FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Visually-assisted Decomposition of Monoliths to Microservices

Breno Salles

Mestrado em Engenharia de Software

Supervisor: Prof. Jácome Cunha

July 19, 2023

Visually-assisted Decomposition of Monoliths to Microservices

Breno Salles

Mestrado em Engenharia de Software

July 19, 2023

Abstract

In the world of software development, the concept of microservices is popular. This architectural style has received much attention in both business and academia, and converting a monolithic application into a microservice-based application has become a regular practice. Companies struggle with migrating their existing monolithic applications to microservices, and software architects and developers frequently face challenges due to a lack of complete awareness of alternative migration methodologies, making the migration process even harder.

This dissertation aims to structurally analyse the state of the art in migrating monolithic applications to microservices architectural style, mainly which tools help architects, engineers, and developers and how automated they are. A systematic literature review identified 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.

Furthermore, we present an extensible framework to help architects, engineers, and developers during the migration process by addressing gaps in understanding various migration tools and approaches, allowing for easy comparison between multiple options. The application combines multiple tools into one platform, allowing a comprehensive visualisation of migration proposals and making it easy to compare different options.

To evaluate the efficacy of the application, we conducted an empirical study that focused on assessing its usability and the associated workload experienced by users during its utilisation. The study yielded favourable outcomes concerning usability, indicating that participants found the application user-friendly and intuitive. However, the workload results were diverse, implying that the application's efficacy in mitigating overall workload during the decomposition process was not universally experienced among the participants.

Keywords: microservices, monoliths, decomposition, visualisation

Acknowledgments

I want to thank my advisor, Prof. Jácome Cunha, for his invaluable guidance and support throughout this work. His mentorship has been instrumental in keeping me focused during challenging times, and I sincerely appreciate his continued assistance.

I sincerely thank my friends and colleagues for their companionship and the memorable experiences we shared during group projects and late-night study sessions. Your presence and support have made these years more enjoyable and meaningful.

I am particularly grateful to my dear friend Hugo Pais for his calming presence and tranquillity, which gave me peace of mind even during the most challenging times.

I want to warmly thank Cátia Condez and Fátima Fonseca for their unconditional love, unwavering support, and profound presence in my life. Your support has made the challenging phases of my life something laughable.

Lastly, I owe a special thank you to my father, Vital Salles. I am indebted to you for shaping my character's positive and negative aspects, and I hope this work would have made you proud.

Breno Salles

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 [14], microservices tend to be separated, and loosely coupled [20]. Also while monoliths tend to be easier to develop they may scale poorly and are harder to maintain when compared to microservices [19]. Microservices are increasingly being used in the development of modern applications, particularly in the areas of cloud computing [1]. Many organizations, including large enterprises and startups, are adopting microservices as a way to build and deploy applications more quickly and efficiently [46]. Microservices are particularly well-suited for distributed, cloud-based environments, where they can take advantage of the flexibility and scalability of the cloud [36]. This type of architecture is already being applied in multiple well-known companies, like Uber, Netflix, eBay [47] [22], and also being followed by the rest of the herd when compared to monolith architecture [24].

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, and companies struggle with migrating their already existing monolithic applications to microservices [13]. To help address this, some tools were developed that help with the refactor [55, 57, 58], 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 decompositions with different possibilities exists.

In this dissertation, we structurally analyse the state of the art in regards to the migration of monolithic applications to the microservices architectural style, mainly which tools help architects, engineers and developers in this migration, and how automated they are.

To achieve this, the guidelines presented by Kitchenham and Charters [16] 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 [16], 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. Is there an application that aggregates those tools to help architects, engineers and developers in their monolith decomposition?

Furthermore, we also develop an application that aims to aggregate existing tools into a single platform and provide the means to extend and incorporate new tools. This application offers a convenient and comprehensive way to access and use various tools that help the decomposition from monoliths to microservices and provide them with a perspective on several decomposition proposals, allowing for easily comparable and different combinations options.

The purpose of the application is to answer the following research question:

RQ3. Can we devise an application that aggregates existing tools in a single comprehensive environment to help in the decomposition to microservices?

RQ3.1. Can the application exhibit a good usability for decomposing?

RQ3.2. Does the application provide a low workload?

It is important to mention that the final objective is not to create a new technique for discovering microservices from a monolith system, but rather to aggregate the already existing ones into a single and comprehensive framework.

We conducted an empirical study to evaluate the quality of the application, focusing on assessing the usability and workload associated with its usage. The study yielded favourable outcomes in terms of usability, indicating that participants found the application to be user-friendly and intuitive. However, the workload results were varied, suggesting that the application's impact on reducing the overall workload during the decomposition process was not consistently observed among participants.

The rest of this dissertation is structured as follows: Chapter 2 introduces the reader to various concepts of monolithic and microservices architectures. Chapter 3 contains the literature review of the current state of the art. The architecture and design of the application is explained Chapter 4. Chapter 5 discusses the development of the application. In Chapter 6, we present the study design, its execution and corresponding analysis of results. Chapter 7 discusses the results obtained and their threats to validity. Finally, Chapter 8 ends with with some conclusions and future work.

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 [14]. 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 [14].

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 [20]. 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 [37].

One key benefit of this approach is that it allows for greater flexibility and scalability [19]. 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 [19, 20] [37].

Another advantage of microservices is that they can be developed and maintained by small, autonomous teams [6]. 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 [18].

However, there are also challenges to consider when adopting a microservices architecture [38]. 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 [20]. In addition, the communication between microservices can add latency to the system, which may impact the performance of the overall application [38] [21].

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 [20]. However, it is important to carefully evaluate the trade-offs and consider whether the benefits of a microservices architecture are worth the added complexity [38].

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 [19].

There are several factors to consider when refactoring an existing codebase for a microservices architecture [19]. 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 [19].

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, 15, 16]. 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 [16].

As per Kitchenham and Charters guidelines [16], 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, 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.

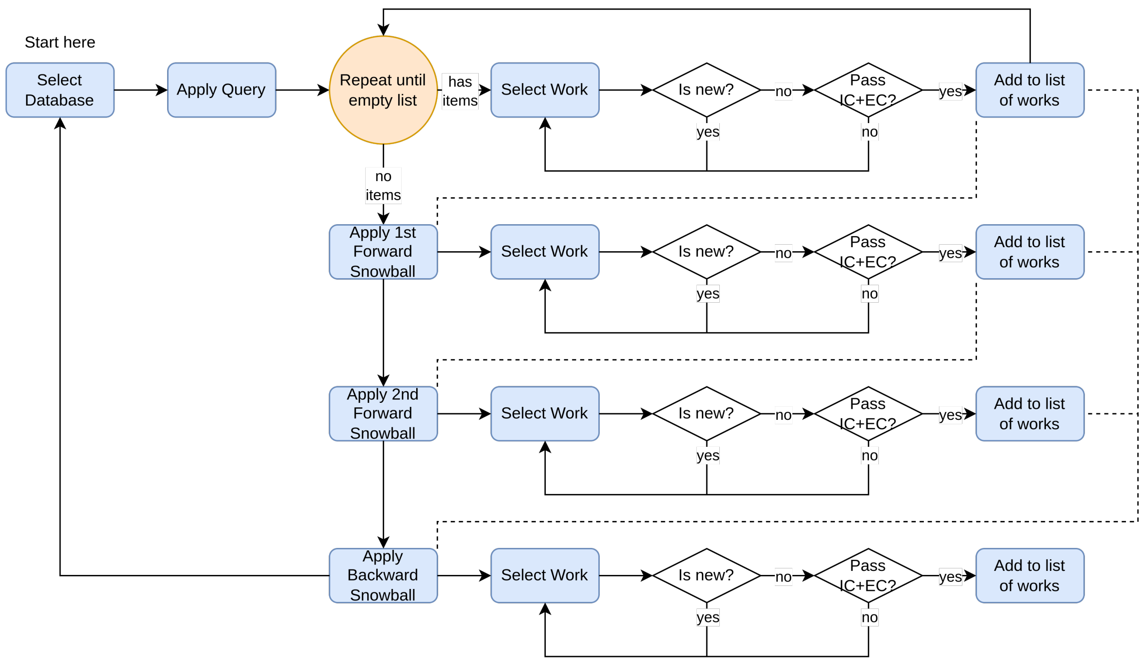


Figure 3.1: Review iterations

Table 3.1: Database Selection

ID	Search Engine	Website
ACM	ACM Digital Library	`https://dl.acm.org/`
IEEE	IEEE Xplore	`https://ieeexplore.ieee.org/`
SPL	Springer Link	`https://link.springer.com/`
WLY	Wiley	`https://onlinelibrary.wiley.com/`
SCI-D	Science Direct	`https://www.sciencedirect.com/`
ENG-V	Engineering Village	`https://www.engineeringvillage.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

Initially the focus was in 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 (in an initial phase, we did not have all terms in Table 3.2):

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

To increase the number of works found, we changed the focus to be on the location of 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.

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 some databases, 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, since in these parts of the documents, the authors tend to give more focus to what the work is really about. 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. Publication is not accessible.

3.2 Research Results

The initial query mentioned in the Section 3.1.3 was applied to 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 3.3.

The reason we used these search engines in detriment of others were:

- GitHub and GitLab: both are one of the most popular source code hosting platforms, which would increase our chance of finding relevant results.
- DuckDuckGo: the one we thought would less likely influence results.

Table 3.3: 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

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.

Applying the the query to the databases yielded 1394, with 34 that were extracted for having passed the selection criteria defined in Section 3.1.4, as shown in Table 3.4.

In the case of Science Direct, as presented in Table 3.4, two queries were done. The main reason for this is that Science Direct is limited to 7 *OR* conditions, therefore it was necessary to 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:

Table 3.4: DB Results

Database	**Total number of results**	**Extracted Results**
ACM	568	15
IEEE	4	0
SPL	678	3
WLY	9	3
SCI-D (1st)	21	1
SCI-D (2nd)	0	0
ENG-V	114	12

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 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 3.5. This helped us to increase the number of relevant publications that we were able to consider in the next steps of the process.

Table 3.5: Snowballing Results

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

Having iterated 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 marked the end of our general search for relevant publications. We were able to find 106 relevant publications.

All the results that were analysed from of the search are available in a gist[1].

3.2.1 Publications Grouping and Selection

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

[1] `https://gist.github.com/Guergeiro/c3baefb0ac6fdf673866f6515f1416a3`

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 3.6.
  - *Data flow*.
  - *Dependency analysis*.
  - *Execution log*.
  - *etc*.
- The current **status** of the publication, Table 3.7.
  - 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 3.8.
  - *Java*.
  - *Cpp*.
  - *C*.
  - *Language Agnostic*.
  - *etc*.

The publications that were selected are grouped in Tables 3.6, 3.7 and 3.8. 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[2].

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. The tools that fulfilled these requirements are:

- `https://github.com/HduDBSI/MsDecomposer` [68]

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

Table 3.6: 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

Table 3.7: Status grouping

Status	**Amount**
Method	48
Algorithm	7
Tool	25
Unknown	1

- `https://github.com/FranciscoFreitas45/MicroRefact` [57]
- `https://github.com/miguelfbrito/microservice-identification` [55]
- `https://github.com/gmazlami/microserviceExtraction-backend` [50]
- `https://github.com/socialsoftware/mono2micro` [64]
- `https://github.com/antbucc/Migration` [56]

Table 3.8: Language grouping

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

- https://github.com/tiagoCMatias/monoBreaker [61]

3.3 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.

3.3.1 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. [57] tool, MicroRefact[3], 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 [51,68].

The tool proposed by Al-Debagy and Martinek [51] 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 [23] which results in the identification and grouping of similar microservices, helping in the decomposition of the monolithic system.

MsDecomposer[4] [68] 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, 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 [67] or utilising the history of changes to understand in addition to common artefacts like classes and methods [66].

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

3.3.2 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 [51] 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 [58–60, 62]. 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. [57], and another is Mono2Micro [58–60].

3.3.3 Tool target language

The majority of works utilised Java as their primary programming language for input [50, 52, 54, 56, 58–60, 63, 65, 70]. 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 [55, 57, 64–66, 69], 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 [53, 61].

3.4 Summary

Based on the literature review and analysis presented in the preceding sections, the research question *RQ1 - "What tools already exist that aid in the migration process of monoliths to microservices?"* was examined. As mentioned in Section 3.2.1, limited tools are available to decompose monolithic architectures into microservices. We identified seven free and open-source tools [50, 55–57, 61, 64, 68], each with varying degrees of completeness, as discussed in the relevant literature. However, it is worth noting that the most promising tool identified, IBM's Mono2Micro [58–60], is not accessible to the general public.

Addressing the following research question, *RQ2 - "Is there an application that aggregates those tools to help architects, engineers, and developers in their microservice migration?"* the research findings revealed a lack of such an application. We found no existing application that served as an aggregator of multiple decomposition tools, offering users a graphical and unified interface. Therefore, this dissertation addresses this gap by proposing a solution that precisely fulfils the need for an application that aggregates decomposition tools, providing a user-friendly and consolidated platform for microservice migration activities.

Chapter 4

Tool Design

During our analysis described in Section 3.3, locating any relevant works that could address the second research question (RQ2) was impossible. As a result, it will be necessary to tackle this question and strive to answer it.

In order to address RQ2, we intend to develop an application with several functionalities. The functionalities we aim to include should answer a specific problem.

Problem

Multiple tools exist, but how do we present them for the user to choose?

We create a list of all available tools that users can select which one they want to use for a decomposition task.

Problem

Each tool accepts a language that might not be the same to others, how can we show and handle this information?

In the tool list we also show the languages compatible with it. Upon selecting one of them, you can only select more if they share a common language.

Problem

Tools may also have a parameters that may be tuned to optmise the decomposition output, therefore we should allow users to tune them.

For tools that are selected, and if the tool allows parameter tuning, we allow users to fill in the parameter values.

Problem

For each of the tools selected, one or multiple decompositions may be produced, how to compare them efficiently?

Give a interface where users are able to select the decompositions they want to compare, toggle to view more information of each specific decomposition, evaluate the relationship between each microservice in the cluster of microservices that make the decomposition.

In Section 4.1, we gather some functional requirements that break down system features and functions as well as some non-functional requirements that determine how the system will implement these features. Some functional requirements were derived of the problems stated. Section 4.3 contains some mockups of a possible frontend interface.

4.1 Requirements

Software requirements, 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 [25]. We used the MoSCoW method to prioritise each requirement, which is an acronym from the first letter of each prioritisation category: Must have, Should have, Could have, Will not have [3].

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.

4.2 Architecture

The application architecture consists of three distinct components, as presented in Figure 4.1. 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 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

Table 4.1: Functional Requirements

ID	Requirement	Priority
FR01	Frontend allows for user selection of a tool.	Must have
FR02	Frontend allows for user selection of a language.	Must have
FR03	Frontend allows for user parameter tunning.	Must have
FR04	Frontend allows for an user visualisation of the identified microservices.	Must have
FR05	Frontend allows for user code upload (either source code or specification).	Must have
FR06	Frontend allows comparison between multiple identifications.	Must have
FR07	Frontend allows for user session.	Should have
FR08	Frontend allows for download of output.	Could have
FR09	Frontend allows for upload of previous outputs.	Could have
FR10	Backend does not halt when an identification is running.	Must have
FR11	Backend starts the identification as soon as the input is uploaded.	Must have
FR12	Backend must serve the current existing approaches/languages/parameters to the user.	Must have
FR13	Backend must signal the user when the identification is completed.	Must have
FR14	Backend must signal the tool domain to start processing a code.	Must have
FR15	Tool domain has at least one approach.	Must have
FR16	Tool domain with each approach running separatly.	Should have
FR17	Tool domain with extensible adapter for other approaches.	Must have
FR18	Tool domain signals the backend when it finishes processing.	Must have

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.

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.

Table 4.2: Non-Functional Requirements

ID	Requirement	Priority
NFR01	Frontend must be deployed individually.	Must have
NFR02	Frontend must be deployed using Docker.	Must have
NFR03	Frontend could be split into multiple microservices.	Could have
NFR04	Backend must be deployed individually.	Must have
NFR05	Backend must be deployed using Docker.	Must have
NFR06	Backend could be split into multiple microservices.	Could have
NFR07	Tool domain should be deployed individually.	Should have
NFR08	Each tool implementation deployed individually.	Could have

4.3 Interface Overview

We created a set of mockups to guide the implementation of the application's user interface. These mockups served as visual representations of the intended design and layout of the user interface, aiding in the visualisation and communication of the desired interface elements and interactions. The mockups acted as a blueprint for the development process, assisting the implementation in achieving the envisioned user interface design.

Figure 4.2 primarily facilitates the selection of the tool and programming language to be used, thereby addressing the requirements FR01 and FR02. Figure 4.3 is dedicated to fulfilling FR05, which pertains to the capability of uploading the project's source code. Figure 4.4 addresses FR03, which involves the ability to perform parameter tuning for each selected tool. This figure showcases the user interface elements and options that enable users to adjust and fine-tune parameters associated with the chosen tool. Figure 4.5 illustrates the provision of feedback regarding the status of the decomposition process after the project upload. This feedback mechanism keeps users informed about the progress and status of the decomposition, allowing them to track the ongoing process. Figure 4.6 focuses on the frontend's capability to visualise the identified microservices and facilitate comparisons between multiple identifications, addressing FR04 and FR06. This figure demonstrates the user interface elements that enable users to visually explore and analyse the identified microservices, supporting the comparative analysis of different decompositions.

In Figure 4.6, we propose representing each microservice as a circular geometric shape, with the shape's size determined by a metric derived from the underlying tool. This metric could be based on factors such as the number of classes, the amount of code, or any other relevant metric determined by the tool. Additionally, we aim to illustrate the relationships between microservices within the visualisation by a line in which microservice dependencies, call stacks, or data flow, among other factors, define its stroke size.

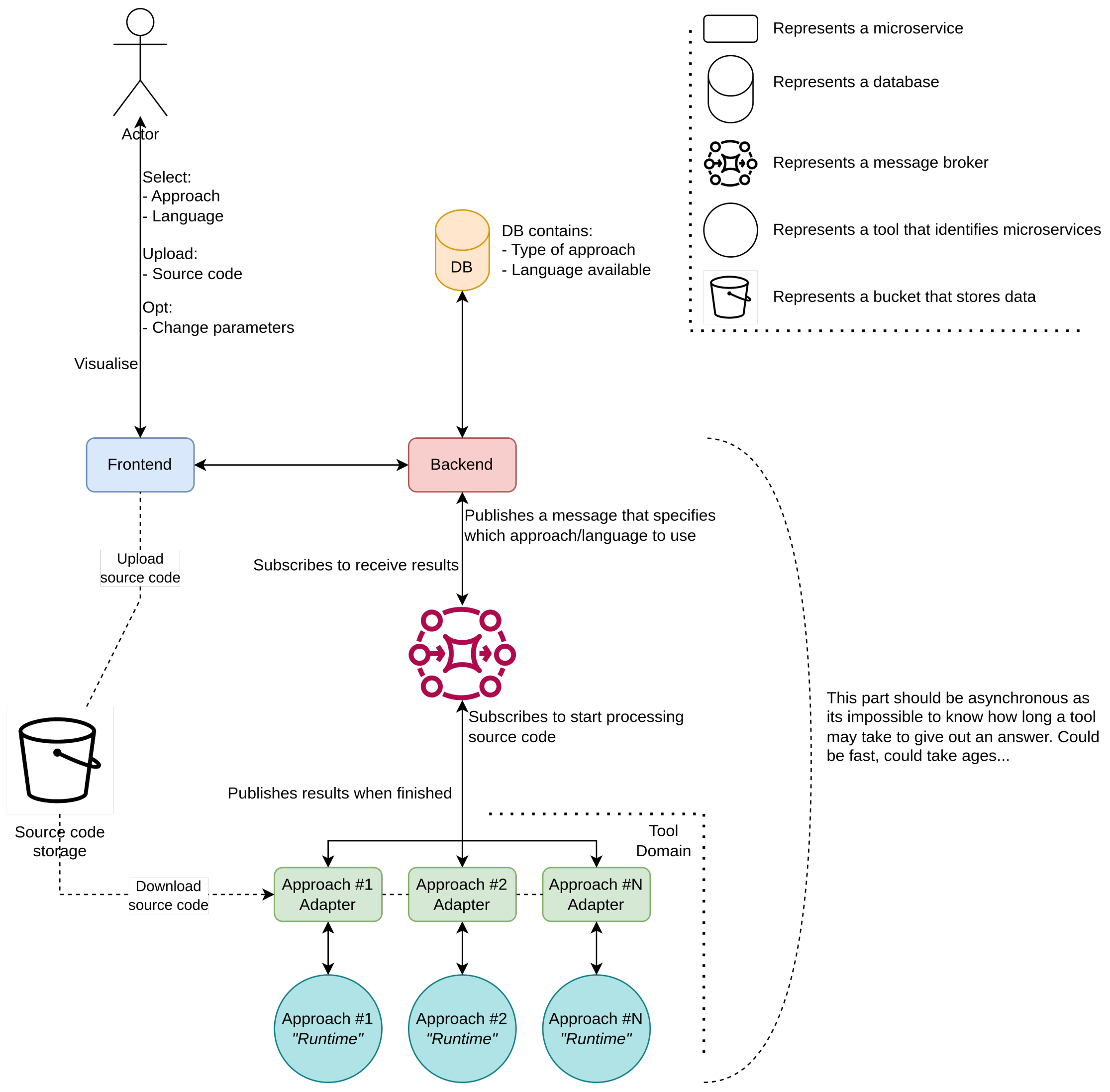
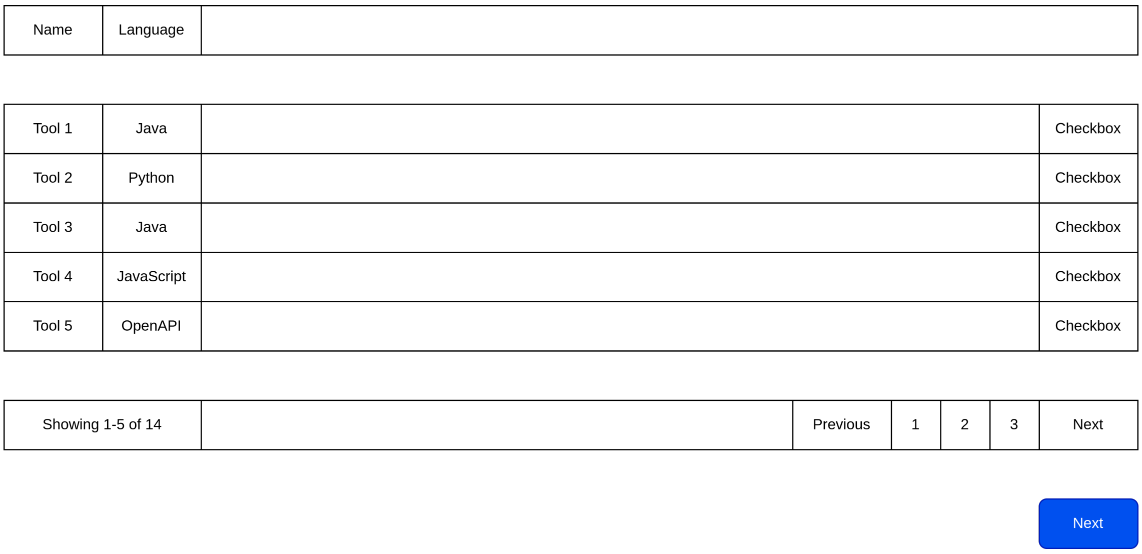

Represents a microservice
Represents a database
Represents a message broker
Represents a tool that identifies microservices
Represents a bucket that stores data
Actor
Select:
- Approach
- Language
Upload:
- Source code
Opt:
- Change parameters
Visualise
DB
DB contains:
- Type of approach
- Language available
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
Source code
storage
Download
source code
Tool
Domain
Approach #1
Adapter
Approach #2
Adapter
Approach #N
Adapter
Approach #1
"Runtime"
Approach #2
"Runtime"
Approach #N
"Runtime"
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...

