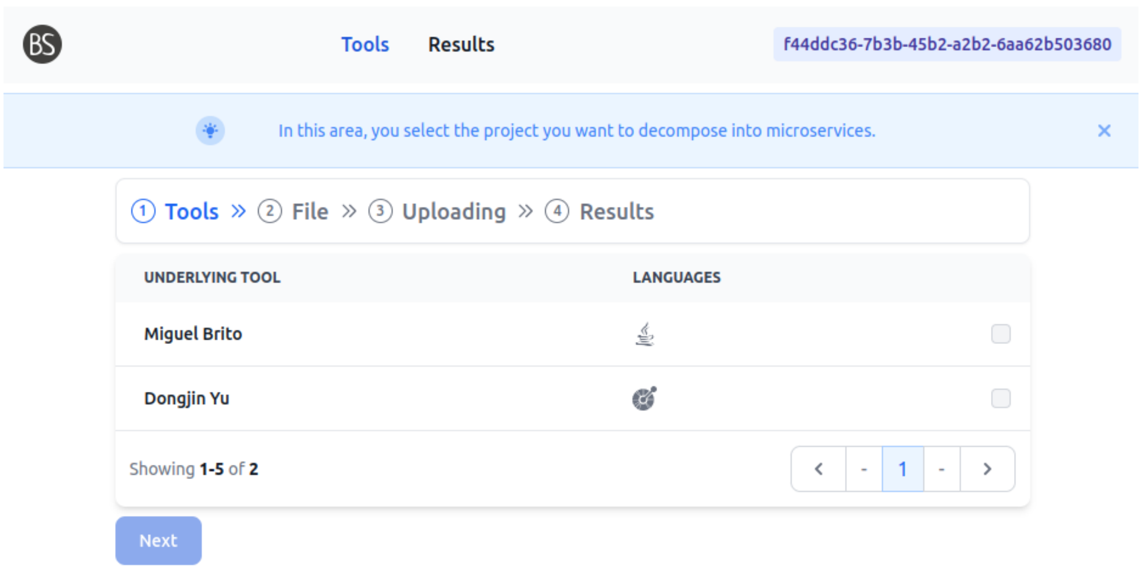
5.4 Frontend

The application frontend uses Astro as the base for the implementation, with the underlying UI frameworks being mainly React and Solid. This application has three main pages: the tool selection page - which is also the homepage, the results page and the comparison page.

5.4.1 Tool Selection

On the tool selection page, the user is required to select the underlying tool for decomposing. At the moment, the only implemented tool is by Miguel Brito.

Figure 5.3: Tool Selection

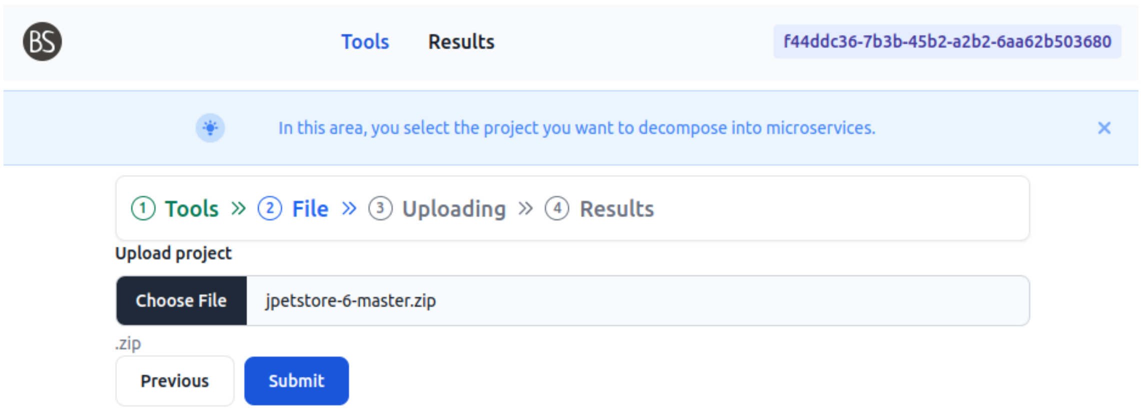

BS
Tools
Results
f44ddc36-7b3b-45b2-a2b2-6aa62b503680
1 Tools » 2 File » 3 Uploading » 4 Results
Next

In this area, you select the project you want to decompose into microservices.

