FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Website Development for the VCMI Research Group

José Pedro Moreira de Almeida Baptista



Mestrado em Engenharia Informática e Computação

Supervisor: Hélder P. Oliveira

First Co-supervisor: Tiago Gonçalves

Second Co-supervisor: João Ribeiro Pinto

July 24, 2022

Website Development for the VCMI Research Group

José Pedro Moreira de Almeida Baptista

Mestrado em Engenharia Informática e Computação

July 24, 2022

Abstract

The researchers of the group of Visual Computing and Machine Intelligence (VCMI) of the Centre for Telecommunications and Multimedia (CTM) from INESC TEC are from the fields of artificial intelligence and computer vision and focus on the development of pattern recognition algorithms. In 2019, they built a website to publish the team member's scientific publications, projects, news, and events.

As time passed, the team grew, and so did the information stored on the application's database. This natural growth introduced severe performance problems and inconsistencies that negatively impacted the visitor's experience and the application's usability. Furthermore, preliminary meetings with the group revealed that the application was still missing some fundamental functionalities, such as a way to publish and let the users interact with the algorithms developed by the team. On the other hand, the team members reported that the website's database management system was inadequate. Hence, the main objectives of this dissertation project consisted of implementing solutions for such issues.

In this project's initial phase, we compared the properties and performance of multiple compatible database management systems and switched to the more adequate and performant one. In the second phase, we improved the application's usability (with a focus on efficiency), as proved by the different usability evaluations. The third phase consisted of incorporating interactive demonstrations in the application, supporting the most common types of algorithm inputs and outputs appointed, and allowing team members to create them easily. Finally, in the last development phase, we concretised some of the suggested enhancements from the requirements and implemented other features to support all the solutions from previous stages.

We successfully concluded all the objectives of this dissertation by delivering a new application version with better usability and responsiveness, interactive demonstrations publishable in a dedicated section for the team's software, and a new database management system. Furthermore, we addressed several security issues and created automated pipelines for some frequent administration tasks. All these changes brought advantages, not only to the visitors, who now have a better experience but also to the team members, who have an appealing way to advertise their work.

Keywords: database management systems, web development, usability, performance, interactive demonstrations

Resumo

Os investigadores do grupo de Visual Computing and Machine Intelligence (VCMI) do Centro de Telecomunicações e Multimédia (CTM) do INESC TEC são das áreas de inteligência artificial e visão de computador e focam-se no desenvolvimento de algoritmos de reconhecimento de padrões. Em 2019, um website foi desenvolvido para publicar as publicações científicas, projetos, notícias e eventos da equipa.

Com o passar do tempo, a equipa cresceu, bem como a informação armazenada na base de dados da aplicação. Este crescimento natural trouxe problemas severos de performance e inconsistências que afetaram negativamente a experiência dos visitantes e a usabilidade da aplicação. Além do mais, reuniões preliminares com a equipa revelaram que algumas funcionalidades indispensáveis ainda estavam em falta na aplicação, nomeadamente uma forma de publicar e permitir que os utilizadores interajam com os algoritmos por eles desenvolvidos. Por outro lado, os membros da equipa apontaram que o sistema de gestão de base de dados que estava a ser utilizado não era adequado. Por isso, os objetivos principais deste trabalho consistem em solucionar estes problemas.

Na fase inicial deste projeto, comparamos as propriedades e performance de múltiplos sistemas de gestão de base de dados compatíveis, e trocamos para o mais adequado e com melhor performance. Na segunda fase, melhoramos a usabilidade da aplicação (focando-nos na eficiência), como é comprovado pelas avaliações de usabilidade conduzidas. A terceira fase consistiu em incorporar demonstrações interativas na aplicação, suportando os tipos mais comuns de entradas e saídas dos algoritmos apontados, e permitir que os membros da equipa as criem facilmente. Finalmente, na última fase do desenvolvimento, implementamos umas funcionalidades de modo a melhorar soluções anteriores, e mais alguns dos requisitos previamente recolhidos em fases anteriores.