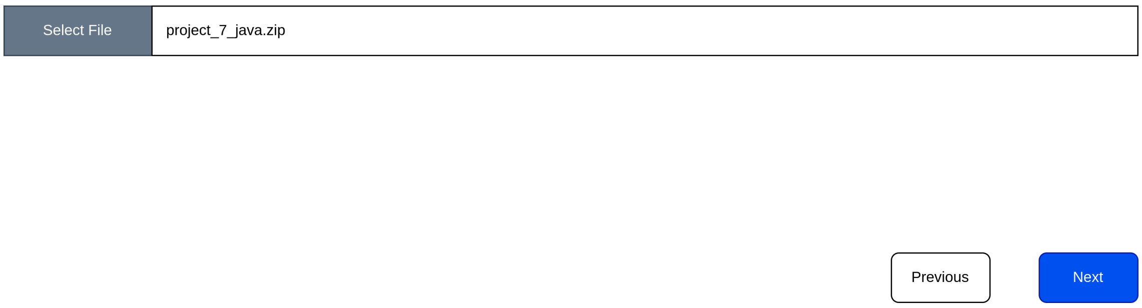
Figure 4.1: Application architecture



Name	Language		
Tool 1	Java		Checkbox
Tool 2	Python		Checkbox
Tool 3	Java		Checkbox
Tool 4	JavaScript		Checkbox
Tool 5	OpenAPI		Checkbox

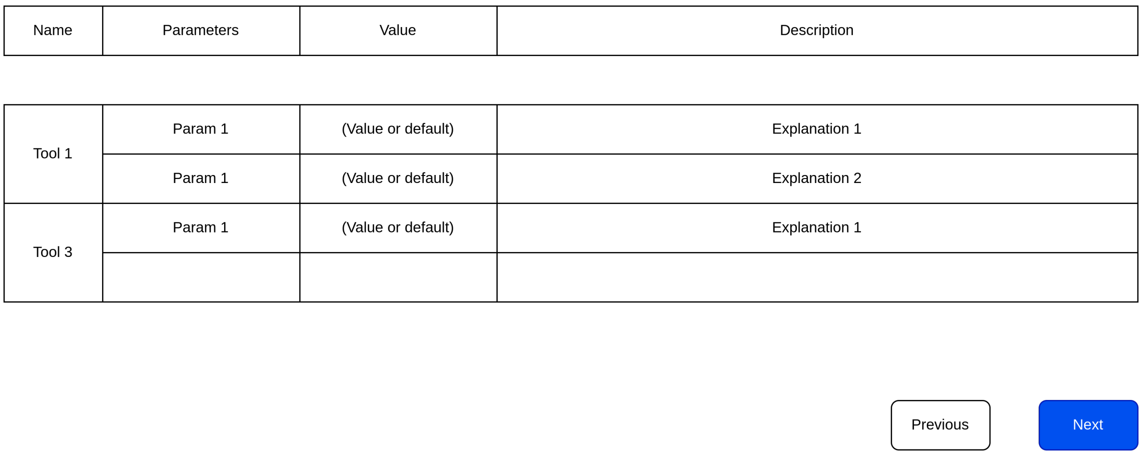
Showing 1-5 of 14

Figure 4.2: Tool/Language Selection



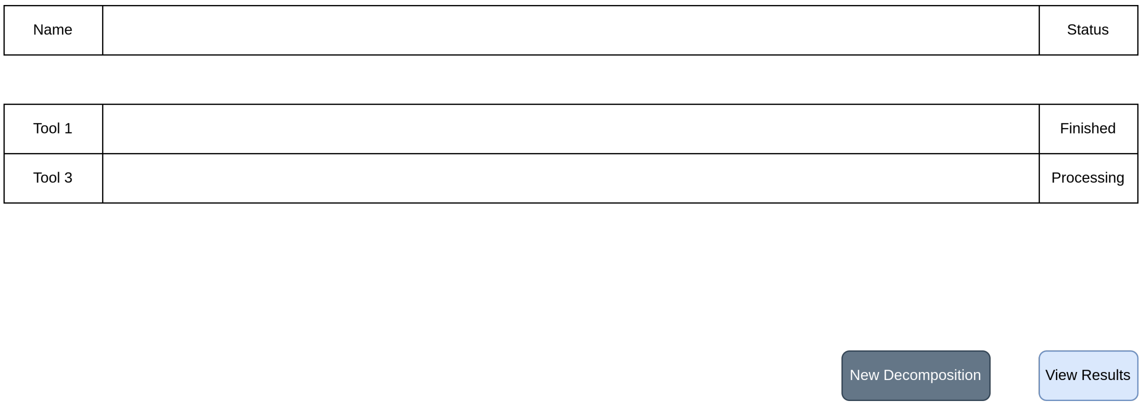
project_7_java.zip

Figure 4.3: Project or Source Code upload



Name	Parameters	Value	Description
Tool 1	Param 1	(Value or default)	Explanation 1
	Param 1	(Value or default)	Explanation 2
Tool 3	Param 1	(Value or default)	Explanation 1

Figure 4.4: Per tool parameter tuning



Name		Status
Tool 1		Finished
Tool 3		Processing

Figure 4.5: Decomposition Status

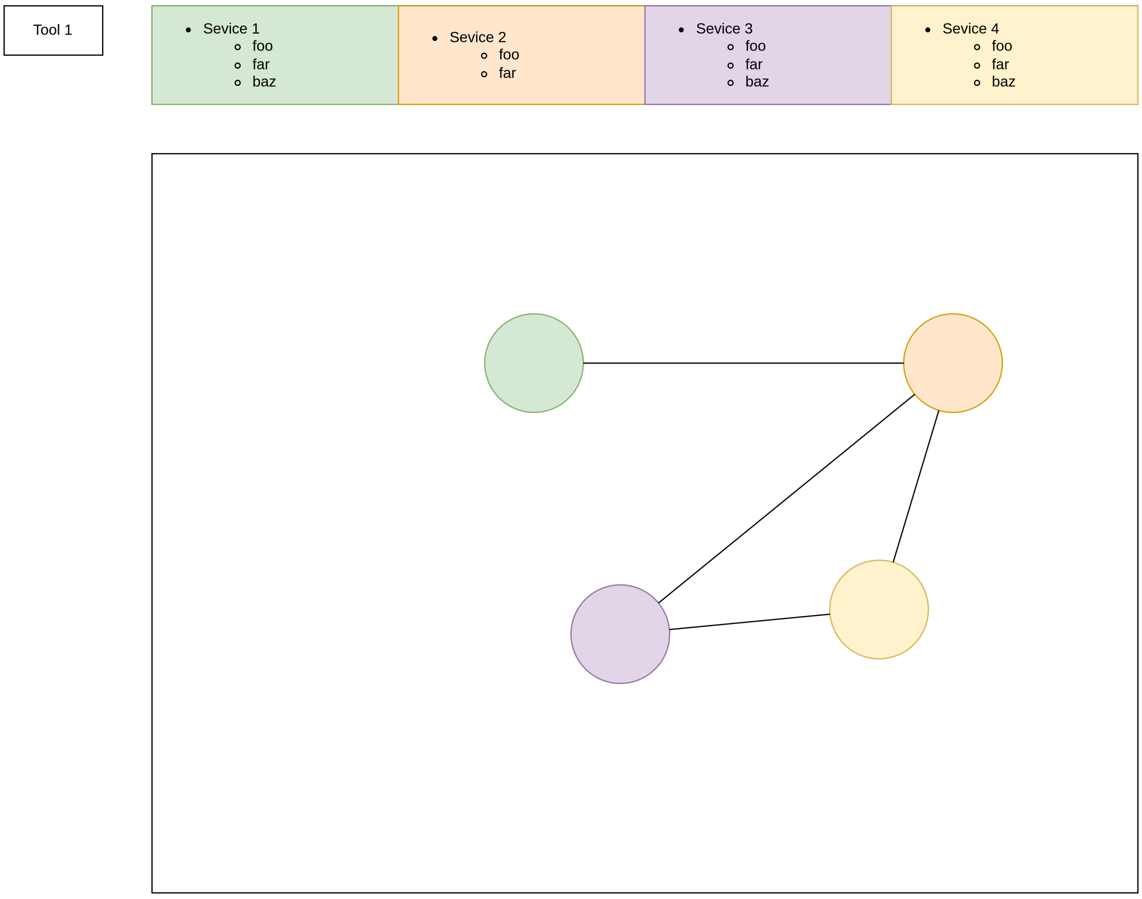


Figure 4.6: Decomposition Visualisation

Chapter 5

Solution Development

This section presents the prototype solution to the design mentioned in Chapter 4. We implemented the system's backend 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. As for the frontend, we used UI frameworks like React and SolidJS alongside TypeScript.

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 [30].

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 [33] paradigm to manage asynchronous I/O operations. This event-driven architecture enables Node.js to handle concurrent requests efficiently, maximising performance [34].

Moreover, the Node.js ecosystem offers modules via npm[1] 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[2], Vue[3], and Angular[4] utilise Node.js for tooling purposes, specifically

[1] `https://www.npmjs.com/`
[2] `https://github.com/facebook/react`
[3] `https://github.com/vuejs/core`
[4] `https://github.com/angular/angular`

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 [48], is considered a superset of JavaScript that introduces static typing to the language [41]. 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 [42] 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 [42].

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 adapter implementation of the tool.

Redis

Redis is an in-memory multi-model database famous for its sub-millisecond latency. It was created in 2009 by Salvatore Sanfilippo [49] based on the idea that a cache can also be a durable data store. Around this time, applications like Instagram [39] 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 [45], was adopted by some of the most popular 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 [43]. It can be used as a distributed key-value store, cache, and message broker [44]. 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 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 that Windows is the most widely used desktop operating system, followed by macOS and Linux [27], 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.

One of the most difficult decisions of a frontend developer is choosing between the UI framework; this is primarily due to the difficulty in switching once a particular framework is adopted. Astro addresses this by adopting a framework-agnostic approach [28]. It enables the creation of components in popular frameworks such as React, Svelte, Vue, Solid, among others, which translates into being unrestricted by the technological specifics of each framework, thereby enabling the integration of components from various ecosystems to speed up the development process.

React, also known as React.js or ReactJS, is a popular frontend library utilised for constructing component-based user interfaces [40]. Developed by Meta (formerly Facebook), React enables the development of single-page applications by providing a framework for building reusable and modular components.

Solid is a JavaScript framework developed by Ryan Carniato [29], explicitly designed for creating interactive web applications. It empowers developers to leverage their existing knowledge of HTML and JavaScript to construct reusable components that can be utilised across the entire application. It shares most of React's functionality but heavily focuses on bundle size and performance [29].

Docker

Docker is a tool that can package software into containers that run reliably in any environment [31]. One way to package an application is with a Virtual Machine (VM), where the hardware is simulated and installed with the required Operating System (OS) and dependencies. It enables the ability to run multiple applications on the same infrastructure; however, because each VM runs alongside its 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 [32]. Figure 5.1 demonstrates this difference.

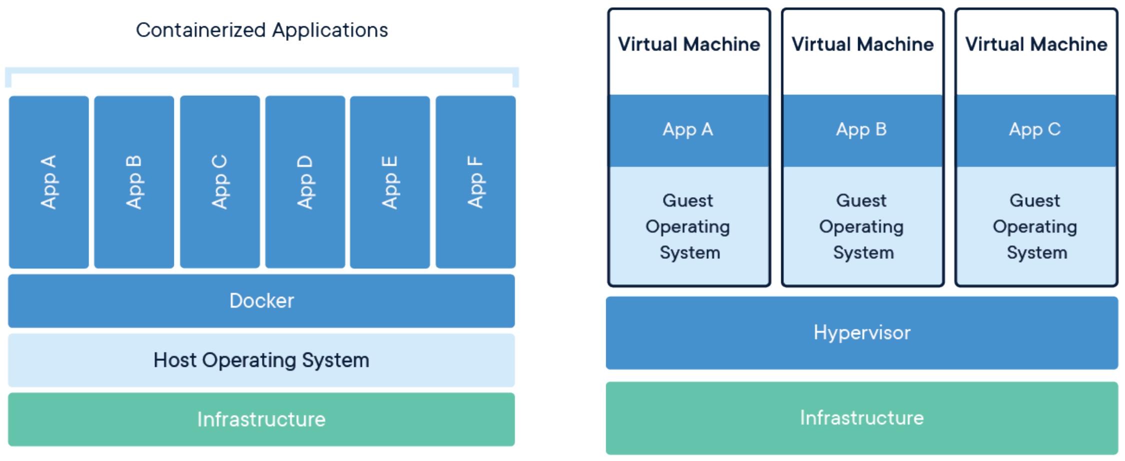


Figure 5.1: Container and Virtual Machines

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

5.2 Database

Structured Query Language (SQL) is generally considered the most effective programming language for managing structured data. In our system, there are five fundamental entities:

1. Tool - represents the underlying tool utilised during the decomposition process. It encompasses the specific software or framework for breaking down a system into smaller components.

2. Language - signifies the programming languages that can be targeted for decomposition. It encapsulates the various programming languages that the system supports for component extraction.

3. Result - denotes whether a decomposition job succeeded or encountered any issues.

4. Decomposition - encapsulates the actual decomposition and includes all the necessary data to represent it comprehensively.

5. Service - represents each microservice within the system.

The relationships between these entities are illustrated in Figure 5.2, visually depicting their interconnections.

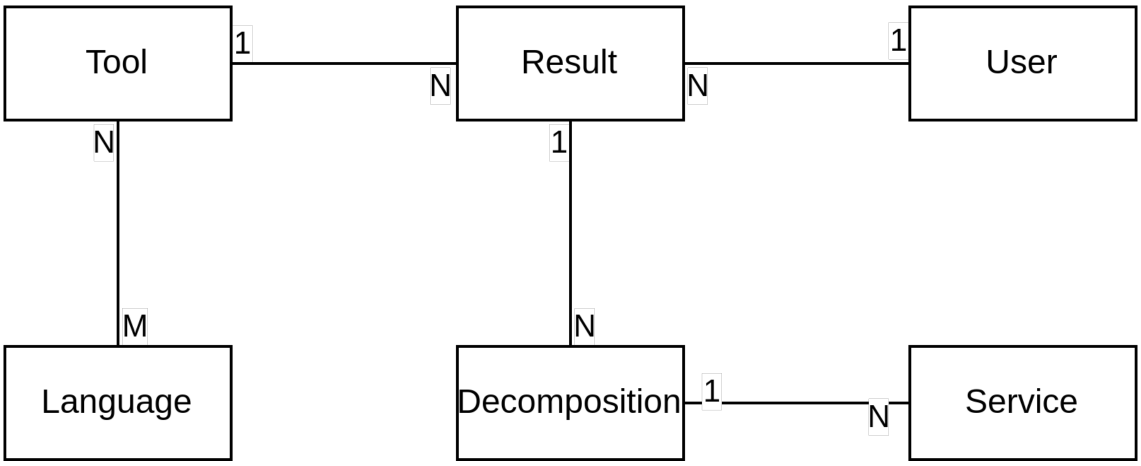


Figure 5.2: Database Model

5.3 Backend

As stated in Section 5.1, the entire backend of the system is developed using the Node.js runtime, the TypeScript language, and the NestJS framework. Following the specifications outlined in Chapter 4, both the backend and the tool adapter components are monolithic applications.

These components communicate through a message broker, utilising Redis as the underlying technology for message queuing and delivery. This messaging system enables seamless interaction and data exchange between the backend and the tool adapter, facilitating the flow of information within the system's infrastructure.

Figure 5.3 contains the entire decomposition workflow since the user requests a decomposition for a given tool, until it receives the result.

1. Client uploads (FR05) the project alongside the tools it want to use for decomposition.

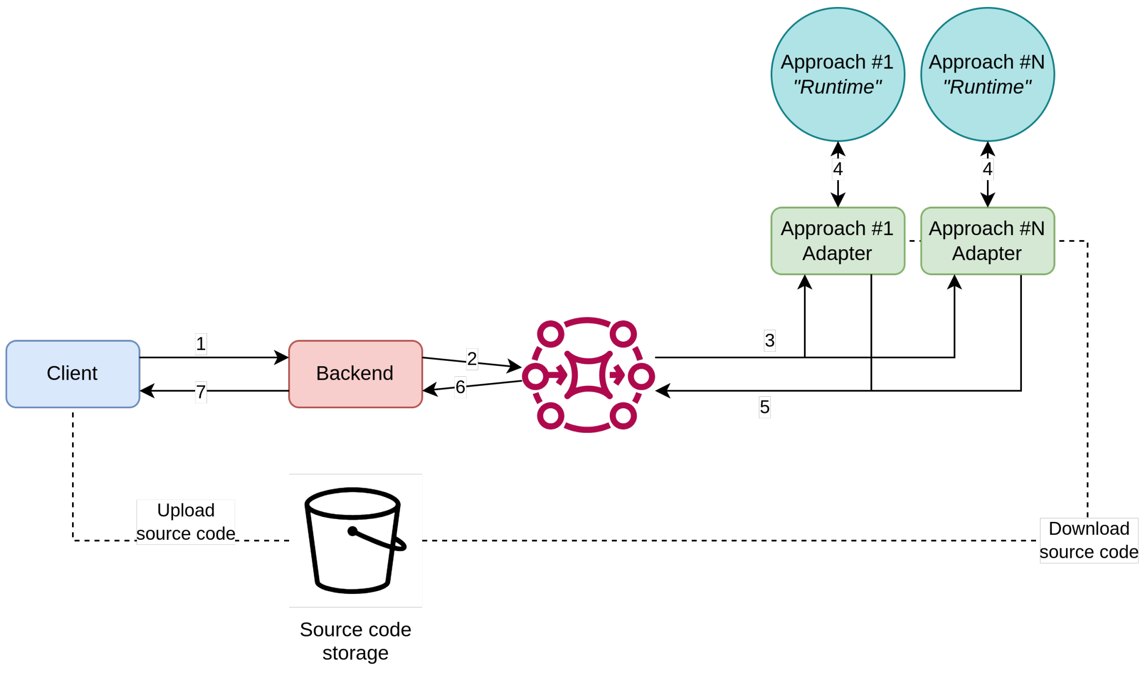


Figure 5.3: Decomposition Lifecycle

2. The backend signals each tool adapter (FR11, FR14) that matches the client's request with the unique identifier for that decomposition result as well as the identifier of the client's project on S3. This means the backend is free to handle new requests (FR10).