UNDERLYING TOOL	LANGUAGES	
Miguel Brito		☐
Dongjin Yu		☐

Showing **1-5** of **2**

Figure 5.4: Project Upload


BS
Tools
Results
f44ddc36-7b3b-45b2-a2b2-6aa62b503680
1 Tools » 2 File » 3 Uploading » 4 Results
Choose File
Previous
Submit

In this area, you select the project you want to decompose into microservices.

Upload project: jpetstore-6-master.zip

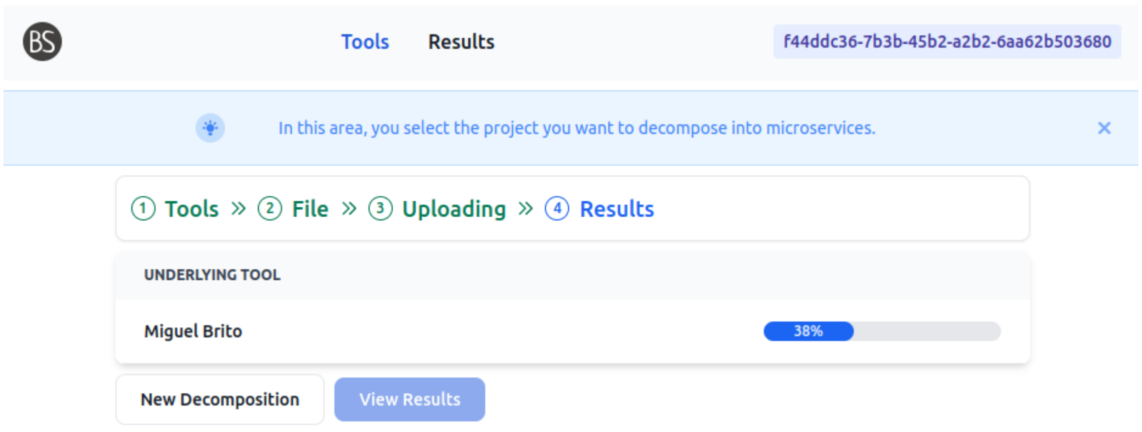
.zip

Figures 5.3 to 5.5 illustrate the process of selecting a tool, uploading a file, and waiting for a decomposition.

5.4.2 Results

The results page contains all the current user's results from previous decompositions. On this page, the user can select up to five decompositions it wants to use for comparison, and it can mix and match previous runs of the tool.