Concluímos com sucesso os objetivos desta dissertação, tendo produzido uma nova versão da aplicação com melhor usabilidade e responsividade, a possibilidade de publicar demonstrações interativas numa secção dedicada ao software da equipa, e um novo sistema de gestão de bases de dados. Além do mais, resolvemos alguns problemas de segurança e automatizamos tarefas comuns da administração. Todas estas alterações trouxeram vantagens, não só para os visitantes, que terão uma melhor experiência ao utilizar a aplicação, mas também para os membros da equipa, que passaram a ter uma forma apelativa de publicitar o seu trabalho.

Palavras-chave: sistemas de gestão de bases de dados, desenvolvimento web, usabilidade, performance, demonstrações interativas

Acknowledgments

I want to express my gratitude to my supervisors Prof. Hélder Oliveira, João Pinto and Tiago Gonçalves, for allocating time from their busy schedules to guide me through the restless path of writing a dissertation. I want to thank Tiago, particularly, for his constant availability for regular meetings and advice throughout the past seven months.

I want to express my appreciation to my parents, who put up with me every day, and whose encouragement and care have undoubtedly helped me. I also want to thank my sister and brother for all their support and incentivisation.

Finally, I want to thank the rest of my family and friends who, in their own way, always encourage me to persevere and pursue my dreams.

José Pedro Baptista

"Once you have commitment,
you need the discipline and hard work to get you there."

Haile Gebrselassie

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Motivation . . . 1
- 1.3 Objectives . . . 2
- 1.4 Document structure . . . 2

2 Background **3**
- 2.1 Web-based applications and services . . . 3
  - 2.1.1 The World Wide Web . . . 3
  - 2.1.2 HTTP and HTTPS . . . 4
  - 2.1.3 Web pages, websites, and web applications . . . 5
  - 2.1.4 Web services . . . 5
  - 2.1.5 Virtualisation and Docker . . . 6
- 2.2 Database Management Systems and databases . . . 7
  - 2.2.1 Why use a database? . . . 7
  - 2.2.2 Database models . . . 8
  - 2.2.3 Types of data . . . 9
  - 2.2.4 Relational and non-relational databases comparison . . . 9
- 2.3 Usability . . . 11
  - 2.3.1 Usability measurement models . . . 11

3 State of the Art **14**
- 3.1 Web development and web-based applications . . . 14
  - 3.1.1 Web frameworks . . . 14
  - 3.1.2 Web services and their APIs . . . 16
- 3.2 Database Management Systems . . . 18
  - 3.2.1 Examples and classification . . . 18
  - 3.2.2 How to choose? . . . 19

4 Problem and Methodology **21**
- 4.1 Application architecture . . . 21
- 4.2 Problems . . . 23
  - 4.2.1 Usability improvements and study . . . 23
  - 4.2.2 Interactive algorithm demonstrations . . . 23

5 Requirements Gathering **25**
- 5.1 Application analysis . . . 25
  - 5.1.1 Subjective analysis . . . 25

5.1.2 Audit reports . . . 26
5.1.3 Technology analysis . . . 28
5.2 Interviews with team members . . . 29
5.2.1 Problem enumeration . . . 29
5.2.2 Feature suggestions . . . 29
5.3 Interactive demonstrations . . . 30
5.4 Summary . . . 30

6 Development and Implementation **32**
6.1 Switching DBMS . . . 32
6.1.1 Choice process . . . 32
6.1.2 Implementation and data transfer . . . 36
6.1.3 Database benchmark . . . 38
6.1.4 Conclusions and considerations . . . 39
6.2 Usability study . . . 40
6.2.1 First usability evaluation . . . 40
6.2.2 Usability improvement . . . 43
6.2.3 Second Lighthouse audit . . . 60
6.2.4 Second usability evaluation . . . 62
6.2.5 Conclusion, considerations and limitations . . . 64
6.3 Interactive demonstrations . . . 65
6.3.1 Implementation approaches . . . 65
6.3.2 Software publishing implementation . . . 68
6.3.3 Demo creation helper . . . 74
6.3.4 Conclusions, considerations and criticisms . . . 79
6.4 HTTPS, production environment, automatic publication fetching, and other helpful periodic jobs . . . 79
6.4.1 HTTPS . . . 79
6.4.2 Production webserver and environment . . . 80
6.4.3 Periodic publication fetching . . . 81
6.4.4 Periodic database backups . . . 82
6.4.5 Automatic image conversion . . . 84