3. Each separate tool adapter (FR16) receives the unique identifier and starts an internal job for decomposition.

4. Each separate tool adapter (FR16) receives the unique identifier and starts an internal job for decomposition.

5. Upon finishing, each tool signals the backend (FR18) about the results, that are either success or failure.

6. When the backend receives the results from each tool adapter, it updates the database with the results so that it can always be accessible.

7. In the end, the backend reports back to the client with the status of the result (FR13).

5.3.1 API Design

The backend architecture follows the principles of the REST style and utilises JSON as the format for data transportation. The implemented REST API adheres to the guidelines of Level 2 in the Richardson Maturity Model [35], which signifies a high level of API maturity and conformity to REST principles.

Table 5.1 provides a comprehensive depiction of the available endpoints within the backend. These endpoints represent the various functionalities and operations that can be performed through the API, allowing clients to interact with the backend system and access its resources. By conforming to Level 2 of the Richardson Maturity Model, the backend ensures standardised and efficient communication between clients and the server, enhancing interoperability and scalability.

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

A lot of the time, when making calls to a REST API, there will be many results to return. Therefore, we paginate the results to ensure responses are easier to handle. Let us say the initial request asks for all the tools available; the result could be a massive response with hundreds of thousands of tools. There are better places to start than that. Therefore, we have built a limit on results to ensure that only that amount of results will be returned. This limit defaults to five results per page but can be changed at will.

We employed pagination on endpoints that may return multiple results, that is on GET requests to */tools*, */results* and */decompositions*. This allow us to confidently serve the content the client needs, for example, serving the existing a tools and corresponding data that composes each tool, thereby completing FR12.

5.3.2 Authentication and Authorisation

The authentication and authorisation mechanisms implemented within the system serve multiple purposes. It allows users to retrieve their previous decompositions and separate their own findings and knowledge from those of others. This ensures a personalised and enhanced user experience when returning to the application (FR07).

Authentication, as a standalone concept, is not explicitly implemented. Users can be created by making a POST request to the */users* endpoint, which generates a new user without requiring any additional fields.

On the other hand, authorisation is enforced for specific paths within the API. The paths */results* and their subpaths, as well as */decompositions* and their subpaths, require a user id to be accessible. This mechanism ensures that each user's content remains segregated, preventing mixing or unauthorised access to other users' data.

To provide a seamless user experience, the API client (e.g., the frontend application described in Section 5.4) needs to store the user id locally. This enables smooth integration when the user returns to the application, allowing them to retrieve their personalised data and settings effortlessly.

Further implementation details of the frontend application can be found in Section 5.4.4, providing a comprehensive overview of how the frontend interacts with the backend API to deliver the desired functionality and user experience.

5.4 Frontend

The application frontend is developed using Astro as the foundational framework while primarily utilising React and Solid as the underlying UI frameworks. The frontend implementation encompasses three main pages: the tool selection page, which also serves as the homepage, the results page, and the comparison page.

1. Tool Selection Page/Homepage (Section 5.4.1) - serves as the initial interface where users can select the desired tool for the decomposition process. It provides a user-friendly environment for tool exploration and selection, enabling users to make informed decisions about the tools they wish to utilise.

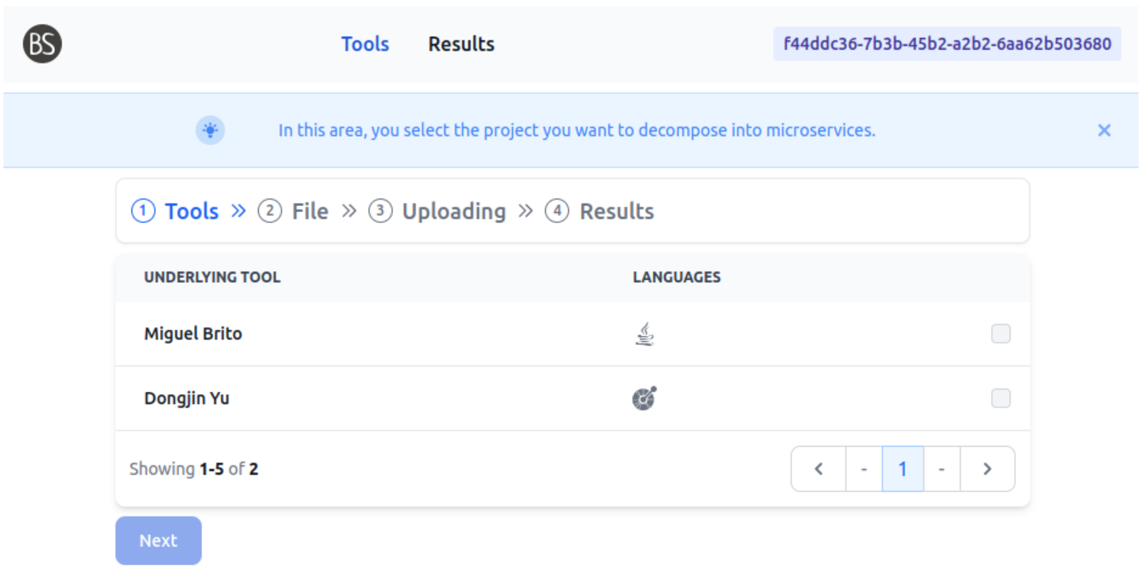
2. Results Page (Section 5.4.2) - displays the outcomes of the decomposition process. It presents the findings and relevant information, showcasing the success or failure of each decomposition job. Users can review and analyse the results, as well as select the decompositions for further comparison.

3. Comparison Page (Section 5.4.3) - facilitates a side-by-side evaluation of different decomposition results or tool combinations. It allows users to compare and contrast the outcomes of various decompositions, aiding in the identification of patterns, trends, and performance differences among the different tools employed.

5.4.1 Tool Selection

The tool selection page necessitates the user to choose the underlying tool for the decomposition process. Currently, the only implemented tool available for selection is the one developed by Brito et al. [55].

Figures 5.4 to 5.6 visually depict the step-by-step procedure involved in selecting a tool (FR01, FR02), uploading a file (FR05), and waiting for the decomposition to be performed. These illustrations provide a clear representation of the user interface and the corresponding actions taken by the user during the tool selection process. Figure 5.4 showcases the tool selection interface, where the user can choose the desired tool. Figure 5.5 exhibits the file upload functionality, allowing the user to upload a file that contains the project to be decomposed. Finally, Figure 5.6 illustrates the waiting process as the system performs the decomposition, providing the user with a progress indicator or other relevant information.


BS
Tools
Results
f44ddc36-7b3b-45b2-a2b2-6aa62b503680
In this area, you select the project you want to decompose into microservices.
① Tools » ② File » ③ Uploading » ④ Results

UNDERLYING TOOL	LANGUAGES
Miguel Brito	
Dongjin Yu	

Showing 1-5 of 2
‹ - 1 - ›
Next

Figure 5.4: Tool Selection

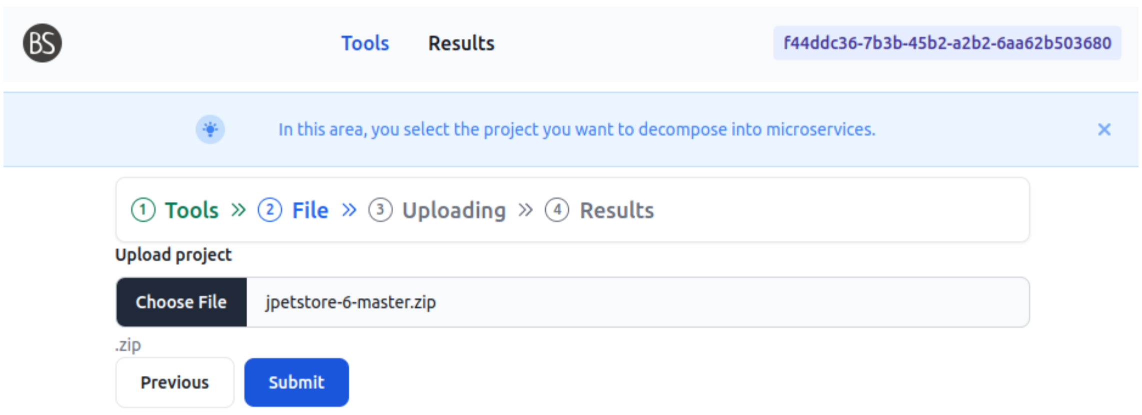

BS
Tools
Results
f44ddc36-7b3b-45b2-a2b2-6aa62b503680
In this area, you select the project you want to decompose into microservices.
① Tools » ② File » ③ Uploading » ④ Results
Upload project
Choose File
jpetstore-6-master.zip
.zip
Previous
Submit

Figure 5.5: Project Upload

5.4.2 Results

The results page displays a comprehensive overview of all the results from the user's previous decompositions. This page allows the user to select up to five decompositions for comparison purposes, providing the flexibility to mix and match results from different runs of the tool.

Figures 5.7 and 5.8 provide a visual representation of the results page and showcase the user interface elements and features available on this page, such as the list of decompositions, selection checkboxes, and any other relevant information the underlying tool provides, like weigths, and metadata.

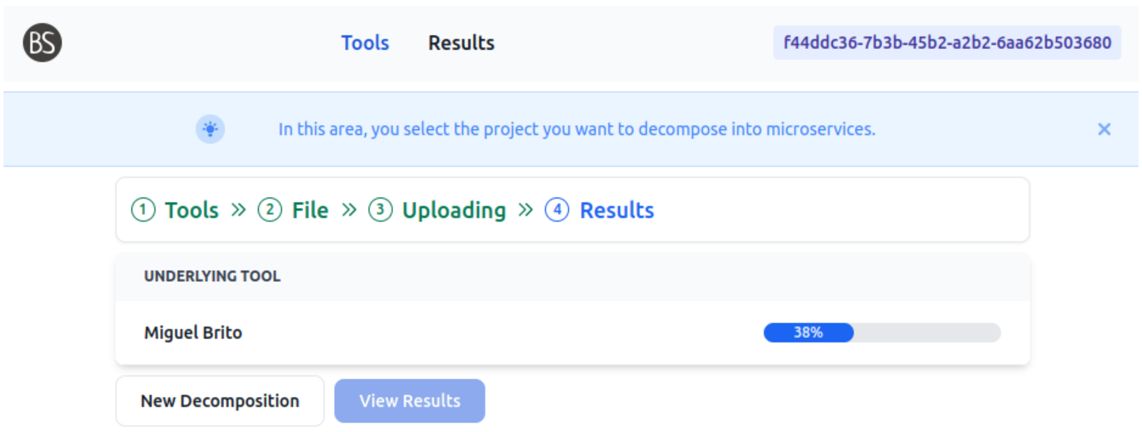
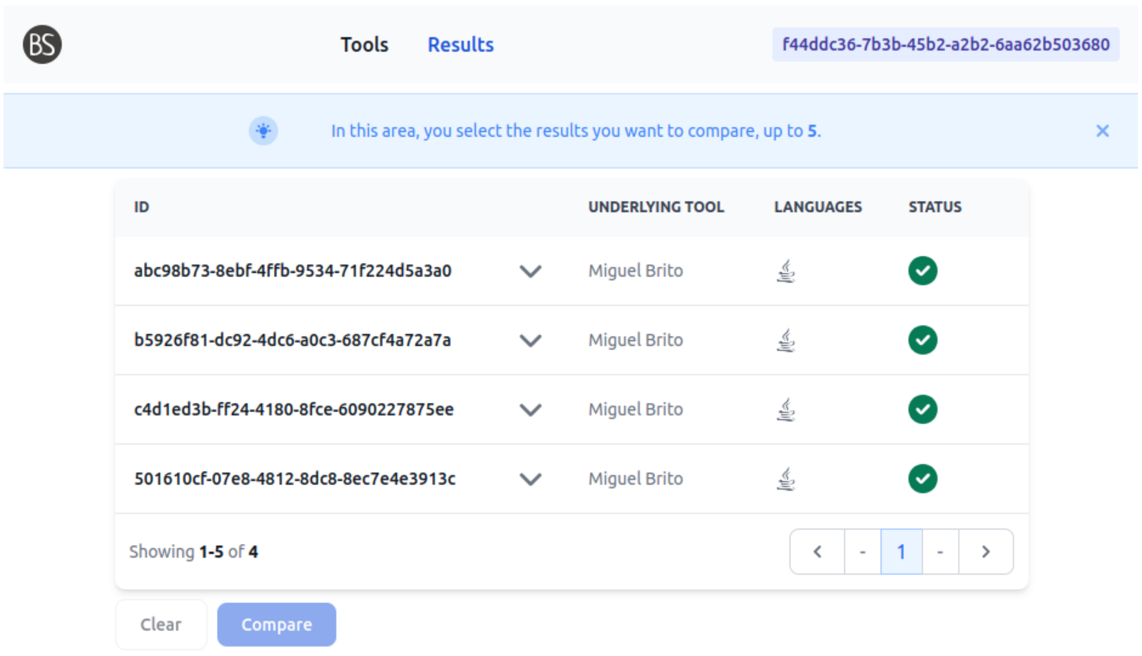

Tools
Results
f44ddc36-7b3b-45b2-a2b2-6aa62b503680
In this area, you select the project you want to decompose into microservices.
① Tools » ② File » ③ Uploading » ④ Results
UNDERLYING TOOL
Miguel Brito
38%
New Decomposition
View Results

Figure 5.6: Awaiting Decomposition


Tools
Results
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		✓
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**

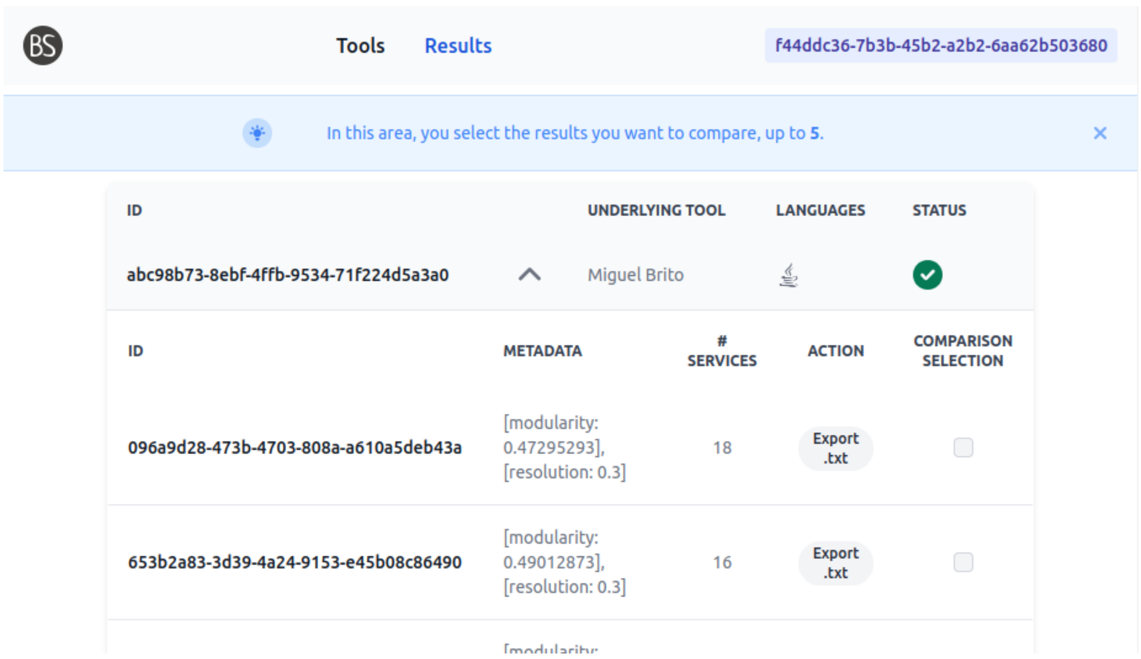
Figure 5.7: All Results

5.4.3 Comparison

The comparison page serves as the core component of the solution, providing users with essential functionalities. On this page, users can:

1. Visualise microservices generated by each decomposition, enabling users to gain insights into the composition and structure of the system components (Section 5.4.3).

2. Toggle the visibility of decompositions allowing users to focus on the ones they want to compare (Section 5.4.3).



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	☐

Figure 5.8: Expanded Result

3. Focus on a microservice and check its constituent modules where users can delve into the details of each microservice and understand its internal components (Section 5.4.3).

4. Compare modules across different decompositions, which helps users identify similarities, differences, and variations in the composition of the system components (Section 5.4.3).

During the evaluation of various tools for visualising the comparison view, four options were considered: Graphviz[5], D3.js[6], Gephi[7], and Chart.js[8]. Each tool was assessed based on four categories: Customizability, Ease of use, Charts available, and Real-time interactivity. A matrix analysis was conducted (as presented in Table 5.2) to assess the strengths and weaknesses of each tool.

Considering the industry's preference for node graphs in presenting microservices [5] and after careful consideration of the pros and cons, D3.js was determined to be the most powerful and suitable visualisation tool for the specific use case.

In addition to having a powerful tool, the way information is presented and the visual elements used are crucial. As suggested by Moody [17], using different shapes, colours, strokes, and line dashes to depict entities and their relationships is an effective way of expressing variation between entities in visualisations.

Table 5.3 illustrates how each visual representation tool expresses different entities, highlighting the specific shapes and visual cues employed to convey information effectively.

[5]https://graphviz.org/
[6]https://d3js.org/
[7]https://gephi.org/
[8]https://chartjs.org/

Table 5.2: Visualisation Tool Comparison

Visualisation Tool	**Customizability**	**Ease of use**	**Charts available**	**Real-time interactivity**
Graphviz	Limited customisation 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 customisable, suitable for creating custom and interactive visualisations.	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 visualisations.
Gephi	Customisable 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 visualisations.	Allows users to interact with graphs and manipulate layouts in real-time.
Chart.js	Customisable 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.*

Microservices Visualisation

As mentioned in Section 3.4, there is currently a lack of existing tools that provide a unified interface designed to inspect decomposed monolithic architectures. Most available tools generate output files, such as JSON, primarily intended for machine-to-machine communication rather than human interpretation. As humans, our natural inclination is to comprehend information visually, and visual representations allow us to understand better and grasp the differences between decompositions. Visual representations can convey complex concepts more effectively than textual representations, which require a comprehensive understanding of the underlying structure. Therefore, developing an application that offers a visual representation of decomposed monolithic architectures can significantly enhance the ability of humans to comprehend and analyse the decomposition process.

Figure 5.9 provides a visual representation of three different decompositions. Among these decompositions, two consist of clusters of microservices generated through the decomposition of the same project. Consequently, a connection exists between the modules within these two decompositions, indicating the relationship between the corresponding microservices. On the other hand, the third decomposition does not exhibit any connection or relationship with the other two decompositions. This visual representation helps users differentiate between the decompositions and perceive the interconnection of modules within related decompositions while highlighting the third decomposition's independent nature.