Figure 5.5: Awaiting Decomposition

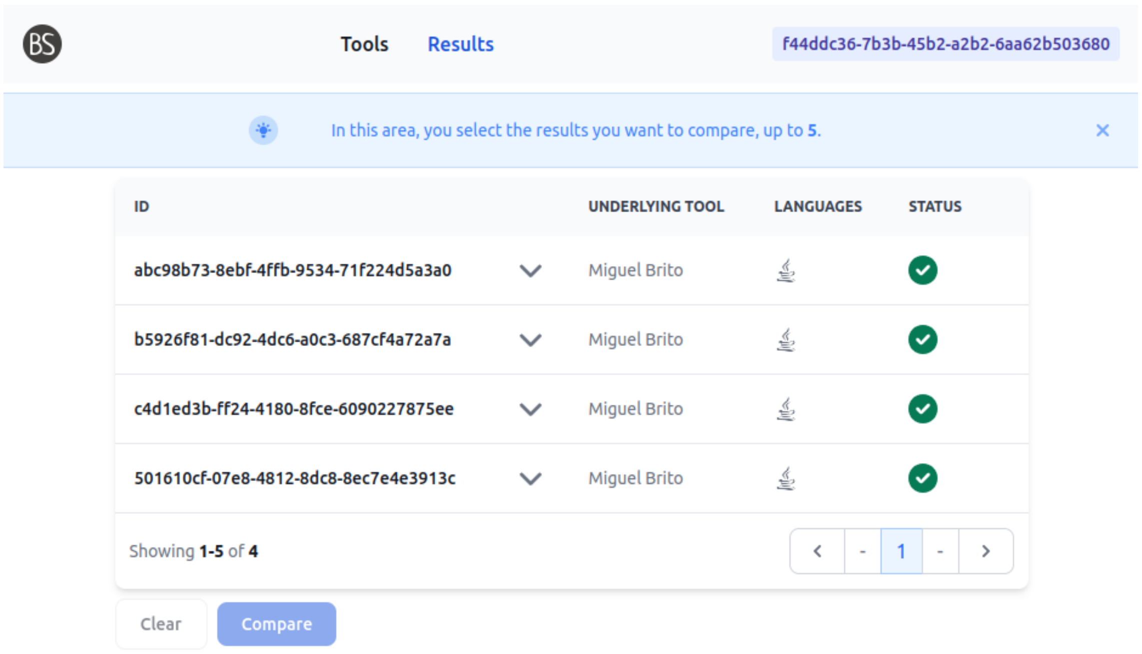


In this area, you select the project you want to decompose into microservices.

UNDERLYING TOOL	
Miguel Brito	38%

BS
Tools
Results
f44ddc36-7b3b-45b2-a2b2-6aa62b503680
① Tools » ② File » ③ Uploading » ④ Results
New Decomposition
View Results

Figure 5.6: All Results



In this area, you select the results you want to compare, up to **5**.

ID	UNDERLYING TOOL	LANGUAGES	STATUS
abc98b73-8ebf-4ffb-9534-71f224d5a3a0	Miguel Brito		✓
b5926f81-dc92-4dc6-a0c3-687cf4a72a7a	Miguel Brito		✓
c4d1ed3b-ff24-4180-8fce-6090227875ee	Miguel Brito		✓
501610cf-07e8-4812-8dc8-8ec7e4e3913c	Miguel Brito		✓

Showing **1-5** of **4**

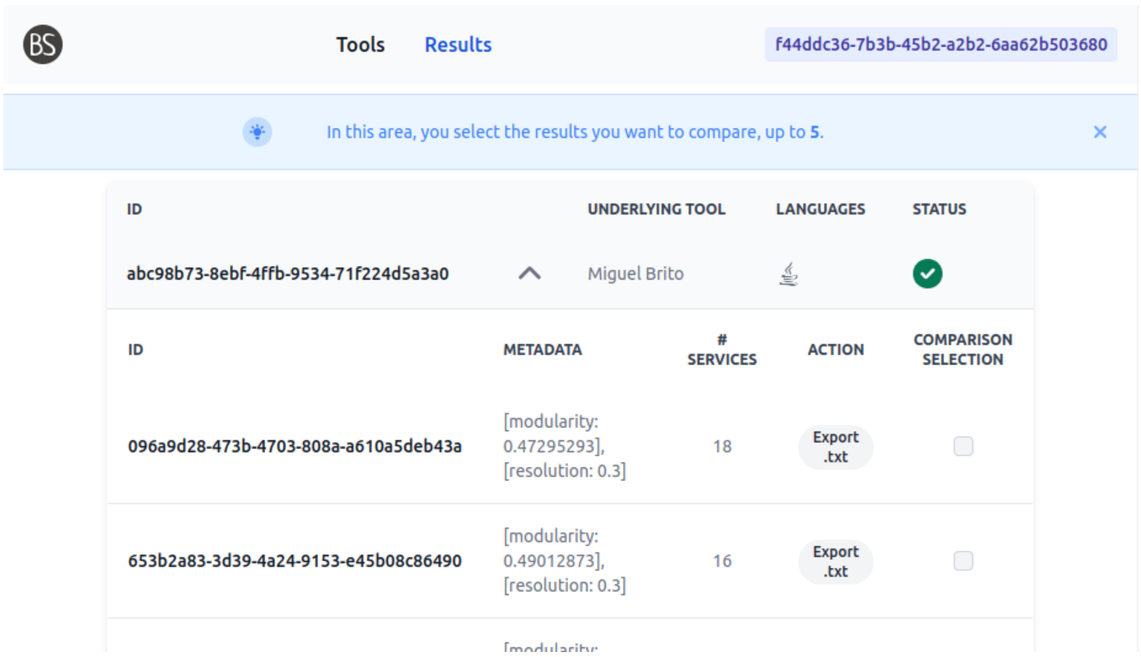
The results page can be observed in Figures 5.6 and 5.7, as depicted in the visual representations provided.