7 Conclusion and Future Work **86**
7.1 Conclusion and main contributions . . . 86
7.2 Future work . . . 87

References **88**

A **92**
A.1 Interview with team members . . . 92
A.1.1 Questions . . . 92
A.1.2 Results . . . 96
A.2 Interactive demonstration questionnaire . . . 100
A.2.1 Questions . . . 100
A.2.2 Results . . . 102
A.3 Initial database diagram . . . 104
A.4 First usability evaluation: SUS . . . 107
A.5 Second usability evaluation: SUS . . . 112

List of Figures

2.1 HTTP protocol: client-server interactions. 4
2.2 Hypervisor-based virtualisation and container-based virtualisation (Docker) architectures [35]. 6
2.3 System Usability Scale - Example item with question and Likert scale. 12

3.1 Most commonly used web frameworks, according to a survey made by Stack Overflow in March 2021. 15

4.1 Application component diagram. 21
4.2 Interaction between the front-end and the back-end. 22

5.1 Lighthouse audit results (desktop view). 26
5.2 Lighthouse audit results (mobile view). 27

6.1 Application's component diagram after database implementation. 40
6.2 Publications page after changes. 54
6.3 News page after changes. 55
6.4 Loading animation (frames). 56
6.5 Validated field example. 56
6.6 Floating message example. 56
6.7 Homepage layout before and after the changes. 57
6.8 Second Lighthouse audit results (desktop view). 60
6.9 Second Lighthouse audit results (mobile view). 61
6.10 Preliminary component diagram, assuming producer/consumer-based solution. . 65
6.11 Component diagram, assuming Streamlit-based solution. 66
6.12 Software index page. 69
6.13 Software page. 69
6.14 Example demo page. 73
6.15 Demo helper's first page. 76
6.16 Demo helper's second page. 77
6.17 Demo helper's third page. 78
6.18 Demo helper's fourth page. 78

A.1 Interview questions - Introduction. 92
A.2 Interview questions - Usability Introduction. 93
A.3 Interview questions - Web application usage. 93
A.4 Interview questions - Usability evaluation. 94
A.5 Interview questions - Issues affecting users' experience. 94
A.6 Interview questions - User interface issues. 94

A.7 Interview questions - Admin panel issues. 95
A.8 Interview questions - Back-end issues. 95
A.9 Interview questions - Missing features. 95
A.10 Usability interview section. 96
A.11 Front-end issue enumeration 1. 97
A.12 Front-end issue enumeration 2. 98
A.13 Back-end issue enumeration 1. 98
A.14 Back-end issue enumeration 2. 99
A.15 Feature suggestion section. 99
A.16 Interactive demo questionnaire - Introduction. 100
A.17 Interactive demo questionnaire - Question 1. 100
A.18 Interactive demo questionnaire - Question 2. 101
A.19 Interactive demo questionnaire - Question 3. 101
A.20 Interactive demo questionnaire - Question 4. 101
A.21 Interactive demo questionnaire - Question 1 results. 102
A.22 Interactive demo questionnaire - Question 2 results. 102
A.23 Interactive demo questionnaire - Question 3 results. 103
A.24 Interactive demo questionnaire - Question 4 results. 103
A.25 Database diagram - part 1. 104
A.26 Database diagram - part 2. 105
A.27 Database diagram - part 3. 106
A.28 Usability evaluation 1 - SUS - item 1. 107
A.29 Usability evaluation 1 - SUS - item 2. 107
A.30 Usability evaluation 1 - SUS - item 3. 108
A.31 Usability evaluation 1 - SUS - item 4. 108
A.32 Usability evaluation 1 - SUS - item 5. 109
A.33 Usability evaluation 1 - SUS - item 6. 109
A.34 Usability evaluation 1 - SUS - item 7. 110
A.35 Usability evaluation 1 - SUS - item 8. 110
A.36 Usability evaluation 1 - SUS - item 9. 111
A.37 Usability evaluation 1 - SUS - item 10. 111
A.38 Usability evaluation 2 - SUS - item 1. 112
A.39 Usability evaluation 2 - SUS - item 2. 112
A.40 Usability evaluation 2 - SUS - item 3. 113
A.41 Usability evaluation 2 - SUS - item 4. 113
A.42 Usability evaluation 2 - SUS - item 5. 114
A.43 Usability evaluation 2 - SUS - item 6. 114
A.44 Usability evaluation 2 - SUS - item 7. 115
A.45 Usability evaluation 2 - SUS - item 8. 115
A.46 Usability evaluation 2 - SUS - item 9. 116
A.47 Usability evaluation 2 - SUS - item 10. 116