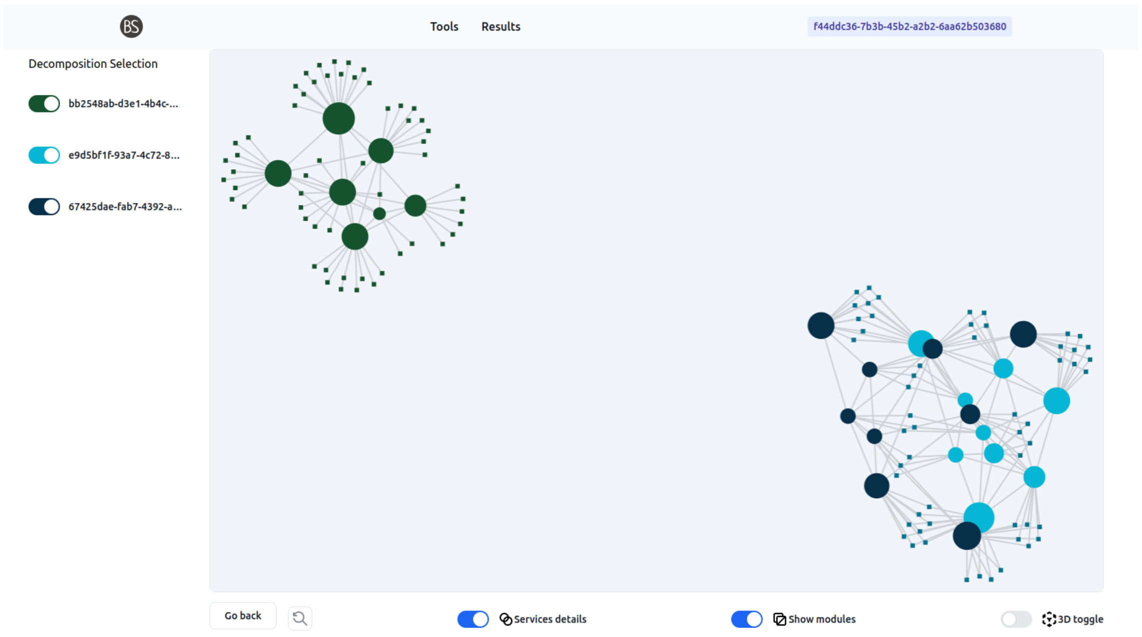
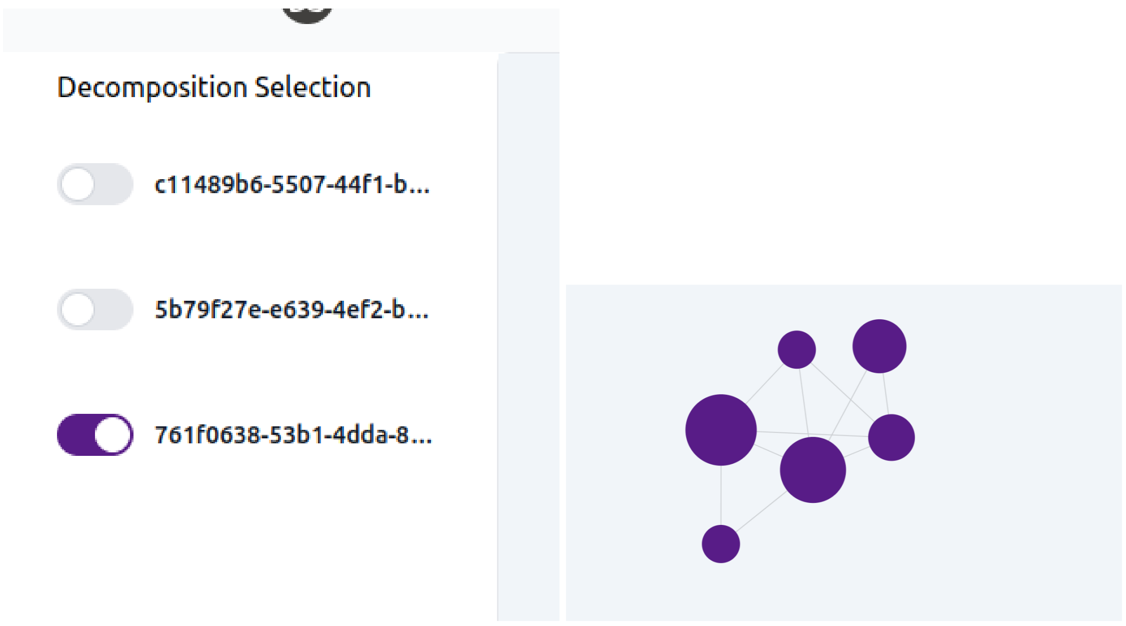


Figure 5.9: Comparison Page

Toggle Decompositions

The toggle visibility feature in the graphical interface allows users to control the display of specific decompositions. This functionality proves beneficial in various scenarios, such as when users want to concentrate solely on a single decomposition, compare two decompositions side by side, or exclude a discarded decomposition from the view. Without this feature, all selected decompositions would remain in an active state, potentially cluttering the visual interface. Figure 5.10 showcases this toggle visibility feature on and off states, enabling users to easily switch between displaying or hiding specific decompositions according to their needs and preferences.

(a) Off State



(b) On State

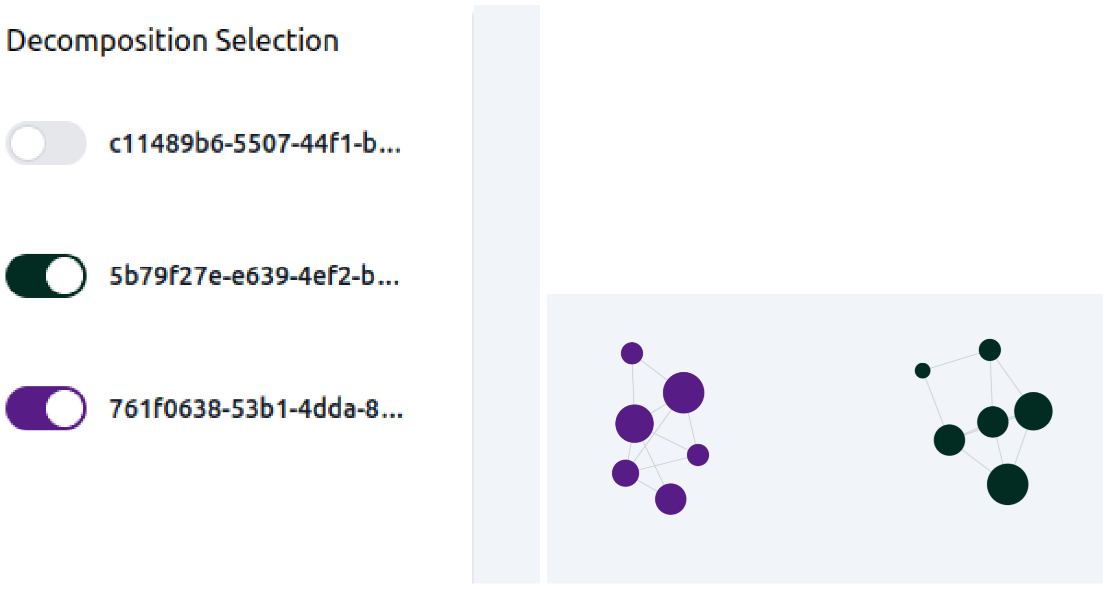


Figure 5.10: Toggle Decomposition

Microservice Focus

When a user clicks on a specific microservice within the graphical interface, the graph zooms into the selected microservice's centre. Additionally, a new window opens, providing detailed information about the modules contained within the focused microservice. This approach enables users to conduct a more in-depth analysis of each microservice, examining its composition and internal structure. Furthermore, this functionality facilitates comparisons at the service level, allowing users to compare multiple services across different decompositions. Figure 5.11 visually represents this feature, showcasing the zoomed-in graph with the focused microservice and the accompanying window displaying the modules associated with the selected microservice. This capability enhances the user's ability to analyse and compare microservices within the application.

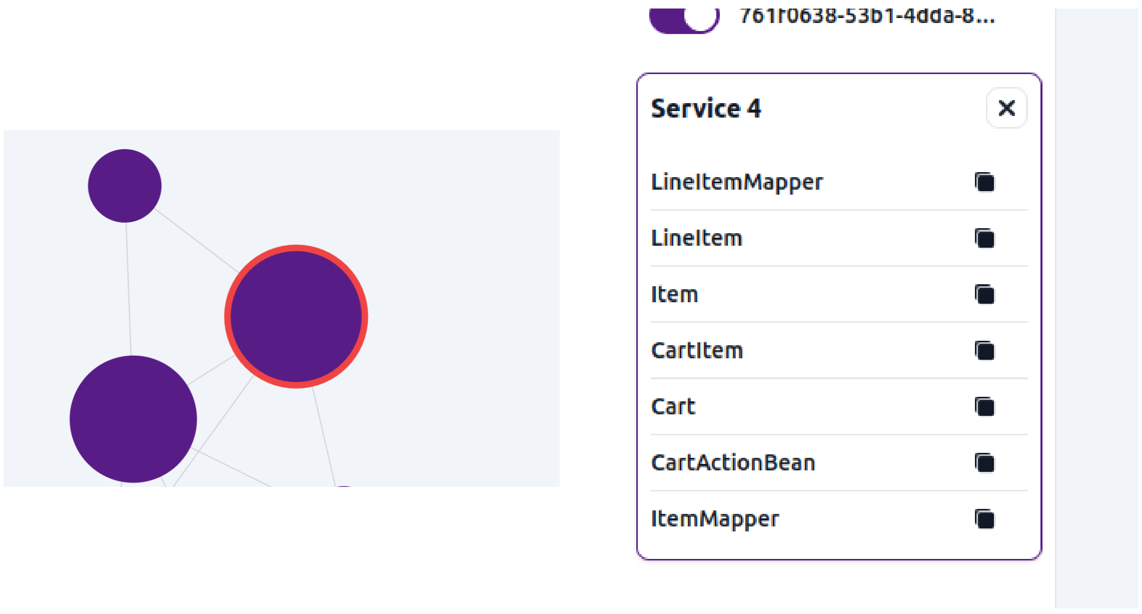


Figure 5.11: Microservice Focus

Modules Comparison

The application includes a toggle for visualising the modules within each microservice. This feature is the visual counterpart to the Modules Comparison functionality discussed in Section 5.4.3. However, there is a significant distinction. The toggle for visualising modules enables users to observe the connections and relationships between modules across different microservices and microservices from different decompositions. This capability aids users in visualising the mapping of components from their original monolithic structure to their placement within microservices. Figure 5.12 illustrates this functionality, showcasing the visualisation of module connections across microservices, facilitating a clearer understanding of the distribution and organisation of components within the application.

5.4.4 Authentication and Authorisation

In order to enhance the user experience of the authentication and authorisation mechanism described in Section 5.3.2, the application utilises local storage to store the user id. This approach

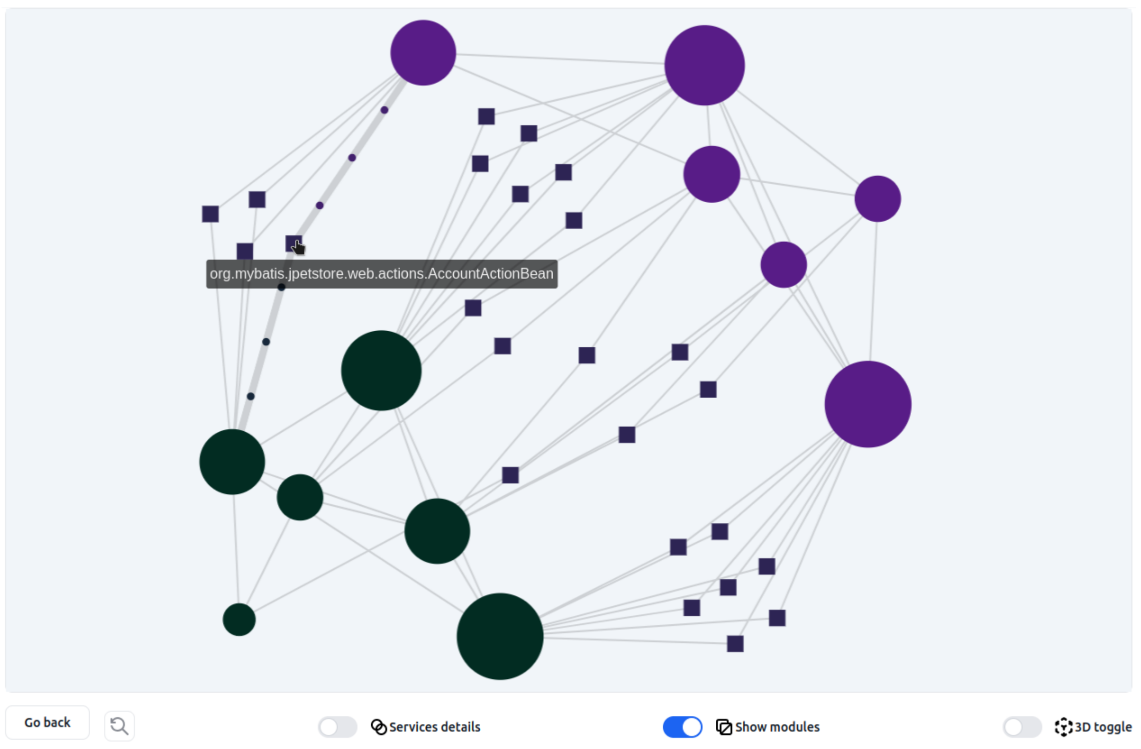


Figure 5.12: Modules Visualisation

provides a seamless and persistent user experience by retaining the user's identification information across sessions (FR07). The flowchart depicting the logic of this process is presented in Figure 5.13.

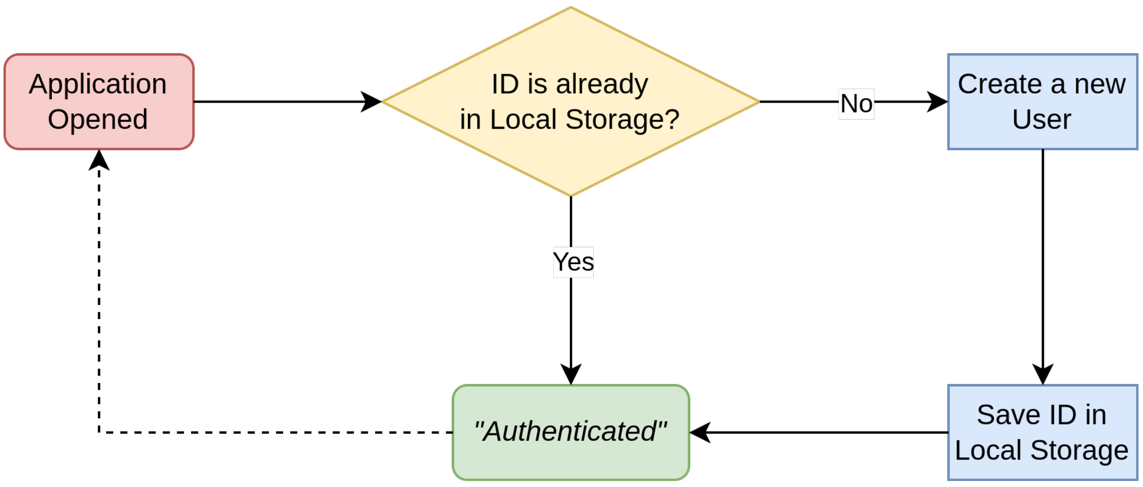


Figure 5.13: User ID storage Flowchart

However, it is essential to note that users can manually remove their user id from local storage. This action carries the risk of losing access to all their previous decompositions. Therefore, users should exercise caution when manually deleting their user id, as it may result in losing important data and restrict their access to previous work within the application.

The idea behind this *"crude"* implementation was not have typical persistent authentication, but more so to have a minimalist mechanism that helps the users in their decompositions and evaluation.

5.5 Deployment

Each system component is deployed as a microservice (NFR01, NFR04, NFR07, NFR08) in a Docker container (NFR02, NFR05) to facilitate scalability and cross-platform deployment. This is especially important for the tool runtime component, as we do not care how the specific tool is implemented as long as it subscribes to decompositions requests and publishes the decomposition results from and to the message broker.

The application incorporates a Continuous Integration and Delivery (CI/CD) process. Whenever a new push is made to the *master* branch of the Git repository, it triggers a GitHub Action that initiates the build process and deploys the applications. The hosting platform used for the application is Railway[9]. However, since the application is packaged with Docker, it can be hosted on any platform that supports Docker images, as explained in Section 5.1. This flexibility allows the application to be deployed to various hosting environments based on specific requirements and preferences. The tool is currently available at `https://frontend-mesw.brenosalles.com/`, and there is a video tutorial of the tool[10].

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

ID	**Requirement**	**Priority**
FR03	Frontend allows for user parameter tunning.	Must have *
FR08	Frontend allows for download of output.	Could have **
FR09	Frontend allows for upload of previous outpus.	Could have

** Parameter tuning was not implemented on the frontend, but the backend end expects the request to have parameters.*
*** It is possible to export the decomposition, but the exportation content would need to be revised to work with the upload requirement.*

[9] `https://railway.app/`
[10] `https://youtu.be/WQAAU9vQddA`

Table 5.4 contains requirements that primarily pertain to enhancing the experience 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

Empirical validation is used to validate a developed application. Thus, we prepared an empirical study to confirm that it helps decompose monoliths.

In this section, we present an empirical study we designed and ran to evaluate to which extent our tool actually aids in the migration of monoliths to microservices. The study design is presented in Section 6.1, outlining the methodology and approach employed. Subsequently, Section 6.2 explains the execution of the study, providing details on how the validation process was carried out. Data analysis is the primary focus of Section 6.3, where the collected data from the empirical study is analysed and evaluated.

6.1 Design

As mentioned in Chapter 4, we developed an application to address RQ2 stated in Chapter 1. It is important to note that the validation of such a tool is not solely based on the quality of the microservices decomposition itself, as this responsibility lies with the underlying tools used for decomposition. Instead, the validation focuses on how effectively and efficiently the application aids architects, engineers, and developers in their migration processes.

The study conducted aims to evaluate the effectiveness and efficiency of users utilising the application in performing decomposition tasks. Once participants complete the task, they are requested to fill out a questionnaire based on the System Usability Scale (SUS) [4] and Raw Task Load Index (Raw-TLX) [11].

The target audience for this study is software developers, engineers, and architects at various levels of expertise. We created a questionnaire (Appendix A) that serves as a structured tool for collecting data and participant feedback, aiding in evaluating and analysing the application's usability and task workload performance.

6.1.1 Subjects and Objects

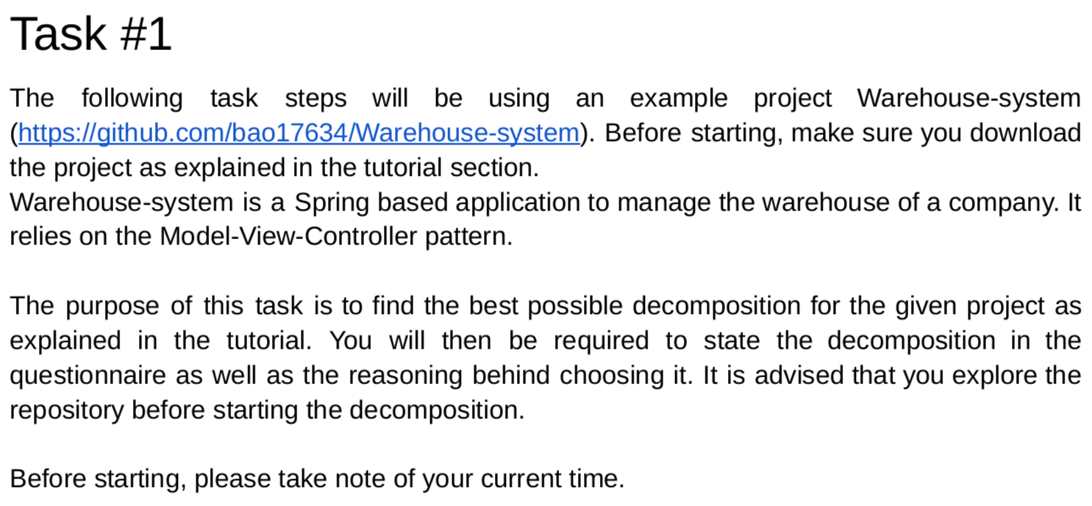
In the study, the subjects refer to the individuals who are the target participants for evaluation. In contrast, the objects refer to the tasks or activities the participants perform during the study.

For the subjects, the study aims to target a broad demographic of participants. The population includes senior-level developers, engineers, and architects who can assess the viability of the application in their migration processes and more junior-level users looking into a more accessible entry point for monolith decompositions. By including participants from both ends of the experience spectrum, the study can gather insights and feedback from a diverse range of users.

As for the objects, the subjects are required to complete a monolith decomposition task. The specific tasks are consistent across all participants, but the monolith projects targeted for decomposition differ. Three projects are considered, one serving as a tutorial or instructional example, while the other two are reserved for the participants to apply the application to their respective tasks. The choice of projects varies regarding the number of classes and complexity, providing a heterogeneous study environment.

6.1.2 Instrumentation

As mentioned in Section 6.1.1, three specific projects have been selected to be used in this study. The tutorial project, designated as YarkAdminMS[1], consists of sixty-nine classes [55]. Task number one (Figure 6.1a) utilises a Warehouse System[2] project with a total of seventy-nine classes [55], while task number two (Figure 6.1b) is based on a Petclinic System[3] project with forty classes. All of these projects are implemented in Java since, currently, the application supports Java as the primary language for decomposition. It is worth noting that the projects used in the study were forked into the author's GitHub account to ensure permanent accessibility and availability for the study.