5.4.3 Comparison

The core of the solution is on the comparison page, where the users:

1. Are able to toggle a decomposition to be visible or not.
2. Can visualise all the microservices generated by each decomposition.

Figure 5.7: Expanded Result



f44ddc36-7b3b-45b2-a2b2-6aa62b503680

In this area, you select the results you want to compare, up to **5**.

ID	UNDERLYING TOOL	LANGUAGES	STATUS
abc98b73-8ebf-4ffb-9534-71f224d5a3a0	Miguel Brito		✓

ID	METADATA	# SERVICES	ACTION	COMPARISON SELECTION
096a9d28-473b-4703-808a-a610a5deb43a	[modularity: 0.47295293], [resolution: 0.3]	18	Export .txt	☐
653b2a83-3d39-4a24-9153-e45b08c86490	[modularity: 0.49012873], [resolution: 0.3]	16	Export .txt	☐

3. Can focus on a microservice and check the modules that make each service.

4. Are able to look at which modules are in each microservice across different decompositions.

During the evaluation of how to build the the comparison view, multiple tools that present visualisation data were considered. We end up focusing on four that caught our eye, Graphviz [1], D3.js [2], Gephi [3] and Chart.js [4]. After deeply analysing the pros and cons of each, we decided to create a matrix to better understand each in four categories, *Customizability*, *Ease of use*, *Charts available* and *Real-time interactivity*.

Taking a look at what each tool can offer as presented in Table 5.2 and taking into account that the majority of the industry leans towards node graphs for presenting microservices TODO: https://doi.org/10.1109/SOSE55356.2022.00011 , D3.js was concluded to be the best and most powerful visualisation tool for our use case.

Although having a powerful tool is important, knowing how to present the information is just as important, meaning the shapes of the visualisation tool are relevant. As mentioned by Daniel Moody (TODO: citation), one of the best ways of expressing variation between entities is by using different shapes, different colours, and relationships between entities using various strokes and line dashes that carry weight while the user is looking at the visualisation.

Table 5.3 shows how the visualisation tool expressed each different entity.

[1] https://graphviz.org/
[2] https://d3js.org/
[3] https://gephi.org/
[4] https://chartjs.org/

Table 5.2: Visualisation Tool Comparison

Visualisation Tool	**Customizability**	**Ease of use**	**Charts available**	**Real-time interactivity**
Graphviz	Limited customization options, suitable for creating static diagrams and graphs.	Easy-to-use syntax, accessible to non-experts.	Suitable for creating static diagrams and graphs, with limited support for charts.	Limited interactivity options.
D3.js	Highly customizable, suitable for creating custom and interactive visualizations.	Steep learning curve, requires a good understanding of JavaScript and web technologies.	Suitable for creating a wide range of charts, including bar charts, line charts, scatterplots, and more.	Provides advanced interactivity options, such as brushing and zooming, making it suitable for real-time visualizations.
Gephi	Customizable with a range of features, but may have limited design options.	User-friendly interface, easy to learn for beginners.	Suitable for creating network graphs and visualizations.	Allows users to interact with graphs and manipulate layouts in real-time.
Chart.js	Customizable with various options for color schemes, font styles, and animation effects.	Simple and easy-to-use API, minimal setup required.	Supports common chart types such as line charts, bar charts, pie charts, and more.	Provides basic interactivity features such as hover effects and tooltips.

Table 5.3: Visual Expressiveness

Entity	**Shape**	**Size**	**Colour**
Service	Circle	Variable according amount of modules	Based selected decomposition *
Module	Square	Static	Mix between selected decompositions
Relationships	Line	Static	Static