List of Tables

3.1 DBMS and their respective classification based on the data model and transaction properties [39]. 19

5.1 Summarised audit reports. 27

6.1 Virtual machine characteristics. 33
6.2 DBMS comparison - Query time (best result in bold). 35
6.3 DBMS comparison - Overall time (best result in bold). 35
6.4 Database benchmark results (TPS). 39
6.5 Task completion, by tester. 41
6.6 Time taken in each usability task, by tester. 42
6.7 User satisfaction evaluation - SUS scores. 43
6.8 Number queries executed to fetch page data (before changes). 44
6.9 Number queries executed to fetch page data (after changes). 46
6.10 Summarized audit reports. 61
6.11 Task completion, by tester (second evaluation). 62
6.12 Time taken in each usability task, by tester (second evaluation). 63
6.13 User satisfaction evaluation - SUS scores (second evaluation). 64
6.14 Software page model fields. 68
6.15 Demo model fields. 76

Listings

3.1 Example REST API request. . . . 17
3.2 Example REST API response. . . . 17
3.3 Example JSON returned as response to a GET API request. . . . 18
6.1 Export data command. . . . 36
6.2 Exported data portion. . . . 36
6.3 PostgreSQL service (db) added to docker-compose.yml. . . . 37
6.4 Django's default database configuration. . . . 38
6.5 Load data command. . . . 38
6.6 Pgbench initialisation command. . . . 38
6.7 Pgbench benchmark commands. . . . 39
6.8 Members QuerySet. . . . 45
6.9 Members serializer. . . . 45
6.10 Members QuerySet (optimised). . . . 45
6.11 Script used to rename the images. . . . 47
6.12 NGINX text compression configuration. . . . 48
6.13 NGINX image caching configuration. . . . 48
6.14 DRF default pagination configuration. . . . 50
6.15 Individual pagination configuration. . . . 50
6.16 Simplified "PaginatedNews" APIView. . . . 51
6.17 Example paginated data API response. . . . 51
6.18 Publication searching QuerySet. . . . 53
6.19 Adding a new header bar link. . . . 58
6.20 Adding a new page model. . . . 58
6.21 Resulting SQL table generated from Django's model. . . . 58
6.22 Fields of "AboutPage" to be returned by API. . . . 59
6.23 Software pages QuerySet. . . . 68
6.24 Example demo - initial algorithm. . . . 70
6.25 Example demo - page configuration. . . . 70
6.26 Example demo - page header. . . . 70
6.27 Example demo - input and output components. . . . 71
6.28 Example demo - run button. . . . 71
6.29 Example demo - adapted algorithm. . . . 72
6.30 Example demo - Dockerfile. . . . 73
6.31 Dockerfile mustache template. . . . 74
6.32 Template project build function. . . . 75
6.33 Template project API response. . . . 75
6.34 Openssl certificate signing request and private key creation. . . . 80
6.35 NGINX default server configuration changes to allow HTTPS. . . . 80