Task #1

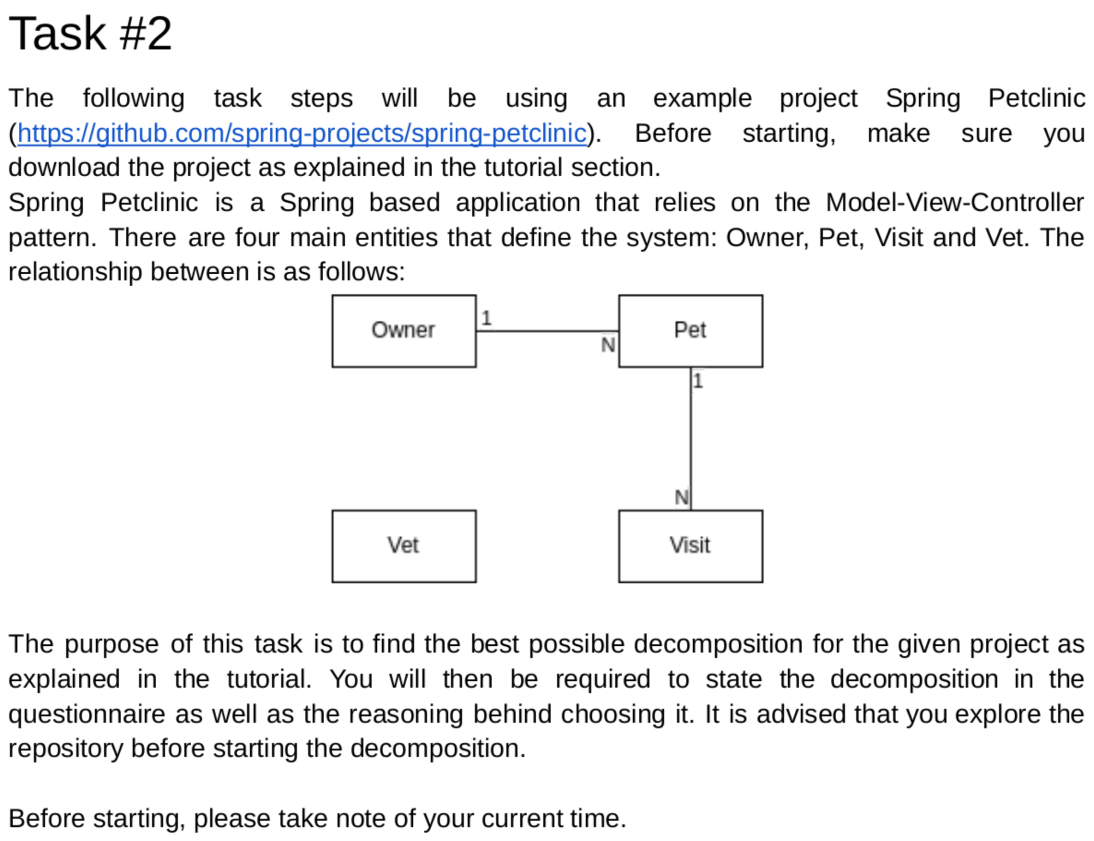
The following task steps will be using an example project Warehouse-system (https://github.com/bao17634/Warehouse-system). Before starting, make sure you download the project as explained in the tutorial section.
Warehouse-system is a Spring based application to manage the warehouse of a company. It relies on the Model-View-Controller pattern.

The purpose of this task is to find the best possible decomposition for the given project as explained in the tutorial. You will then be required to state the decomposition in the questionnaire as well as the reasoning behind choosing it. It is advised that you explore the repository before starting the decomposition.

Before starting, please take note of your current time.

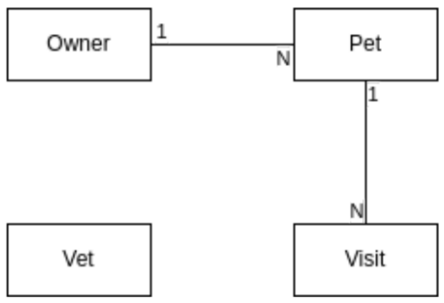
(a) Task 1 Content

[1]https://github.com/Guergeiro/Yark-AdminMS
[2]https://github.com/Guergeiro/Warehouse-system
[3]https://github.com/Guergeiro/spring-petclinic



Task #2

The following task steps will be using an example project Spring Petclinic (https://github.com/spring-projects/spring-petclinic). Before starting, make sure you download the project as explained in the tutorial section.
Spring Petclinic is a Spring based application that relies on the Model-View-Controller pattern. There are four main entities that define the system: Owner, Pet, Visit and Vet. The relationship between is as follows:



The purpose of this task is to find the best possible decomposition for the given project as explained in the tutorial. You will then be required to state the decomposition in the questionnaire as well as the reasoning behind choosing it. It is advised that you explore the repository before starting the decomposition.

Before starting, please take note of your current time.

(b) Task 2 Content

Figure 6.1: Tasks Contents

6.1.3 Pre Study

The purpose of an empirical study is to ensure that the study accurately reflects the functionality and performance of the application. It is crucial to minimise any potential issues arising from the study design or tasks that could affect participants' responses, ensuring that their feedback reflects the application's performance.

We employed an iterative approach of conducting mock studies with live support. Participants' experiences and difficulties were carefully observed and noted during these mock studies. Exactly two mock studies were conducted.

Based on the feedback received and the observations made during the mock studies, we made improvements and adjustments to the study design and tasks. The aim was to address and rectify any issues identified during the mock runs, ensuring that the final study provided participants with a smoother and more authentic experience.

To mitigate bias and prevent participants from being influenced by prior knowledge or experience with the study, individuals who participated in the mock runs were automatically excluded from the final study, helping to maintain the integrity of the study results by ensuring that the responses obtained are unbiased and representative of the application's performance.

6.1.4 Data Collection

The study is divided into eighth parts. The questionaire presented in Appendix A contains the following sections:

1. Small video[4] describing what microservices are.
2. Background questionaire about age, years of industry experience, etc. Figure A.1
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. Figure A.2
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. Figure A.3
6. System Usability Scale. Figure A.4
7. Raw Task Load Index. Figure A.5
8. An optional section where general feedback is appreciated. Figure A.6

6.2 Execution

The study was disseminated among professional colleagues, academic peers, and within the open-source communities to ensure diverse participants. The optimisation described in Section 6.1.3 allowed for asynchronous participation, making it more convenient for participants to engage in the study. Nevertheless, real-time assistance was provisioned in case of issues arising.

Participants were required to validate their knowledge of microservices. Initially, those with limited prior knowledge could watch a small video to provide a basic understanding of microservices concepts. Following this, participants completed a questionnaire that collected background information such as age, gender, industry and academic experience, educational level, and proficiency in programming languages.

A tutorial follows, demonstrating an example task and explaining the expected approach. Participants were then assigned their specific tasks and asked to replicate the steps demonstrated in the tutorial.

Afterwards, participants were asked to explore the application's multiple decompositions, compare them and use the entire feature list to identify the microservice they believed to be the best for their assigned task, as well as providing a rationale for choosing a particular decomposition. Additionally, participants filled out the System Usability Scale and Task Load Index questionnaires to assess the usability and perceived workload of the application.

[4] `https://youtu.be/lTAcCNbJ7KE`
[5] `https://youtu.be/WQAAU9vQddA`
[6] `https://youtu.be/3SVlrkiFNmU`

Finally, an optional section was included where participants were encouraged to provide additional feedback on the tool, allowing them to share any additional thoughts, suggestions, or comments they had.

6.3 Analysis

We have collected fifteen participants' answers, and by analysing the data from these participants, the findings will contribute to understanding the application's effectiveness, usability, and user experience, providing valuable information for further improvements and future research.

6.3.1 Subjects

During the study, basic information about each participant was collected, including their gender, age group, academic level, years of industry and academic experience, regularity of working with microservices, the duration of their experience with microservices, and the number of monoliths they have assisted in decomposing.

Notably, the collected data indicates that 100% of the participants in the study were male, with the age distribution observed in Figure 6.2.

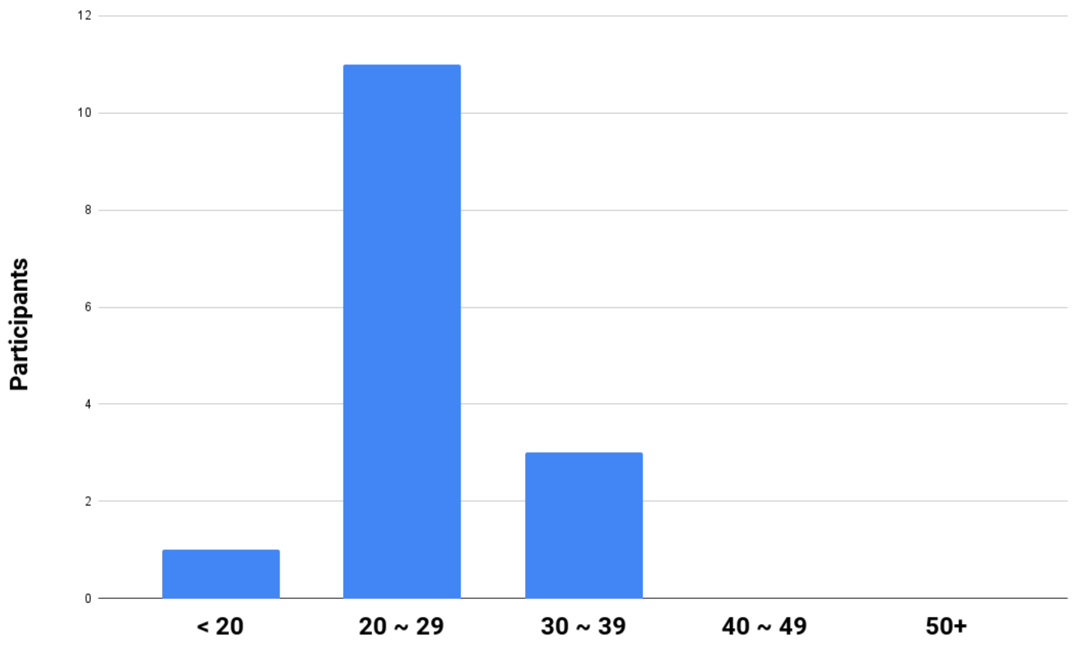


Figure 6.2: Subjects Age Distribution

The distribution of years of experience, considering both industry and academic realms, can be found in Figure 6.3, and by examining this distribution, it is possible to identify individuals with wide industry experience, academic expertise, or a combination of both. This differentiation is vital to understand the varied perspectives and knowledge participants bring to the study. It also aids in analysing how their backgrounds and experience may influence their perceptions, evaluations, and performance during the decomposition tasks.

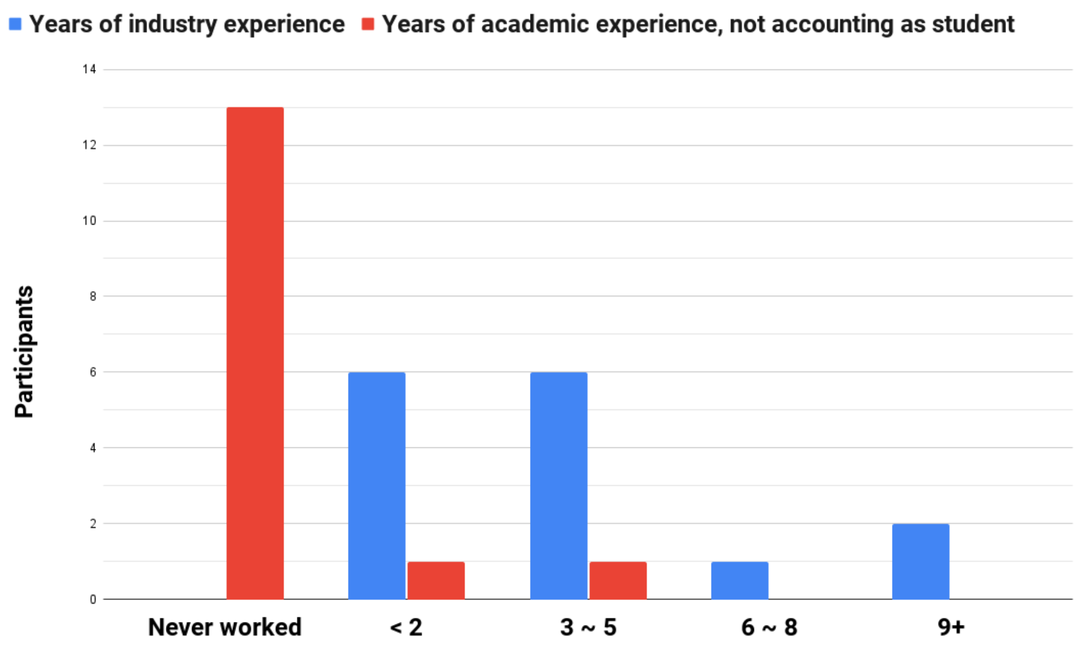


Figure 6.3: Years of Experience Distribution

The study categorised their level of experience with microservices into four categories: those who have never worked with microservices, those who have worked with microservices at least once per month, those who have worked with microservices at least once per week, and those who have worked with microservices daily.

Figure 6.4 provides a visualisation of the participants' frequency of working with microservices in their professional activities.

The study also considered the participants' experience working with microservices and the number of monolith decompositions they had undertaken, as presented in Figures 6.5 and 6.6.

6.3.2 System Usability Scale

The System Usability Scale (SUS) is a ten-question questionnaire, each with a five-point response scale ranging from Strongly Disagree to Strongly Agree. These questions are divided into two types:

1. Odd-numbered items, considered the positive statements (Figure 6.7a).

2. Even-numbered items, considered the negative statements (Figure 6.7b).

The division of the questions into positive and negative statements provides valuable insights into the overall usability of the application. Better user experience and usability mean the questionnaire tends to have higher evaluations (agreement) on positive statements. Conversely, lower

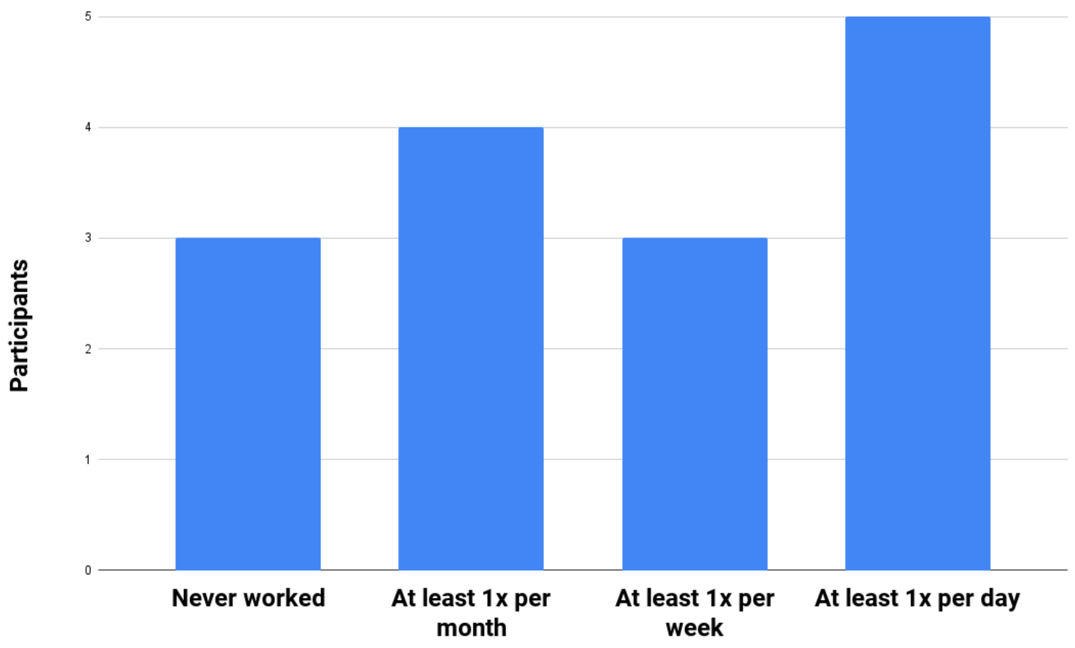


Figure 6.4: Frequency of work related with microservices

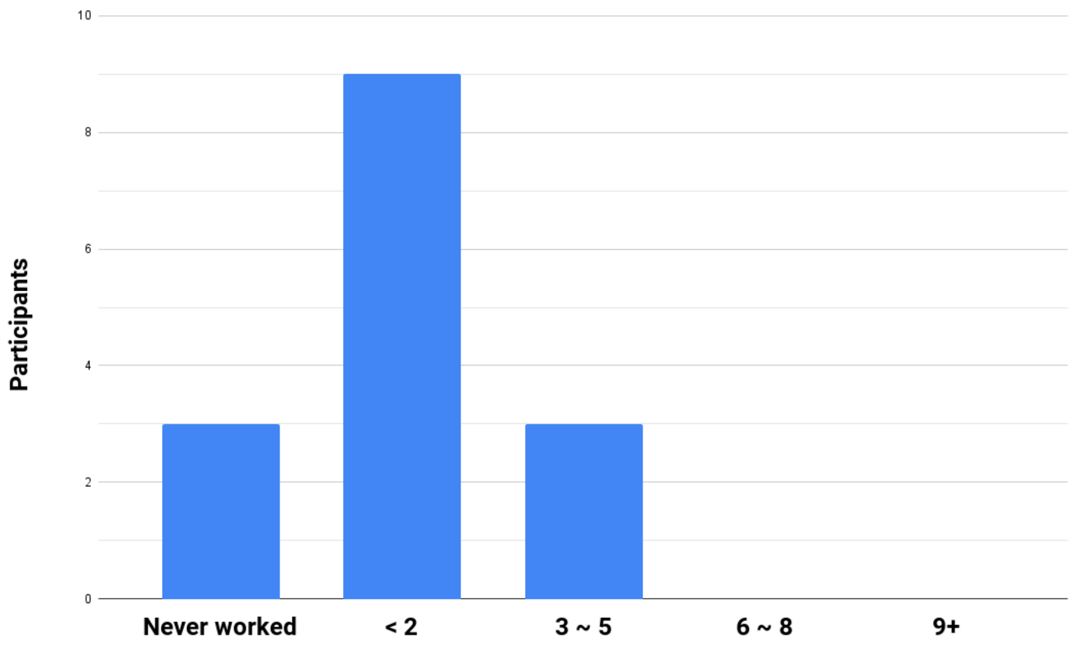


Figure 6.5: Years working with microservices

evaluations (disagreement) are desired on negative statements, indicating fewer usability issues or challenges. The aim is to achieve higher ratings on the positive statements and lower ratings on the negative statements, as this indicates a more favourable user perception of the application's

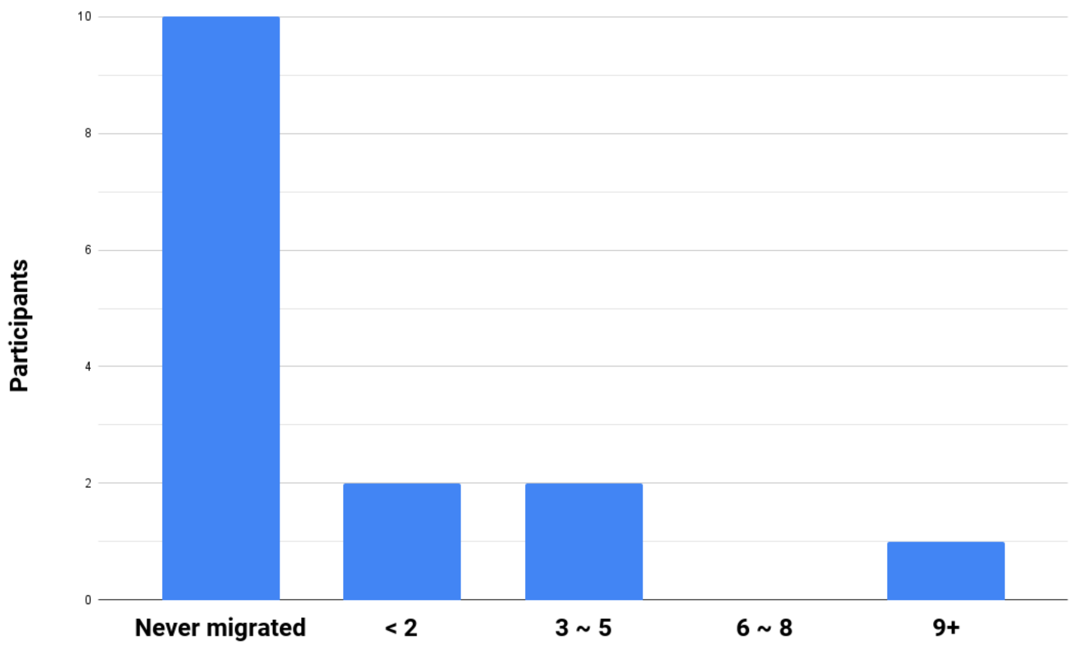


Figure 6.6: Amount of monoliths decomposed

usability.

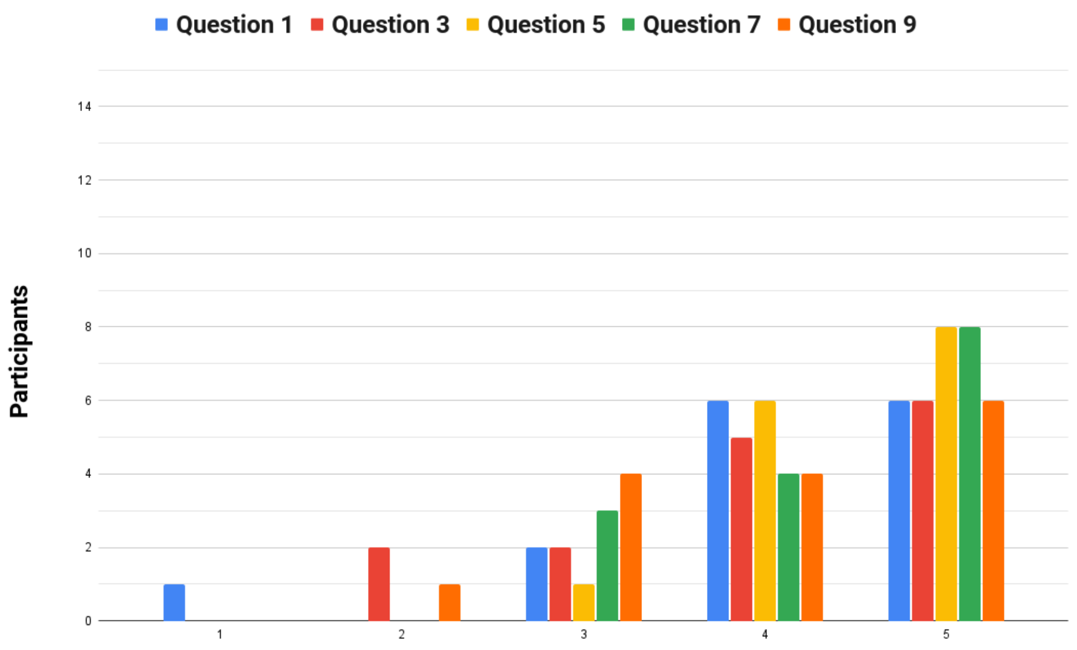
By analysing the results of the statements in Figure 6.7, it becomes evident that most participants provided higher evaluations for the positive statements. Approximately 79% of participants rated these positive statements at level four or higher, indicating a positive perception of the application's usability, while only a tiny proportion (7%) of participants selected level two or lower for the positive statements.