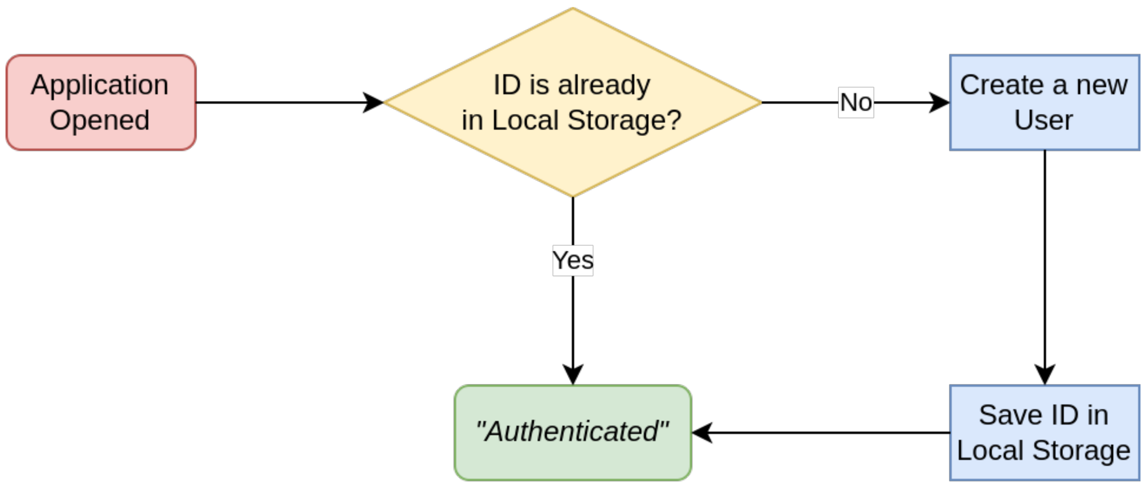
** Each decomposition colour is random.*

5.4.4 Authentication and Authorization

To provide a better user experience to the authentication and authorization mechanism explained in Section 5.3.2, we make use of local storage to store the user id. The flowchart that represents this logic exists in Figure 5.8.

The user can at any point manually remove its own user id, with the risk of losing access to all its previous decompositions.

Figure 5.8: User ID storage Flowchart



5.5 Deployment

5.6 Implemented requirements

From the comprehensive enumeration of requirements delineated in Tables 4.1 and 4.2, it should be noted that their implementation still needs to be fully realised. This limitation in the embodiment of all stipulated requirements can primarily be attributed to the existing time constraints.

Table 5.4: Not Implemented Requirements

Requirement	**Priority**
Web app allows for download of output.	Could have *
Web app allows for upload of previous outpus.	Could have

** It is possible to export the decomposition, but the exportation content would need to be revised to work with the upload requirement.*

Table 5.4 contains requirements that primarily pertain to enhancing the quality of life for users, rather than directly impacting the core workflow of the application. These requirements were identified as non-essential to the fundamental operational processes of the application.

Chapter 6

Empirical Validation

An empirical validation is normally used to validate a developed application, thus we prepared an empirical study to validate if our application helps users perform the decomposition of monoliths into microservices better or not.

Section 6.1 details our study's design, and Section 6.2 explains how it was executed. Section 6.3 is focused on data analysis, followed by an interpretation in Section 6.4.

6.1 Design

As mentioned in Chapter 4, we developed an application to answer the RQ2 explained in Chapter 1. This type of tool can only be validated by how it helps architects, engineers and developers in their migration process, not by how good the microservices decomposition is, as this later validation should be the responsibility of the underlying tools.

With this study, we ought to evaluate the effectiveness and efficiency of users using our application when performing a decomposition task. After that task is completed, the users will fill in a questionaire based on the System Usability Scale (SUS) and NASA's Task Load Index (NASA-TLX). (TODO: citation)

The target audience of this study are the general population of software developers, engineers, architects from any level.

In order to perform this study we have made a questionaire which can be found in Appendix A.

6.1.1 Subjects and Objects

Subjects are who the study is targeted for and objects are what the study will be. In the case of subjects, we have a broad target demographic, since not only we want already senior level developers, engineers and architects to evaluate the visability of using the application in their migrations, but also more junior level users to have an easier entrypoint for monolith decompositions, therefore our study aims to target that demographic population. In the end, the study was able to get some pull both ends of the spectrum. As for objects, the subjects are required to complete a monolith decomposition task. The tasks are all the same, what differs is the monolith project each task is targetting. There are three projects in mind, one of them is used as tutorial, while the other two are reserved for the users themselves. The projects vary in the amount of classes for a heterogeneous study.