6.36 NGINX redirect to HTTPS configuration. 80
6.37 Start back-end with Gunicorn. 81
6.38 Start “cron” daemon command. 82
6.39 Cron-job definition. 82
6.40 Generate ssh key pair. 83
6.41 Authorized-keys file configuration. 83
6.42 The database dump command. 83
6.43 The rsync command. 83
6.44 Django-admin command to rename image path. 84
6.45 Convert image to WebP. 84
6.46 Image conversion cron-job definition. 85

Abbreviations

API	Application Programming Interface
CORBA	Common Object Request Broker Architecture
CSS	Cascading Style Sheets
CTM	Centre for Telecommunications and Multimedia
DBMS	Database Management System
DCOM	Distributed Component Object Model
DRF	Django REST framework
GraphQL	Graph Query Language
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
INESC TEC	Institute for Systems and Computer Engineering, Technology and Science
IoT	Internet Of Things
IP	Internet Protocol
ISO	International Organization for Standardization
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
ORM	Object-Relational Mapper
PNG	Portable Network Graphics
RDBMS	Relational Database Management System
REST	REpresentational State Transfer
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
SSL	Secure Sockets Layer
SUS	System Usability Scale
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TPS	Transactions Per Second
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VCMI	Visual Computing and Machine Intelligence
VM	Virtual Machine
W3C	World Wide Web Consortium
WSGI	Web Server Gateway Interface
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Context

The Visual Computing and Machine Intelligence (VCMI) research group is part of the Centre for Telecommunications and Multimedia (CTM) from the Institute for Systems and Computer Engineering, Technology and Science (INESC TEC). Although its members may have different backgrounds, the team's main work areas are computer vision and artificial intelligence, and they devote their work towards developing pattern recognition algorithms.

The desire to share more than just their scientific publications led the team members to create a simple web application[1]. This website is where the team advertises their work to everyone on the Internet. It is used to publish research-related documents and share news and events the group is involved in. Information about its members and partners is also available.

The application used Vue.js[2] as its front-end framework, Django[3] as its back-end framework, and a SQLite[4] database to store information. All these components were running inside Docker[5] containers, in an INESC TEC server.

1.2 Motivation

Although the VCMI research group already had a web application, it had some shortcomings: on the one hand, it had usability problems, and on the other hand, some important functionalities were missing.

Preliminary meetings with the team members revealed that the application was missing a way of publishing the algorithms developed. An interactive solution for this problem would give the

[1] Available at: `https://vcmi.inesctec.pt/`
[2] Vue.js: `https://vuejs.org/`
[3] Django: `https://www.djangoproject.com/`
[4] SQLite: `https://www.sqlite.org/index.html`
[5] Docker: `https://www.docker.com/`

visitors a reason to spend time on the application and possibly awaken their interest in the team's work.

The issues that harmed the application's usability were more diverse. The most prominent were the high page loading times and lack of responsiveness accompanied by very little visual feedback upon interacting with the page's components. Since keeping the visitors captivated is in the group members' best interests, such issues must be solved.

1.3 Objectives

This dissertation project aimed to improve the VCMI group's application according to their needs by implementing features and enhancements to both the back-end and front-end that they deemed necessary. More precisely, these were its main goals:

1. Gather requirements that correspond to the team's expectations.
2. Switching the application's database management system to a more appropriate one.
3. Improving usability by identifying the most impactful issues and their cause, and fixing them. Once concluded, the visitors should be able to effectively, efficiently and satisfactorily use the applications' functionalities.
4. To make the group members' work more visible and exciting to the visitors by incorporating a way of publishing software and interactive demos of the algorithms developed.

By fulfilling these goals, we expect the number and frequency of visits and the number of interactions with the website to grow.

1.4 Document structure

This dissertation project is composed by six more chapters, organised in the following way:

Chapter 2 defines the fundamental topics to understand this dissertation project.

Chapter 3 presents state-of-the-art technologies and methodologies relevant to this project.

Chapter 4 explains the application's architecture, its main problems, and the methodology followed for each.

Chapter 5 presents the results of all the tasks conducted to gather requirements for this dissertation project.

Chapter 6 details the implementation of the requirements, as well as the reasoning behind most of the decisions made and their limitations.