On the other hand, the negative statements received lower evaluations from the participants. Around 81% of participants rated these negative statements at level two or lower, suggesting a relatively low occurrence of usability issues, and a minimal percentage (5%) of participants provided evaluations at level four or higher for the negative statements.

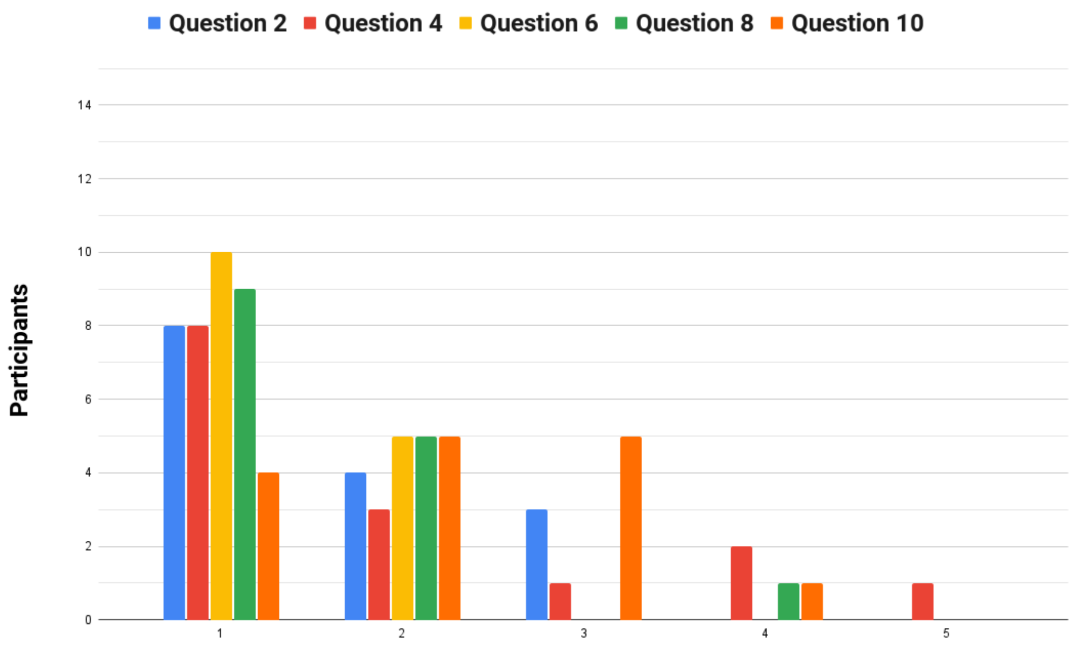
The average SUS Score for each participant was found to be 80.3 out of 100, indicating a generally positive perception of the application's usability. The median SUS Score was calculated to be 85 out of 100, further supporting the positive evaluation.

The quartiles, representing the distribution of the SUS Scores, are presented in Table 6.1. Additionally, a corresponding box plot graphic in Figure 6.8 visualises the distribution of the scores, providing a visual representation of the SUS Score data.

Participants with experience in microservices, those who have worked with microservices and have experience in decomposing monoliths into microservices, have higher evaluations in positive statements and lower evaluations in negative statements, as evidenced in Figures 6.9a and 6.9b, respectively.



(a) Positive Statements



(b) Negative Statements

Figure 6.7: Statements Results SUS

6.3.3 Raw-TLX

The National Aeronautics and Space Administration (NASA) developed NASA Task Load Index (NASA-TLX) to evaluate subjective workload. It employs a multi-dimensional rating ap-

Table 6.1: SUS Quartiles

Quartile	**Value**
0	42,5
1	65.75
2	85
3	93.75
4	97.5

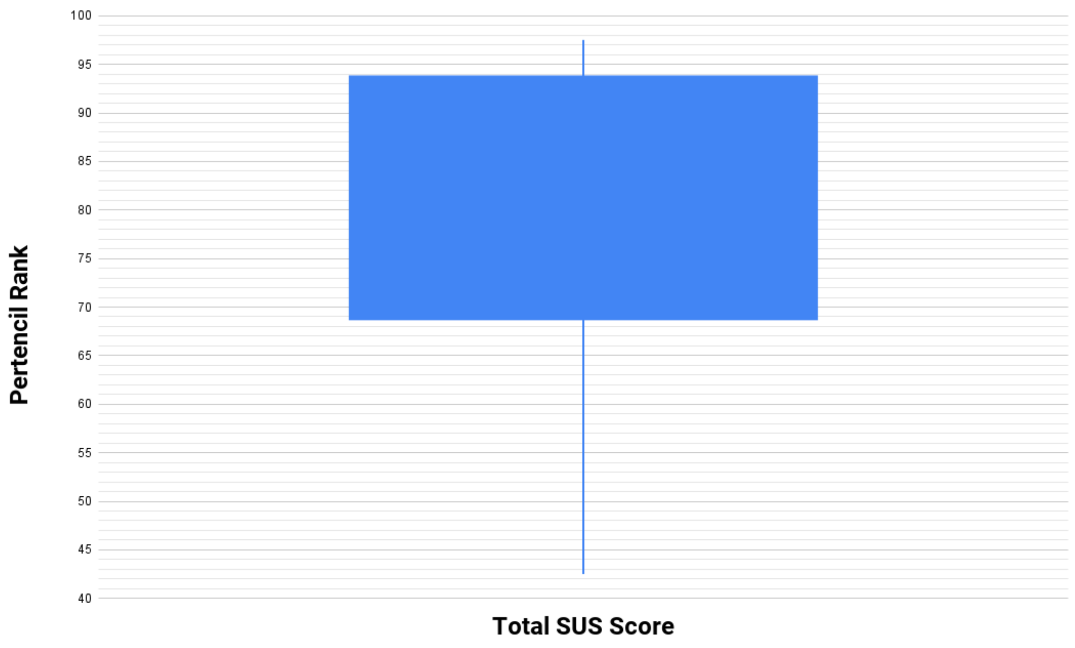
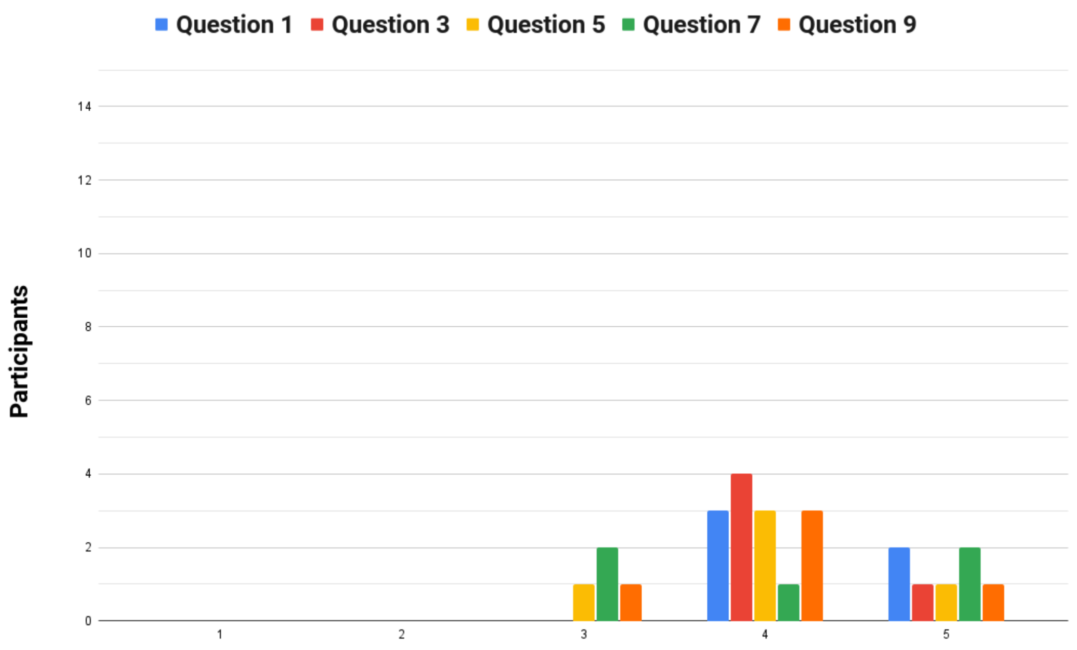


Figure 6.8: SUS Box Plot

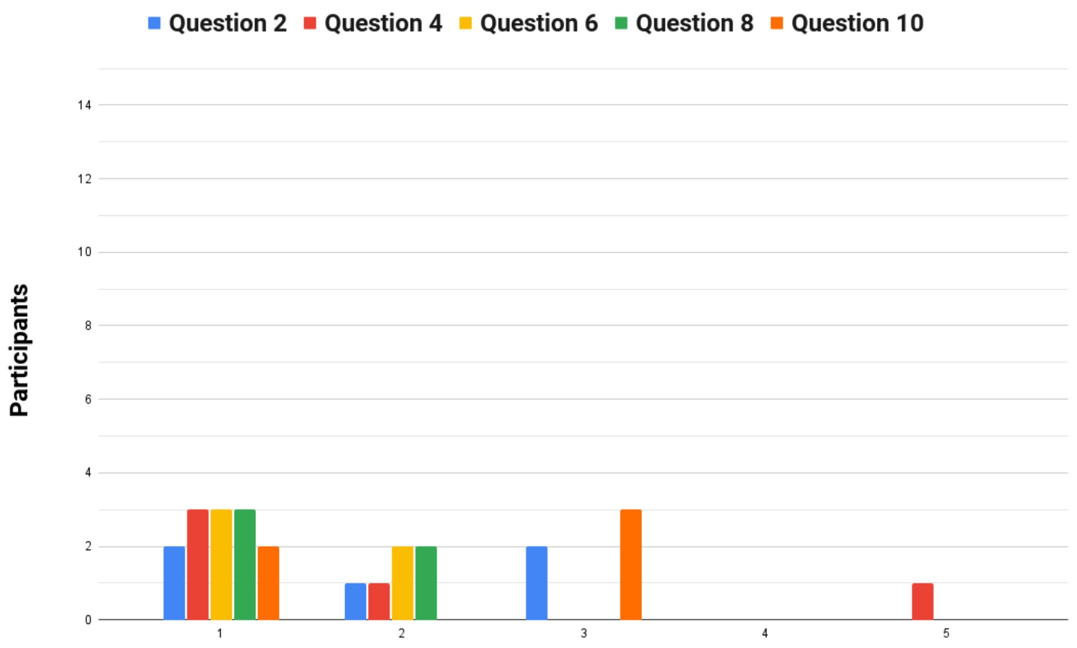
proach considering the weighted average ratings of six subscales. These subscales assess various aspects of workload experienced by participants, including subjective mental demand, physical demand, time demand, performance, effort, and frustration levels in a ten-point response scale [12].

In the study context, the Raw Task Load Index (Raw-TLX) is used as a simplified version of the NASA-TLX. One potential difference is that NASA-TLX incorporates a weighting process to adjust each scale to an individual's perception of workload [12]. In contrast, Raw-TLX omits this weighting process to maintain simplicity and ease of application [11].

In alignment with the use of Raw-TLX, the study also omitted one of the scales, Physical Demand, as it did not apply to the specific context of the task. The omission of this scale was due to its lack of relevance in assessing the workload associated with the decomposition tasks in the study. Not omitting the scale could skew the overall score since all scales have equal weight, that is, $\frac{6}{100}$. From all scales of Raw-TLX, only the Performance scale considers higher evaluations as positive. As for the other four scales, having a high score is considered harmful.



(a) Positive Statements



(b) Negative Statements

Figure 6.9: Experienced Participants Statements Results SUS

The Raw Task Load Index (Raw-TLX) was conducted to collect information on the effort required to complete the monolith decomposition tasks using the application. Then we calculated each category's average and median scores separately for each task. Task 1 and 2 results and scores are presented in Tables 6.2 and 6.3 as well as visualy in Figures 6.10 and 6.11, respectively.

Table 6.4 combines Task 1 and Task 2 results and the general average and median to provide a comprehensive overview of the perceived workload across both tasks. The visualisation in Figure 6.12 allows for comparing and analysing the workload distribution across the different categories for the overall study. Each line in all three graphs represents a different participant and the task they were each assigned.

Table 6.2: Task 1 Results Table

	Mental Demand	**Temporal Demand**	**Performance**	**Effort**	**Frustration**
Average	4.71	4.14	4.71	4.86	3.43
Median	5.0	4.0	4.0	5.0	3.0

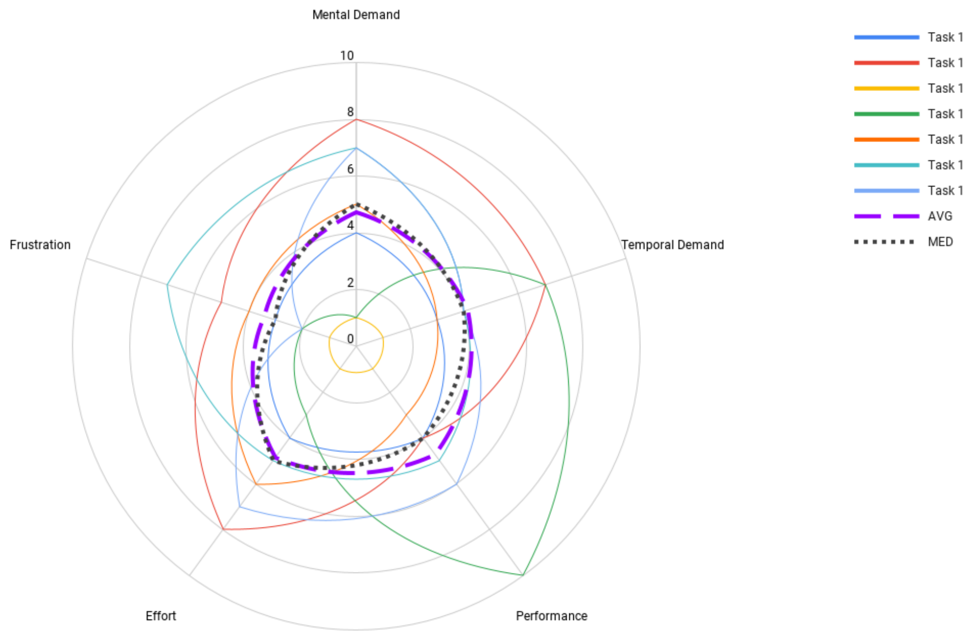


Figure 6.10: Task 1 Graph

Table 6.3: Task 2 Results Table

	Mental Demand	**Temporal Demand**	**Performance**	**Effort**	**Frustration**
Average	4.13	4.75	5.75	5.00	3.25
Median	4.0	5.0	7.0	5.5	2.5

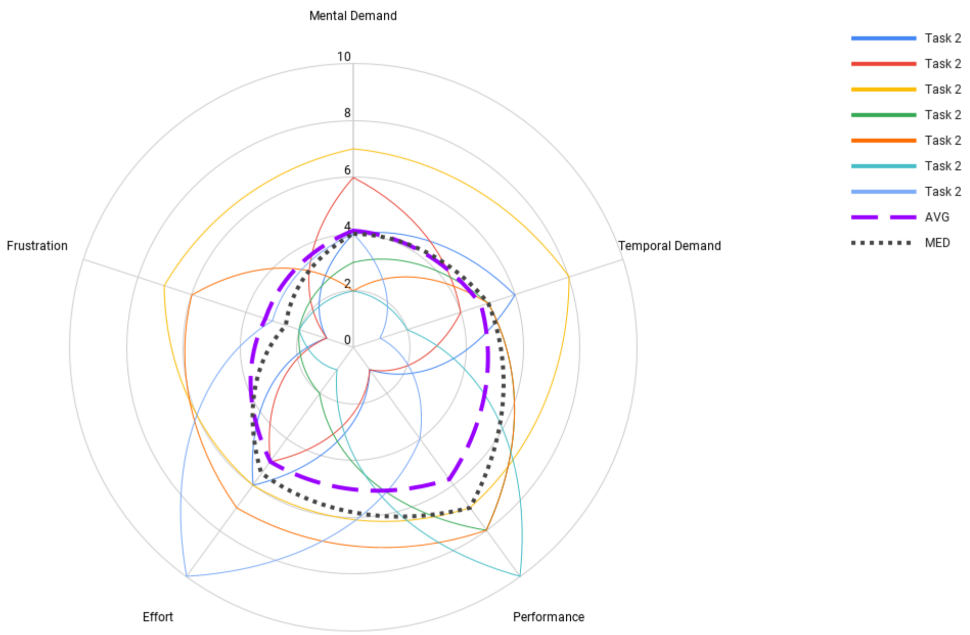


Figure 6.11: Task 2

Table 6.4: Tasks Results Table

	Mental Demand	**Temporal Demand**	**Perfor-mance**	**Effort**	**Frustration**
Average	4.40	4.47	5.27	4.93	3.33
Median	4.0	4.0	5.0	5.0	3.0

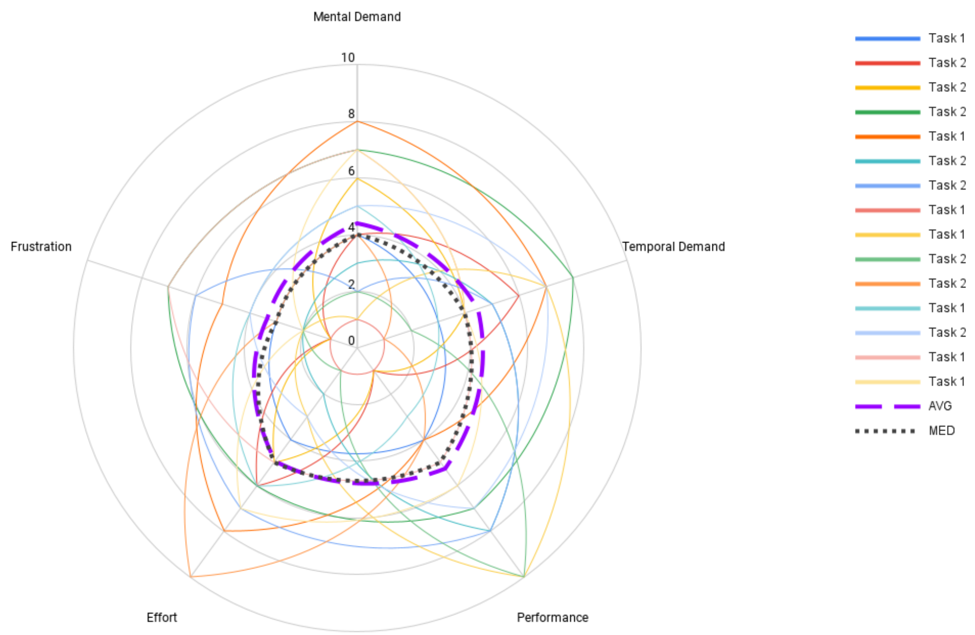


Figure 6.12: Tasks Results

Chapter 7

Discussion

This chapter discusses the results from each evaluation metric, including the SUS and Raw-TLX shown in Section 6.3, their implications and some questions associated with each specific evaluation question.

Additionally, the chapter addresses the threats to the validity of the study results and includes an exploration of potential limitations and biases that may have influenced the findings.

7.1 Results evaluation

Can the application exhibit a good usability for decomposing?

The application demonstrates good usability based on the results and discussions presented in Section 6.3.2. The SUS scores, as indicated by the average of 80.3 and median of 85, reflect positive evaluations from the participants. The SUS is a widely used questionnaire to measure usability, and the high scores suggest that the participants found the application user-friendly, intuitive, and easy to use.

Furthermore, the analysis of the SUS questionnaire's positive and negative statements reveals that the participants generally agreed with the positive statements, indicating their satisfaction with the application's usability. Conversely, they disagreed with the negative statements, suggesting they did not encounter significant usability issues or challenges while using the application.

The positive evaluations and feedback from the participants regarding usability provide strong evidence that the application is well designed and effectively supports the monolith decomposition tasks.

Does the application provide a low workload?