6.1.2 Instrumentation

Like it was mentioned in Section 6.1.1, there are three projects will be used in order to perform this study. The tutorial uses a project designated YarkAdminMS [1] with sixty nine classes. Task number one uses project a Warehouse System [2] seventy nine classes, whereas task number two is based on a Petclinic System [3] with a total of forty classes. All projects are based on Java since the application, at the moment, only has Java as the underlying tool for decomposition. TODO: Citar miguel brito sobre tamanho dos projetos. Some projects were forked into this author Github account, so that they are permanently accessible.

6.1.3 Pre Study

The purpose of an empirical study is for the study to reflect what the application is doing. To do that, we must be sure that the study and the tasks run as smoothly as possible, that is, that the users responses reflect exactly what the application is and not the problems our questionaire or task have. In order to deal with that, we employed a plan of running a mock study iteratively with live support, while taking notes of the difficulty the person felt during it. Afterwards, with tweaked the study to fix the problems felt during it and ran it again with a different person, so that this new person is not influenced by already knowing what to expect. People that were subject to this mock run are automatically discarded from the final study, for the same reasons.

6.1.4 Data Collection

The study is divided roughly into eigth parts. The questionaire presented in appendix TODO contains the following sections:

1. Small video [4] describing what microservices are.

2. Background questionaire about age, years of industry experience, etc. TODO apendix

3. A tutorial about the application in video [5] format.

4. The task section where users are required to take note of start and end time. TODO: appendix

[1] `https://github.com/Guergeiro/Yark-AdminMS`
[2] `https://github.com/Guergeiro/Warehouse-system`
[3] `https://github.com/spring-projects/spring-petclinic`
[4] `https://youtu.be/lTAcCNbJ7KE`
[5] `https://youtu.be/WQAAU9vQddA`

5. A section where users should paste what they think is the best decomposition for the given task, including a video [6] on how to fill it.
6. System Usability Scale. TODO: apendix
7. NASA Task Load Index. TODO: apendix
8. An optional section where general feedback is appriciated. TODO: apendix

6.2 Execution

The study was distributed to work colleagues, college colleagues and open-source communities the author belongs to. Because of the study optimisation described in Section 6.1.3, it was not necessary to have a syncronous participation from the respondents, which made it easier to invite them to participate, although live support was offered if ever an issue occured. To start, the participants had to validate their knowledge of microservices and, to help with that, a small video was added in order to help those who had not prior knowledge of microservices. Afterwards, they would answer a small questionnaire regarding their background, where they would state their current age, their gender, their experience in the industry and academia, their instruction level as well as their prowness with certain programming languages. The tutorial follows where an example task is used to demonstrate what it is expected for the task and then, the participants replicate the steps of the tutorial using the task assigned to them. Finally, the users state which microservice they understand to be the best one as well as felling in the System Usability Scale and Task Load Index. There is also an optional section where participants are invited to give some extra feedback for the tool.

6.3 Analysis

In order to perform a quantitative analysis of the study, we collected the answers of 12 different people.

6.3.1 Subjects

In order to perform a quantitative analysis of the study, we collected the answers of 12 different people.

6.3.2 System Usability Scale Scores

In order to perform a quantitative analysis of the study, we collected the answers of 12 different people.

[6] `https://youtu.be/3SVlrkiFNmU`

6.4 Interpretation

6.4.1 Threats to validity

The goal of the study was to show that it is better to use our application to perform these tasks than to not use it. Multiple validity threats exist, these were analysed and split into four categories as defined in TODO:citation: Internal validity, conclusion validity, construct validity and external validity.

Conclusion validity

The goal of the study was to show that it is better to use our application to perform these tasks than to not use it. Multiple validity threats exist, these were analysed and split into four categories as defined in TODO:citation: Internal validity, conclusion validity, construct validity and external validity.

6.5 Discussion

Chapter 7

Conclusion

Microservices are becoming more popular in the development of new applications. Many businesses, both large and small, are using microservices to design and deliver applications more rapidly and efficiently [19]. Microservices are especially well-suited for distributed, cloud-based systems, where they may benefit from the cloud's flexibility and scalability [4].