Chapter 7 concludes this dissertation project by summarising the main contribution to the team's application and pointing out tasks that should be considered in a future continuation of this work.

Chapter 2

Background

This chapter starts by explaining concepts related to web development, such as web-based applications, web services and protocols. Then, it describes the concepts of database management systems and databases. In the same section, topics such as database models and types of data are covered. Finally, a definition of usability and methods to evaluate it are presented.

2.1 Web-based applications and services

2.1.1 The World Wide Web

The World Wide Web (or "web") is "an information space in which the items of interest, referred to as resources, are identified by global identifiers called Uniform Resource Identifiers (URI)"[1]. This system was proposed in 1989 by Tim Berners-Lee[2] at CERN (research organisation near Geneva, Switzerland), and in 1990, he had already created the first web server, web browser and web page[3]. In 1991, the web became public[3], and shortly after, it started being adopted by educational institutions, research laboratories and businesses [40].

The World Wide Web uses the Internet and some of its core protocols, such as the Transmission Control Protocol (TCP), which existed for many years before its invention [40]. It also relies on the URI to make resources referenceable and retrievable [7], the Hypertext Transfer Protocol (HTTP) to transfer data between client and server[3], and the HyperText Markup Language (HTML), which is understood by the client (web browser) and is used to display content [7]. The first specifications of these pillars of the web were also written by Tim Berners-Lee[1].

[1] World Wide Web (W3C): `https://www.w3.org/TR/webarch/#intro`
[2] Tim Berners-Lee's proposal: `https://www.w3.org/History/1989/proposal.html`
[3] World Wide Web (Mozilla):
`https://developer.mozilla.org/en-US/docs/Glossary/World_Wide_Web`

2.1.2 HTTP and HTTPS

The Hypertext Transfer Protocol (HTTP) is the foundation protocol of the World Wide Web [40]. It is a stateless client/server protocol and the basis of any data exchange on the web. It uses a request/response paradigm, where requests are messages sent by the client (in most cases, a web browser), and the responses are the messages sent by the web server as a reply to such requests. Due to its stateless nature, each request-response pair is independent of the previous ones for the web server [3, 40].

Figure 2.1 illustrates an example of the interactions between a client and a web server, where the client opens a new connection with the server and reuses it for several requests.

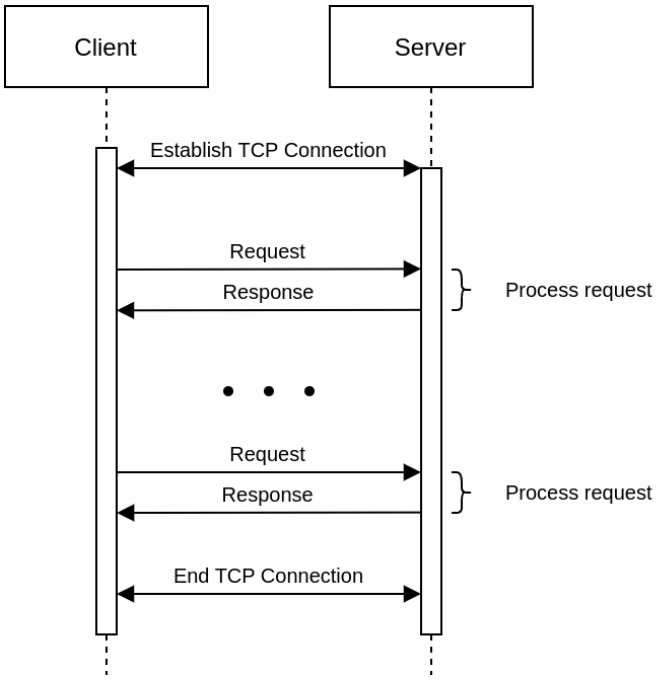


Figure 2.1: HTTP protocol: client-server interactions.

As we can see in the illustration, HTTP relies on a Transmission Control Protocol (TCP) connection to send and receive messages [3]. In websites where sensitive data such as passwords and personal information is transferred, it is common to encrypt this type of connection using the Transport Layer Security (TLS) protocol.