While looking at results obtained in Raw-TLX, Section 6.3.3, the participants rated the workload according to five categories: Mental Demand, Temporal Demand, Performance, Effort and Frustration.

The usage of our application did not frustrate the participants, having an average and median frustration of 3.33 and 3 points, respectively.

In regards to performance, the results are more balanced. Subjects did not feel that the tool would have improved their decompositions, scoring an average of 5.27 and a median of 5. This result contradicts some results of Section 6.3.2 and could be related to the fact that this question's positive answer shifted to the left instead of keeping on the right; that is, higher evaluations are considered negative instead of positive. In fact, 27% of the participants that scored a SUS greater or equal to 70 answered as a 7 out of 10 regarding their success on the decomposition, which could indicate participants were distracted while filling out this part of the questionnaire. This can be visualised in Figure 7.1 (for a better visualisation, Raw-TLX Performance was scaled to match SUS scale).

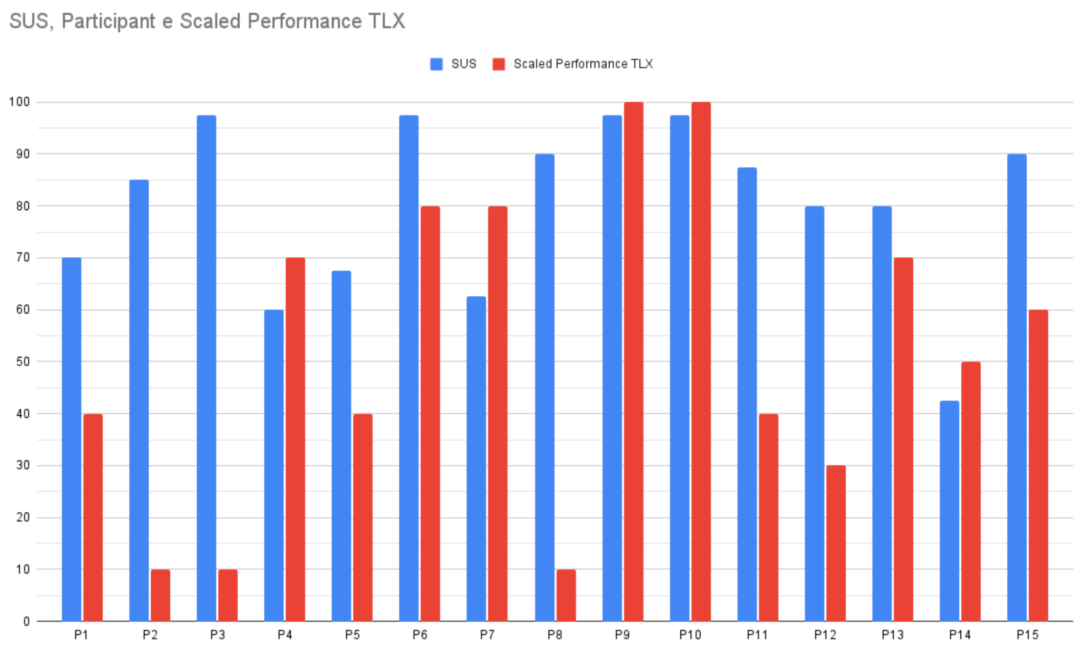


Figure 7.1: SUS vs Raw-TLX Performance

As for effort, subjects scored an average of 4.93 and a median of 5, which means that the participants felt indifferent to how hard they had to work to obtain their performance results.

The two demand levels score similarly, with 4 as the median and 4.40 4.47 for mental and temporal demand, respectively. This means subjects did not find the task mentally demanding or felt time pressure.

Overall, the application provided a mixed workload performance.

Despite the highly positive SUS scores, the Raw-TLX ratings were relatively underwhelming. A possible explanation for this discrepancy is the absence of contextual information provided during the task. Users relied solely on the application to assist them in the decomposition process and encountered particular challenges and frustrations, which may have resulted in slightly negative sentiments. It would be beneficial to introduce a two-step approach to address this issue. Initially, users could engage in a decomposition task without utilising the application, followed by a subsequent task, using a different project, involving employing the tool. This sequential methodology has the potential to yield improved outcomes. Also, since decomposing a monolith into microservices is a difficult task, not having something for participants to compare the results to, may affect the perception on how much the application helps them.

7.2 Threats to validity

The study aimed to demonstrate that using the application for performing monolith decomposition tasks is better than not using it. Throughout the study, various threats to validity were identified and categorised into two main categories according to Wohlin et al [26]: internal validity, conclusion validity, construct validity, and external validity.

7.2.1 Internal validity

A potential threat to the study is the absence of a control or test group. Without a comparison group, it becomes challenging to establish a clear causal relationship between the treatment (use of the application) and the outcome (usability and workload evaluations). The lack of a control group limits our ability to determine whether any observed changes or improvements in usability and workload are specifically attributable to the use of the application or could be influenced by other factors. We advise that this study ins performed with a control group.

7.2.2 Conclusion validity

As outlined in Section 6.3, the study was conducted with a relatively small sample size of fifteen participants, which may lead to reduced statistical power. To help mitigate this, we invited participants with diverse backgrounds, including varying levels of industry expertise, academic expertise, and familiarity with microservices.

Furthermore, the asynchronous nature of the experiment introduces a potential challenge in controlling the participants' environment and level of focus during the tasks. To mitigate this issue, the study aimed to provide a smooth and user-friendly experience, as discussed in Section Section 6.1.3.

There is also the argument that drawing general conclusions from only two tasks presents a limitation in terms of generalizability. It is hard to balance the subject's interest in the study while trying to extract as much data from them as possible. However, to address this limitation, two projects with different sizes and domains were included in the study (as explained in Section 6.1.2).

7.2.3 Construction validity

One threat is the degree to which the study's measurements accurately capture the intended constructs or concepts using the correct metrics to evaluate results. In our case, we wanted to measure the usability and workload of the application developed, and measures such as the System Usability Scale (SUS) and Raw Task Load Index (Raw-TLX) were employed to assess them, as mentioned in Sections 6.3.2 and 6.3.3.

7.2.4 External validity

One potential threat is the pre-test-treatment interaction, where interaction with participants before the study may sensitise them to aspects of the study. The interaction with the subjects before participating in the study was minimal and focused on providing the required instructions and support, essential in maintaining the study's validity and scores.

Chapter 8

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 [46]. Microservices are especially well-suited for distributed, cloud-based systems, where they may benefit from the cloud's flexibility and scalability [36].

We performed a literature review on the subject of migratin 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.

Based on the literature review and analysis, we found limited tools to aid the migration process from monoliths to microservices. Seven free and open-source tools were identified, each with varying levels of completeness.

In terms of an application that aggregates these tools to assist architects, engineers, and developers in microservice migration, we have yet to find an existing application to fulfil this role. Therefore, the dissertation proposes a solution to this gap by providing a user-friendly and consolidated platform for microservice migration activities.

The application 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. Each component of the application architecture is 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.

Decomposing monolithic architectures into microservices primarily involves labour-intensive and time-consuming manual work, coupled with the difficulty of identifying functional units due to the complex analysis required across multiple dimensions of the software architecture. However, the application proposed in this study's experiment favourably impacted the participants.

As evidenced by the positive evaluations and high scores obtained in the SUS, participants perceived the application as user-friendly, intuitive, and easy to navigate.

Regarding the workload, the application did not yield significant effects on the decomposition process of monolithic architectures, as the performance results were mixed.

8.1 Future Work

Given that software development is never finished, there are two areas for future improvements: enhancements to the application and refinements in the evaluation process.

In terms of application improvements, valuable insights were obtained through the feedback questionnaire (depicted in Figure A.6), leading to the identification of several potential future enhancements:

- **Enhancing metadata descriptions for improved comparisons:** The application only presents metadata metrics without accompanying explanations for each metadata. It would be beneficial to provide descriptions for metadata attributes to facilitate better understanding and comparisons. One participant stated *"Add explanation for terms like Modularity and Resolution"*, while other *" The modularity and resolution could be explained a bit further"*.

- **Implementing decomposition tracking features:** Introducing functionality to track decompositions would enable users to manage and differentiate between various decompositions, like selecting favourites or marking decompositions as already viewed. Such features assist users in discarding irrelevant decompositions while retaining others for future comparisons. As suggested by one participant *"When selecting decomposition results for comparison, having an option to mark each decomposition result as viewed or analyzed would help the use"*, and another stated *"Save decomposition id, like a favorites menu"*.

- **Establishing bidirectional relationships between microservices and modules:** When focusing on a specific microservice, the application displays the list of modules that compose it. However, this relationship is unidirectional. It would be advantageous to establish a two-way association, enabling users to select a module and visualise all the corresponding microservices across different decompositions. The suggestion is *"Have a list of classes* *and when selected show in the graph in which service it belongs to (even when comparing multiple decompositions)"*

- **Incorporating refactoring tools:** Providing a preview step for tools before the refactoring. At the moment, there are some tools that already do code refactor [57], but do not provide a way of previewing it. It could be benefitial to "inject" this solution in the middle of the process, going from decomposition, into visualisation, into refactoring.

In terms of evaluation improvements, some considerations should be taken into account for future studies, specifically:

- **Expanding the respondent pool:** Increasing the number of participants in evaluations can enhance the reliability and generalizability of the findings. A more extensive and diverse group of respondents can capture a more comprehensive range of perspectives and experiences, leading to more robust conclusions.
- **Incorporating a comparative approach:** Participants should be engaged in both tasks, one without utilising the application and only using the underlying tools and another while using the application. By comparing the outcomes and experiences between these two scenarios, the added value and impact of the application can be assessed more accurately. Additionally, employing different tasks with similar-sized projects can help evaluate the application's performance across various contexts and validate its usefulness in different scenarios.
- **Broadening the task pool and project size range:** Diversifying the available tasks and expanding the range of project sizes used in evaluations contribute to a more comprehensive assessment of the application.

References

[1] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Migrating to cloud-native architectures using microservices: an experience report. In *Advances in Service-Oriented and Cloud Computing: Workshops of ESOCC 2015, Taormina, Italy, September 15-17, 2015, Revised Selected Papers 4*, pages 201–215. Springer, 2016.

[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] Kevin Brennan et al. *A Guide to the Business Analysis Body of Knowledger*. Iiba, 2009.

[4] John Brooke et al. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[5] Tomas Cerny, Amr S Abdelfattah, Vincent Bushong, Abdullah Al Maruf, and Davide Taibi. Microservice architecture reconstruction and visualization techniques: A review. In *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 39–48. IEEE, 2022.

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

[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] Sandra G Hart. Nasa-task load index (nasa-tlx); 20 years later. In *Proceedings of the human factors and ergonomics society annual meeting*, volume 50, pages 904–908. Sage publications Sage CA: Los Angeles, CA, 2006.

[12] Sandra G Hart and Lowell E Staveland. Development of nasa-tlx (task load index): Results of empirical and theoretical research. In *Advances in psychology*, volume 52, pages 139–183. Elsevier, 1988.

[13] Manabu Kamimura, Keisuke Yano, Tomomi Hatano, and Akihiko Matsuo. Extracting candidates of microservices from monolithic application code. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 571–580. IEEE, 2018.

[14] 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.

[15] 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.

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

[17] Daniel Moody. The “physics” of notations: toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on software engineering*, 35(6):756–779, 2009.

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

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

[20] Sam Newman. *Building microservices*. " O’Reilly Media, Inc.", 2021.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] Karl Wiegers and Joy Beatty. *Software requirements*. Pearson Education, 2013.

[26] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

Webliography

[27] Desktop operating system market share worldwide. https://gs.statcounter.com/os-market-share/desktop/worldwide/. Last accessed 7 January 2023.

[28] Astro. Why astro? https://docs.astro.build/en/concepts/why-astro/. Last accessed 15 June 2023.

[29] Ryan Carniato. Introducing the solidjs ui library. https://dev.to/ryansolid/introducing-the-solidjs-ui-library-4mck. Last accessed 26 June 2023.

[30] Ryan Dahl. Original node.js presentation. https://youtu.be/ztspvPYybIY, 2009. Last accessed 15 June 2023.

[31] Docker. The industry-leading container runtime. https://www.docker.com/products/container-runtime/. Last accessed 15 June 2023.

[32] Docker. What is a container? https://www.docker.com/resources/what-container/. Last accessed 15 June 2023.

[33] Mozilla Foundation. The event loop. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Event_loop. Last accessed 15 June 2023.

[34] OpenJS Foundation. The event loop. https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick#what-is-the-event-loop. Last accessed 15 June 2023.

[35] Martin Fowler. Richardson maturity model. https://martinfowler.com/articles/richardsonMaturityModel.html. Last accessed 15 June 2023.

[36] Martin Fowler. Microservice prerequisites. https://martinfowler.com/bliki/MicroservicePrerequisites.html, 2014. Last accessed 4 January 2023.

[37] Martin Fowler. Microservices. https://martinfowler.com/articles/microservices.html, 2014. Last accessed 4 January 2023.

[38] Martin Fowler. Microservice trade-offs. https://martinfowler.com/articles/microservice-trade-offs.html, 2015. Last accessed 4 January 2023.

[39] Mike Krieger. Storing hundreds of millions of simple key-value pairs in redis. https://instagram-engineering.com/storing-hundreds-of-millions-of-simple-key-value-pairs-in-redis-1091ae80f74
Last accessed 26 June 2023.

[40] Meta. Describing the ui. https://react.dev/learn/describing-the-ui. Last accessed 26 June 2023.

[41] Microsoft. Typescript for the new programmer. https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html. Last accessed 15 June 2023.

[42] Kamil Mysliwiec. Nestjs documentation. https://docs.nestjs.com/. Last accessed 15 June 2023.

[43] Redis. Redis data types. https://redis.io/docs/data-types/. Last accessed 15 June 2023.

[44] Redis. Using redis. https://redis.io/docs/manual/. Last accessed 15 June 2023.

[45] Redis. Where does the name "redis" come from? https://redis.io/docs/getting-started/faq/#where-does-the-name-redis-come-from. Last accessed 15 June 2023.

[46] Chris Richardson. Microservice architecture. https://microservices.io/patterns/microservices.html. Last accessed 4 January 2023.

[47] Chris Richardson. Who is using microservices. https://microservices.io/articles/whoisusingmicroservices.html. Last accessed 4 January 2023.

[48] Jonathan Turner. Announcing typescript 1.0. https://devblogs.microsoft.com/typescript/announcing-typescript-1-0/, 2014. Last accessed 15 June 2023.

[49] VentureBeat. A conversation with salvatore sanfilippo, creator of the open-source database redis. https://venturebeat.com/dev/redis-creator/. Last accessed 26 June 2023.

Systematic Literature Review References

[50] 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.

[51] 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.

[52] 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.

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

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

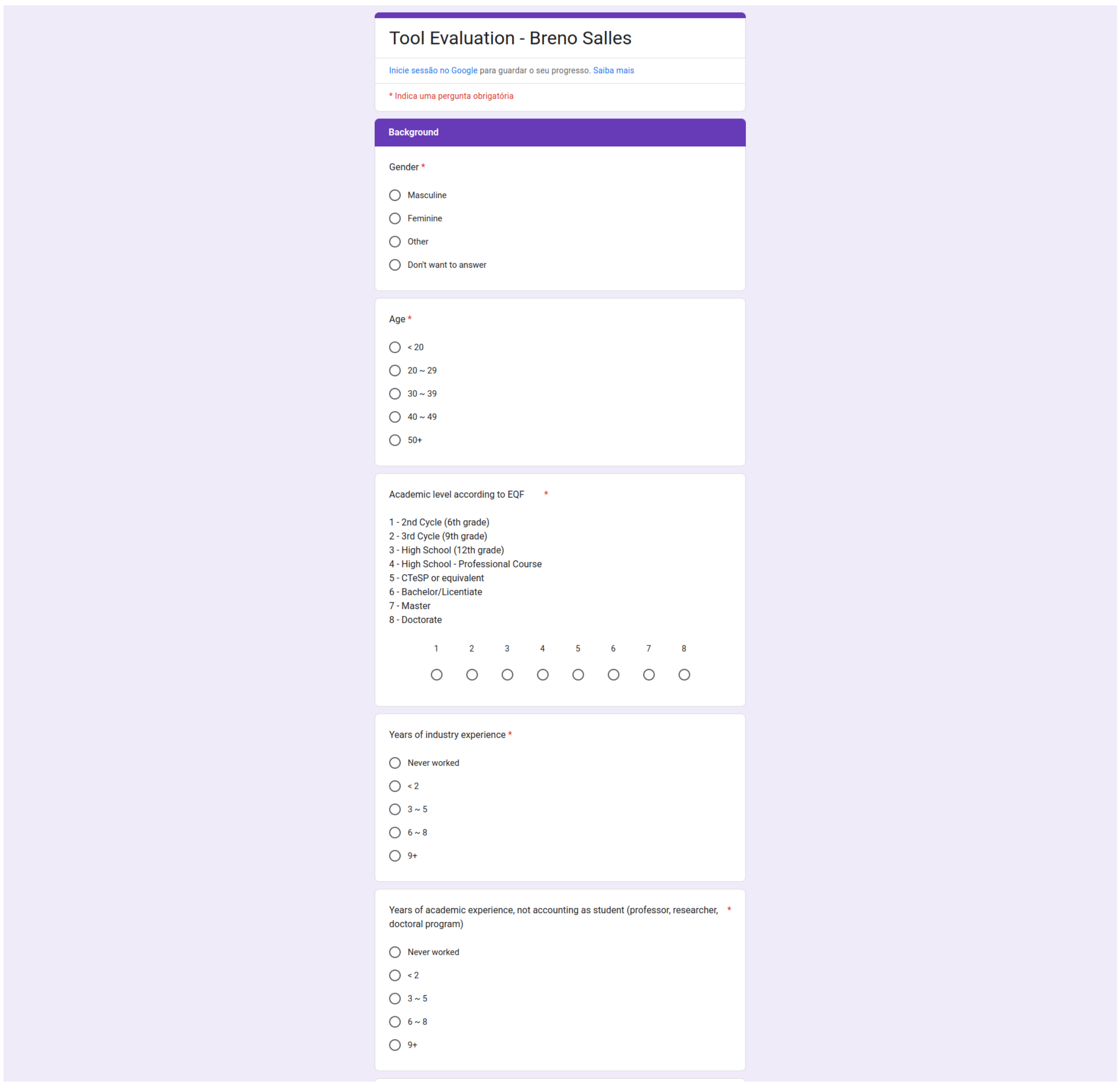
[68] Xiaoxiao Sun, Salamat Boranbaev, Shicong Han, Huanqiang Wang, and Dongjin Yu. Expert system for automatic microservices identification using api similarity graph. *Expert Systems*, page e13158, 2022.

[69] 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.

[70] 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.