We performed a literature review on the subject of refactoring architecture from monolithic applications to microservices for this study. A total of one hundred and six primary research contributions were chosen, categorised, and analysed using a clear research protocol in order to collect pertinent migration data. A tool's target programming languages, the processes it uses to convert monolithic inputs into microservices, and the output of these identified microservices are the subject of analysis in this study.

Tool input needs were examined first. Despite increased interest in microservice migration using automated tools, the topic is still young, and current solutions have strict inputs rather than being adaptable. Raw source code and OpenAPI were one method tools to identify microservices from monoliths.

As for how existing tools output their identified microservices, the outcome would ideally be fully functional code ready for deployment, as it would make the migration process smoother and ensure that the resulting microservices have all necessary components, thus reducing the effort needed for manual migration. From the research analysed, common outputs from microservices identification tools include a list of candidates and source code.

The proposed tool architecture consists of three distinct components. The frontend is responsible for the interface between the tool logic and the user. The backend serves as a bridge between the frontend, the database of available tools, and the tool domain. Each tool's adapter provides a consistent interface for interacting with the tool runtime. Ideally, each component of the proposed tool architecture would be deployed as a microservice in a Docker container to facilitate scalability and cross-platform deployment. This approach allows us to process multiple jobs concurrently and avoid potential bottlenecks in the communication between the tool runtime and the backend.

7.1 Future Work

References

[1] Desktop operating system market share worldwide. Last accessed 7 January 2023.

[2] Paul Becker, Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[3] Lianping Chen. Microservices: architecting for continuous delivery and devops. In *2018 IEEE International conference on software architecture (ICSA)*, pages 39–397. IEEE, 2018.

[4] Martin Fowler. Microservice prerequisites, 2014. Last accessed 4 January 2023.

[5] Martin Fowler. Microservices, 2014. Last accessed 4 January 2023.

[6] Martin Fowler. Microservice trade-offs, 2015. Last accessed 4 January 2023.

[7] Brendan J Frey and Delbert Dueck. Clustering by passing messages between data points. *science*, 315(5814):972–976, 2007.

[8] Erich Gamma, Ralph Johnson, Richard Helm, Ralph E Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[9] Konrad Gos and Wojciech Zabierowski. The comparison of microservice and monolithic architecture. In *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pages 150–153. IEEE, 2020.

[10] David Gough, Sandy Oliver, and James Thomas. *An introduction to systematic reviews*. Sage, 2017.

[11] Justas Kazanavičius and Dalius Mažeika. Migrating legacy software to microservices architecture. In *2019 Open Conference of Electrical, Electronic and Information Sciences (eStream)*, pages 1–5. IEEE, 2019.

[12] Barbara Kitchenham, O Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. Systematic literature reviews in software engineering–a systematic literature review. *Information and software technology*, 51(1):7–15, 2009.

[13] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. 2007.

[14] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.", 2016.

[15] Sam Newman. *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media, 2019.

[16] Sam Newman. *Building microservices*. " O'Reilly Media, Inc.", 2021.

[17] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai Josuttis. Microservices in practice, part 2: Service integration and sustainability. *IEEE Software*, 34(02):97–104, 2017.

[18] Zhongshan Ren, Wei Wang, Guoquan Wu, Chushu Gao, Wei Chen, Jun Wei, and Tao Huang. Migrating web applications from monolithic structure to microservices architecture. In *Proceedings of the tenth asia-pacific symposium on internetware*, pages 1–10, 2018.

[19] Chris Richardson. Microservice architecture. Last accessed 4 January 2023.

[20] Chris Richardson. Who is using microservices. Last accessed 4 January 2023.

[21] Peter J Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.

[22] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, 4(5):22–32, 2017.

Systematic Literature Review References

[23] Shivali Agarwal, Raunak Sinha, Giriprasad Sridhara, Pratap Das, Utkarsh Desai, Srikanth Tamilselvam, Amith Singhee, and Hiroaki Nakamuro. Monolith to microservice candidates using business functionality inference. In *2021 IEEE International Conference on Web Services (ICWS)*, pages 758–763. IEEE, 2021.

[24] Omar Al-Debagy and Peter Martinek. A new decomposition method for designing microservices. *Periodica Polytechnica Electrical Engineering and Computer Science*, 63(4):274–281, 2019.