The Transmission Control Protocol (TCP) is a network protocol that operates in the transport layer and is used to ensure reliable delivery of packets over the Internet. If the connection between the client and the web server is encrypted using the TLS protocol (or its predecessor, Secure Sockets Layer (SSL) protocol), we are in the presence of the Hypertext Transfer Protocol Secure (HTTPS) protocol rather than HTTP.

HTTPS is an extension of the standard HTTP protocol that, by using TLS, provides server authentication, confidentiality and integrity to the connection. This means that it guarantees that

the connection is established with the server (and not an attacker) and that none of the data sent can be read nor modified by third parties in the event of packet interception[4].

2.1.3 Web pages, websites, and web applications

Web pages, websites, and web applications are terms that often get confused and are incorrectly used as synonyms. In the context of this dissertation, it is relevant to distinguish them.

A web page (or "page") is an hypertext document provided by a website that can be displayed by a web browser [46] (e.g. Google Chrome, Firefox, Chromium, Brave and Safari).

A website (or "site") is a collection of web pages that are connected to each other through hypertext links [46]. Although this definition may seem simple, it has underlying concerns: a website should maintain a thematic consistency of content and a similar look and feel (i.e., layout, design and elements) across all of its pages to create and define its identity. When these concerns are addressed, the visitors should quickly identify a particular website just by looking at one of its pages [40].

A web application (or "web app") consists of a client part that is executed on a user's web browser and a server part that runs on a web server [23]. These parts are often called front-end and back-end, respectively. In a server-side rendering configuration, the web server dynamically builds the HTML documents and delivers them to the client[5]. On the other hand, in a single-page client-side application, the web server serves a base HTML document, and the rest of the rendering is done through JavaScript (on the web browser). In both cases, the client requests information, and upon receiving it, the web browser interprets and displays the page. Unlike a website whose content is static, a web application presents content that may be adapted based on multiple sources, such as the request parameters and user session information [23, 40].

2.1.4 Web services

Web services are modular networked applications that expose their business logic as services over the Internet and use standard web protocols over well-defined interfaces, also known as application programming interfaces (APIs) [10, 18]. An API is a set of methods and rules by which applications exchange messages and transfer data. Furthermore, it helps organise interactions between them [26]. Although the type of API may affect the way the service is used, generally, other applications may leverage these services by invoking the desired method in the provided API through the Internet Protocol (IP) of the server where the service was deployed [10, 18]. There are no operating system or programming language requirements to use these services, thus promoting interoperability among devices. Depending on the type of service, data transfer may be done in the Extensible Markup Language (XML) or JavaScript Object Notation (JSON) formats.

[4] TLS assurances: `https://developer.mozilla.org/en-US/docs/Glossary/TLS`
[5] Server-side rendering with Vue: `https://vuejs.org/guide/scaling-up/ssr.html`

2.1.5 Virtualisation and Docker

Server virtualisation is a technological innovation that facilitates building multiple virtual machines (VM) on a single physical machine in the form of containers or hypervisors [35]. Most cloud computing systems rely on container-based or hypervisor-based virtualisation [8].

One of the key differences between hypervisor-based virtualisation and container-based virtualisation is that the first provides isolation at the hardware abstraction layer. In contrast, the second offers operating system level abstraction and isolation. Another distinction factor among them is that container-based virtualisation is much lighter on the system resources (due to sharing them and the operating system among containers), outperforming hypervisor-based VMs in performance and scalability [44]. Although these also provide some degree of isolation, if, for security reasons, a higher degree of isolation is needed, a hypervisor-based VM is a requirement [8]. Figure 2.2 compares the architectures of these virtualisation technologies.