Appendix A

Questionaire



Tool Evaluation - Breno Salles

Inicie sessão no Google para guardar o seu progresso. Saiba mais

* Indica uma pergunta obrigatória

Background

Gender *

- ◯ Masculine
- ◯ Feminine
- ◯ Other
- ◯ Don't want to answer

Age *

- ◯ < 20
- ◯ 20 ~ 29
- ◯ 30 ~ 39
- ◯ 40 ~ 49
- ◯ 50+

Academic level according to EQF *

1 - 2nd Cycle (6th grade)
2 - 3rd Cycle (9th grade)
3 - High School (12th grade)
4 - High School - Professional Course
5 - CTeSP or equivalent
6 - Bachelor/Licentiate
7 - Master
8 - Doctorate

1	2	3	4	5	6	7	8
◯	◯	◯	◯	◯	◯	◯	◯

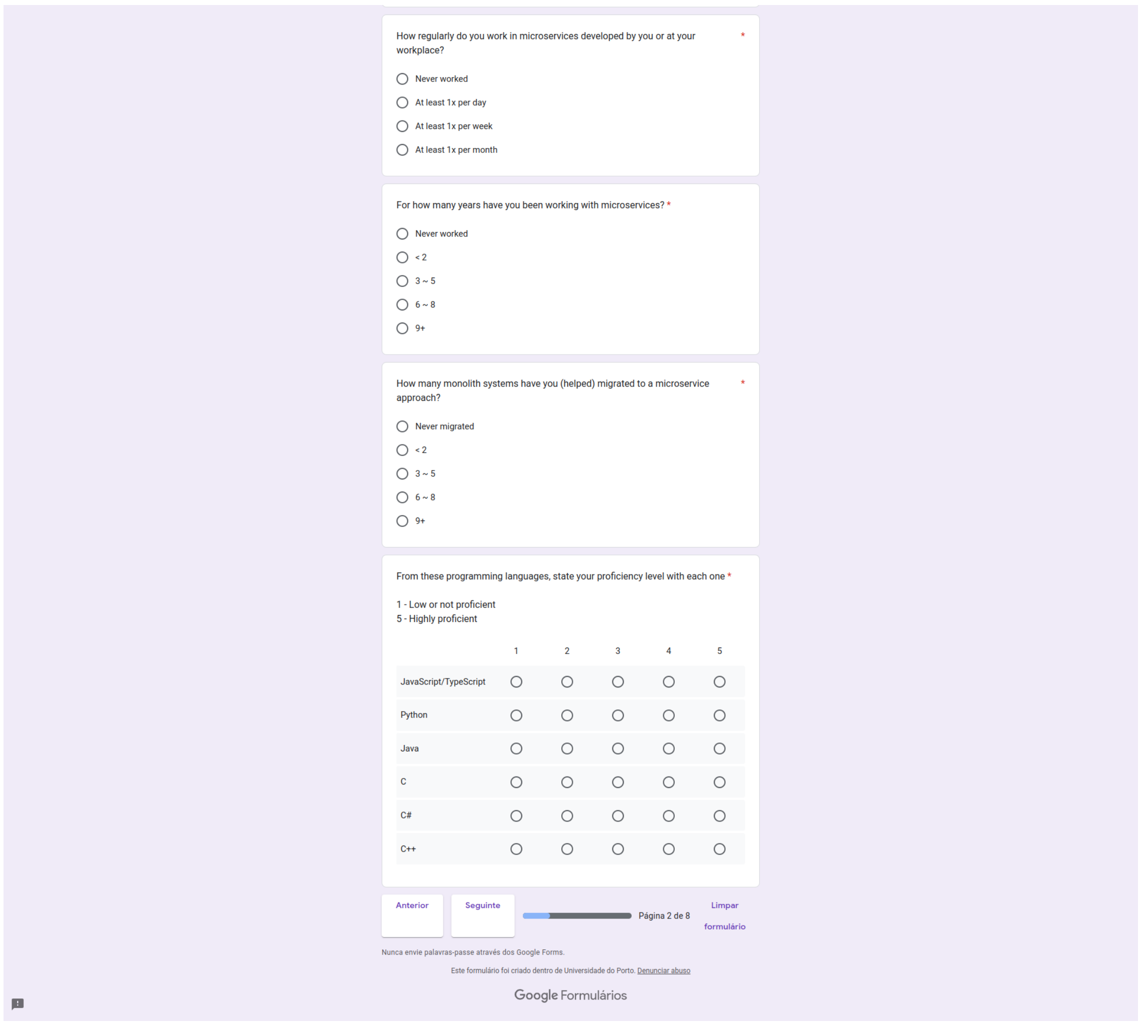
Years of industry experience *

- ◯ Never worked
- ◯ < 2
- ◯ 3 ~ 5
- ◯ 6 ~ 8
- ◯ 9+

Years of academic experience, not accounting as student (professor, researcher, doctoral program) *

- ◯ Never worked
- ◯ < 2
- ◯ 3 ~ 5
- ◯ 6 ~ 8
- ◯ 9+

Figure A.1: Background Questionaire



How regularly do you work in microservices developed by you or at your workplace? *

- ○ Never worked
- ○ At least 1x per day
- ○ At least 1x per week
- ○ At least 1x per month

For how many years have you been working with microservices? *

- ○ Never worked
- ○ < 2
- ○ 3 ~ 5
- ○ 6 ~ 8
- ○ 9+

How many monolith systems have you (helped) migrated to a microservice approach? *

- ○ Never migrated
- ○ < 2
- ○ 3 ~ 5
- ○ 6 ~ 8
- ○ 9+

From these programming languages, state your proficiency level with each one *

1 - Low or not proficient
5 - Highly proficient

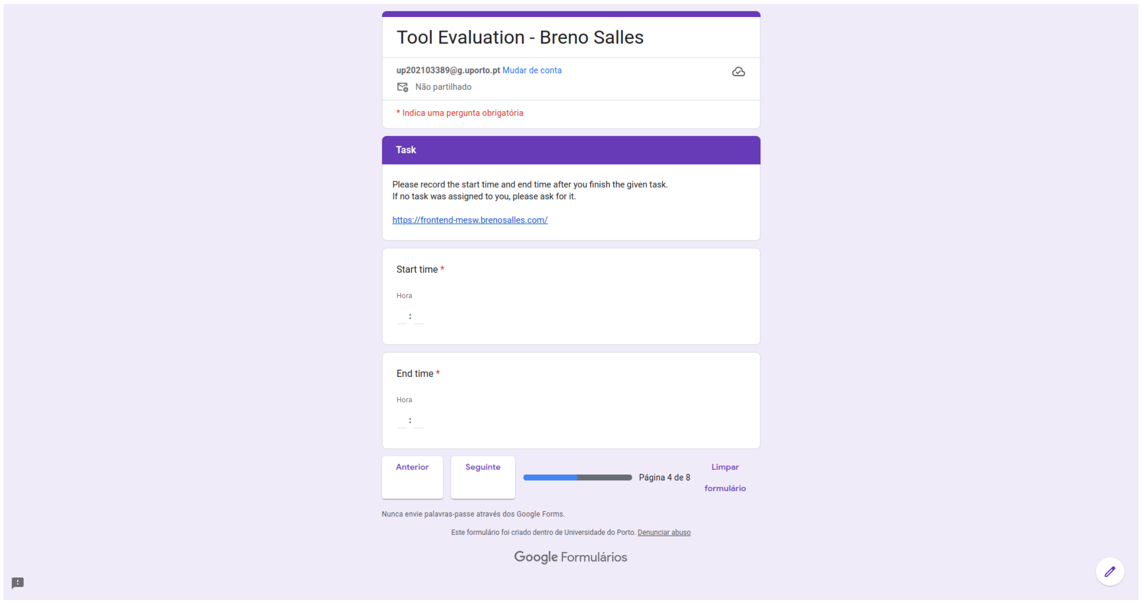
	1	2	3	4	5
JavaScript/TypeScript	○	○	○	○	○
Python	○	○	○	○	○
Java	○	○	○	○	○
C	○	○	○	○	○
C#	○	○	○	○	○
C++	○	○	○	○	○

Nunca envie palavras-passe através dos Google Forms.

Este formulário foi criado dentro de Universidade do Porto. Denunciar abuso

Google Formulários

Figure A.1: Background Questionaire (cont.)


Tool Evaluation - Breno Salles

up202103389@g.uporto.pt Mudar de conta

Não partilhado

* Indica uma pergunta obrigatória

Task

Please record the start time and end time after you finish the given task.
If no task was assigned to you, please ask for it.

https://frontend-mesw.brenosalles.com/

Start time *

Hora

__ : __

End time *

Hora

__ : __

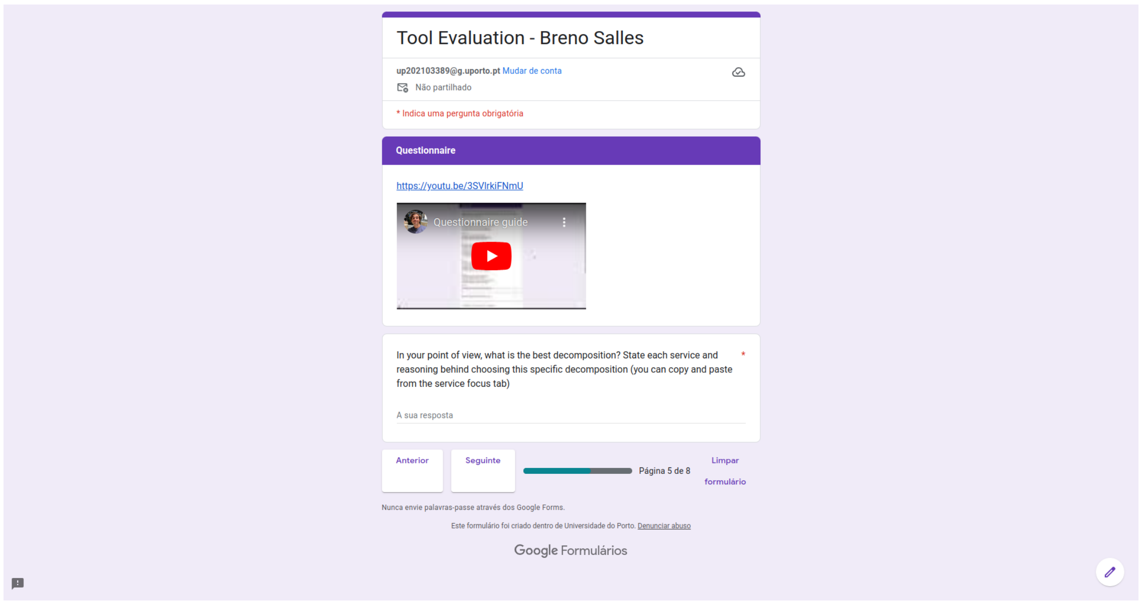
Anterior Seguinte Página 4 de 8 Limpar formulário

Nunca envie palavras-passe através dos Google Forms.

Este formulário foi criado dentro de Universidade do Porto. Denunciar abuso

Google Formulários

Figure A.2: Task Questionaire


Tool Evaluation - Breno Salles

up202103389@g.uporto.pt Mudar de conta

Não partilhado

* Indica uma pergunta obrigatória

Questionnaire

https://youtu.be/3SVIrkiFNmU

In your point of view, what is the best decomposition? State each service and reasoning behind choosing this specific decomposition (you can copy and paste from the service focus tab) *

A sua resposta

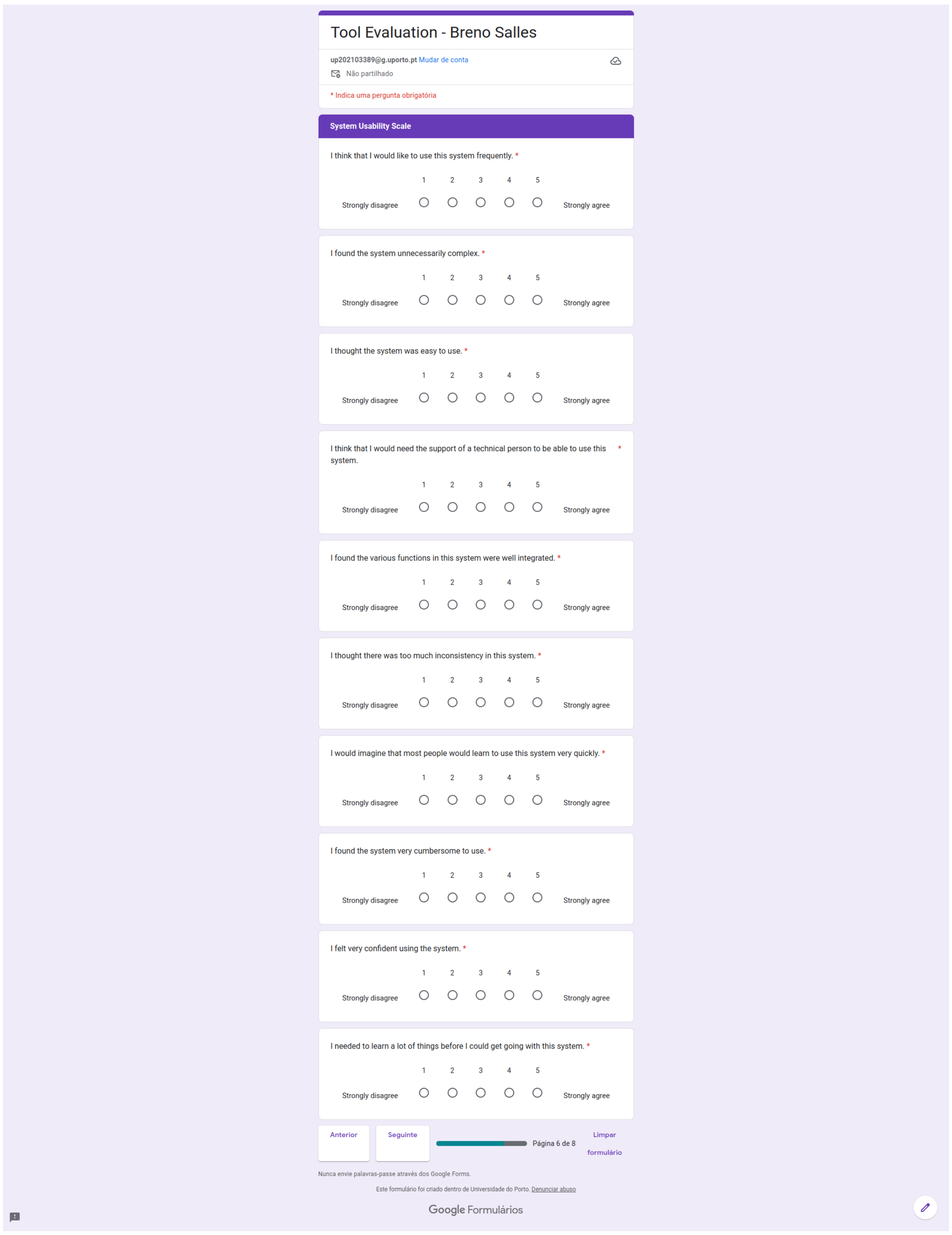
Anterior Seguinte Página 5 de 8 Limpar formulário

Nunca envie palavras-passe através dos Google Forms.

Este formulário foi criado dentro de Universidade do Porto. Denunciar abuso

Google Formulários

Figure A.3: Answer Questionaire



Tool Evaluation - Breno Salles

up202103389@g.uporto.pt Mudar de conta

Não partilhado

* Indica uma pergunta obrigatória

System Usability Scale

I think that I would like to use this system frequently. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I found the system unnecessarily complex. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I thought the system was easy to use. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I think that I would need the support of a technical person to be able to use this system. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I found the various functions in this system were well integrated. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I thought there was too much inconsistency in this system. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I would imagine that most people would learn to use this system very quickly. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I found the system very cumbersome to use. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I felt very confident using the system. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I needed to learn a lot of things before I could get going with this system. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

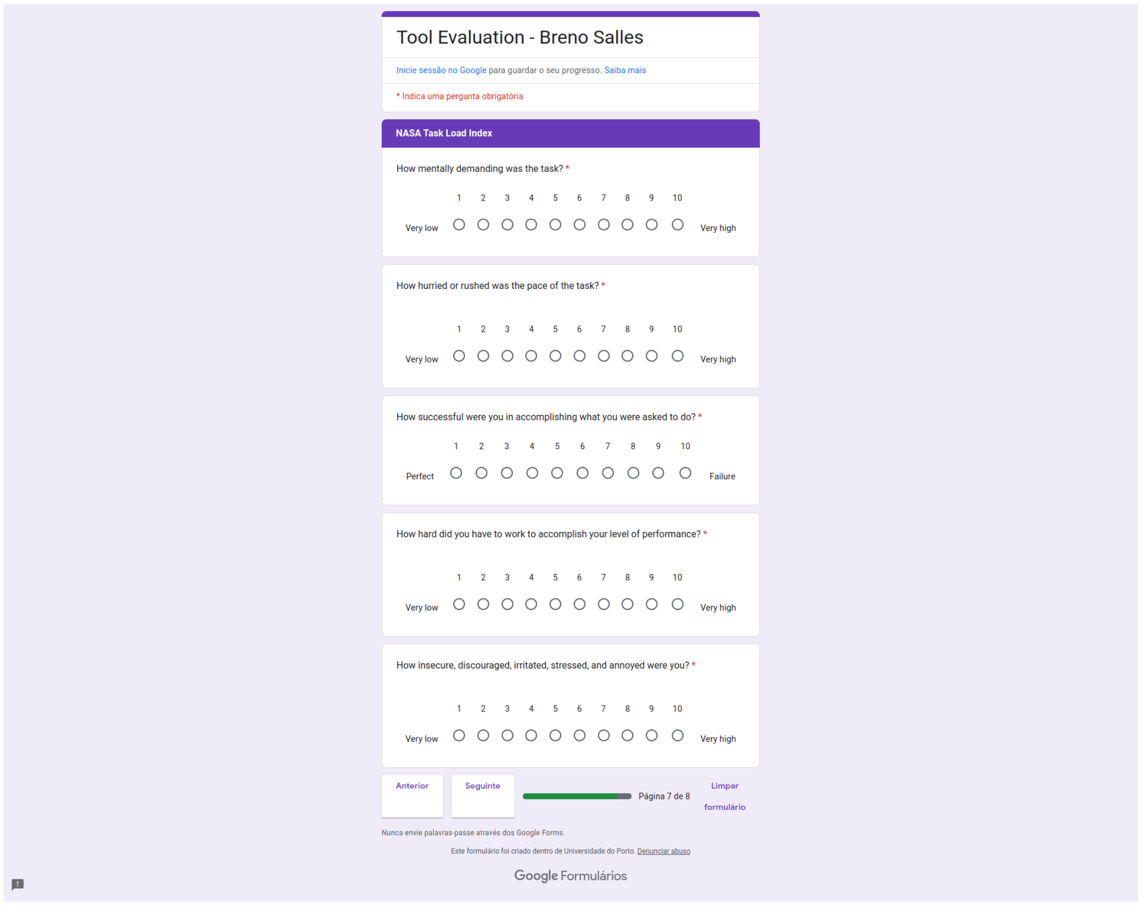
Anterior Seguinte Página 6 de 8 Limpar formulário

Nunca envie palavras-passe através dos Google Forms.

Este formulário foi criado dentro de Universidade do Porto. Denunciar abuso

Google Formulários

Figure A.4: SUS Questionaire



Tool Evaluation - Breno Salles

Inicie sessão no Google para guardar o seu progresso. Saiba mais

* Indica uma pergunta obrigatória

NASA Task Load Index

How mentally demanding was the task? *

	1	2	3	4	5	6	7	8	9	10	
Very low	○	○	○	○	○	○	○	○	○	○	Very high

How hurried or rushed was the pace of the task? *

	1	2	3	4	5	6	7	8	9	10	
Very low	○	○	○	○	○	○	○	○	○	○	Very high

How successful were you in accomplishing what you were asked to do? *

	1	2	3	4	5	6	7	8	9	10	
Perfect	○	○	○	○	○	○	○	○	○	○	Failure

How hard did you have to work to accomplish your level of performance? *

	1	2	3	4	5	6	7	8	9	10	
Very low	○	○	○	○	○	○	○	○	○	○	Very high

How insecure, discouraged, irritated, stressed, and annoyed were you? *

	1	2	3	4	5	6	7	8	9	10	
Very low	○	○	○	○	○	○	○	○	○	○	Very high

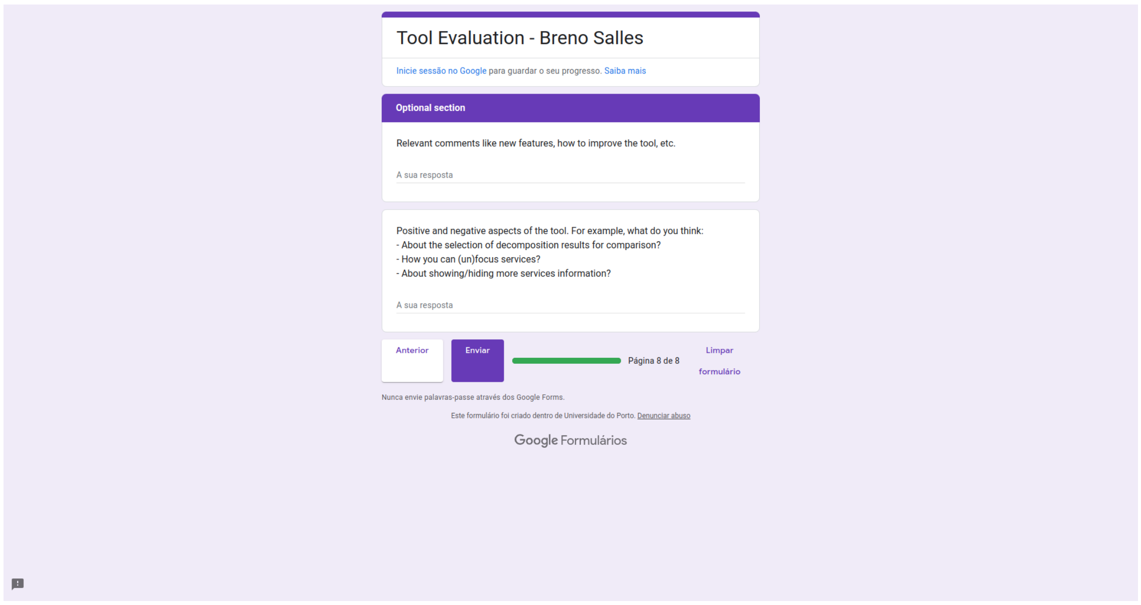
Anterior Seguinte Página 7 de 8 Limpar formulário

Nunca envie palavras-passe através dos Google Forms.

Este formulário foi criado dentro de Universidade do Porto. Denunciar abuso

Google Formulários

Figure A.5: Raw-TLX Questionaire



Tool Evaluation - Breno Salles

Inicie sessão no Google para guardar o seu progresso. Saiba mais

Optional section

Relevant comments like new features, how to improve the tool, etc.

A sua resposta

Positive and negative aspects of the tool. For example, what do you think:
- About the selection of decomposition results for comparison?
- How you can (un)focus services?
- About showing/hiding more services information?

A sua resposta

Anterior Enviar Página 8 de 8 Limpar formulário

Nunca envie palavras-passe através dos Google Forms.

Este formulário foi criado dentro de Universidade do Porto. Denunciar abuso

Google Formulários

Figure A.6: Feedback Questionaire