[25] Omar Al-Debagy and Peter Martinek. A microservice decomposition method through using distributed representation of source code. *Scalable Computing: Practice and Experience*, 22(1):39–52, 2021.

[26] Lars van Asseldonk. From a monolith to microservices: the effect of multi-view clustering. Master's thesis, Utrecht University, 2021.

[27] Wesley KG Assunção, Thelma Elita Colanzi, Luiz Carvalho, Alessandro Garcia, Juliana Alves Pereira, Maria Julia de Lima, and Carlos Lucena. Analysis of a many-objective optimization approach for identifying microservices from legacy systems. *Empirical Software Engineering*, 27(2):1–31, 2022.

[28] Miguel Brito, Jácome Cunha, and João Saraiva. Identification of microservices from monolithic applications through topic modelling. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pages 1409–1418, 2021.

[29] Antonio Bucchiarone, Kemal Soysal, and Claudio Guidi. A model-driven approach towards automatic migration to microservices. In *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 15–36. Springer, 2020.

[30] Francisco Freitas, André Ferreira, and Jácome Cunha. Refactoring java monoliths into executable microservice-based applications. In *25th Brazilian Symposium on Programming Languages*, pages 100–107, 2021.

[31] Anup K Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. Mono2micro: a practical and effective tool for decomposing monolithic java applications to microservices. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1214–1224, 2021.

[32] Anup K Kalia, Jin Xiao, Chen Lin, Saurabh Sinha, John Rofrano, Maja Vukovic, and Debasish Banerjee. Mono2micro: an ai-based toolchain for evolving monolithic enterprise applications to a microservice architecture. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1606–1610, 2020.

[33] Rahul Krishna, Anup Kalia, Saurabh Sinha, Rachel Tzoref-Brill, John Rofrano, and Jin Xiao. Transforming monolithic applications to microservices with mono2micro. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, pages 3–3, 2021.

[34] Tiago Matias, Filipe F Correia, Jonas Fritzsch, Justus Bogner, Hugo S Ferreira, and André Restivo. Determining microservice boundaries: a case study using static and dynamic software analysis. In *European Conference on Software Architecture*, pages 315–332. Springer, 2020.

[35] Rina Nakazawa, Takanori Ueda, Miki Enoki, and Hiroshi Horii. Visualization tool for designing microservices with the monolith-first approach. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 32–42. IEEE, 2018.

[36] Vikram Nitin, Shubhi Asthana, Baishakhi Ray, and Rahul Krishna. Cargo: Ai-guided dependency analysis for migrating monolithic applications to microservices architecture. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[37] Luís Nunes, Nuno Santos, and António Rito Silva. From a monolith to a microservices architecture: An approach based on transactional contexts. In *European Conference on Software Architecture*, pages 37–52. Springer, 2019.

[38] Ilaria Pigazzini, Francesca Arcelli Fontana, and Andrea Maggioni. Tool support for the migration to microservice architecture: An industrial case study. In *European Conference on Software Architecture*, pages 247–263. Springer, 2019.

[39] Ana Santos and Hugo Paula. Microservice decomposition and evaluation using dependency graph and silhouette coefficient. In *15th Brazilian Symposium on Software Components, Architectures, and Reuse*, pages 51–60, 2021.

[40] Simone Staffa, Giovanni Quattrocchi, Alessandro Margara, and Gianpaolo Cugola. Pangaea: Semi-automated monolith decomposition into microservices. In *International Conference on Service-Oriented Computing*, pages 830–838. Springer, 2021.

[41] Xiaoxiao Sun, Salamat Boranbaev, Shicong Han, Huanqiang Wang, and Dongjin Yu. Expert system for automatic microservices identification using api similarity graph. *Expert Systems*, page e13158, 2022.

[42] Yuyang Wei, Yijun Yu, Minxue Pan, and Tian Zhang. A feature table approach to decomposing monolithic applications into microservices. In *12th Asia-Pacific Symposium on Internetware*, pages 21–30, 2020.

[43] Junfeng Zhao and Ke Zhao. Applying microservice refactoring to object-2riented legacy system. In *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*, pages 467–473. IEEE, 2021.