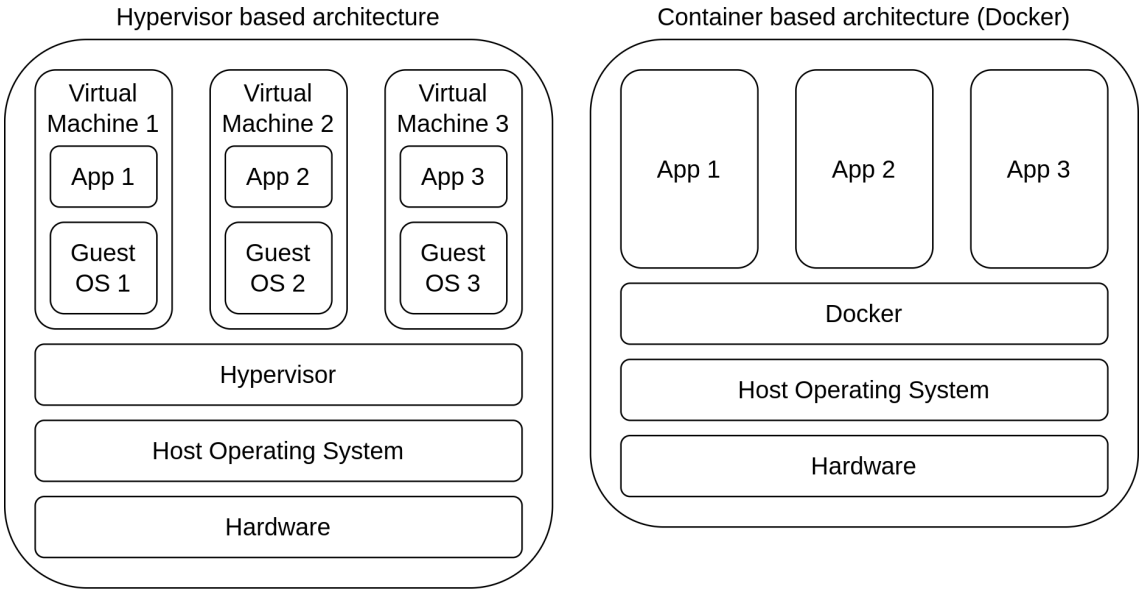


Figure 2.2: Hypervisor-based virtualisation and container-based virtualisation (Docker) architectures [35].

2.1.5.1 Docker

Docker is an open-source container-based virtualisation technology [4, 44]. It was developed to be a lightweight computing technology and make the containerised applications portable, that is, behave the same regardless of the machine it is running on (as long as Docker supports the host system) [4]. In order to build a container, Docker requires an image.

Docker images

Docker images contain application code, libraries and binaries required to run it, and some supporting packages [4]. Images may be based on other images, and every command executed (i.e.

for installing dependencies, moving files or other operations) is layered on top of it [8], thus making image building a multi-step process. The automatic way of building an image involves using a Dockerfile, a script where every step required for building the image is specified [8].

Docker containers
A Docker container is an instance of a Docker image, where the application code is run in a confined way [35]. Docker's implementation depends on the Linux kernel *namespaces* and *cgroups*. The namespaces are used to give containers access to system components such as the CPU, memory and network and to provide isolation from the other processes. In contrast, cgroups are used to manage the resources available to the container [4].

2.2 Database Management Systems and databases

A database is a collection of interrelated data and information that is stored and organised to promote its access and management. Data can be added to a database, removed or updated [19,39].

A Database Management System (DBMS) a is software for creating and managing databases. It acts as an interface between the users (or other programs) and the database to facilitate data organisation and manipulation. Besides managing data, the DBMS also manages the logical structure of the database (schema) and the database engine, whose function is to manipulate data (retrieve, update or lock) [19].

2.2.1 Why use a database?

The usage of a database, as opposed to merely files in the file-system, can be very beneficial, especially for large scale systems. These are some of the advantages that a database may bring [39]:

- **Reduced data redundancy**: Data redundancy is the storage of duplicate data. This not only uses more disk space but also may be the cause of data inconsistencies when updates happen. By reducing (or removing) redundancy, these problems are mitigated.
- **Data searching**: Data stored in a database may easily be accessed and searched based on a given property.
- **Data integrity**: The DBMS guarantees that processes such as insertion, deletion, and updating of records keep the data integrity, that is, that the data follows the imposed constraints over its life-cycle.
- **Atomic updates**: Atomic updates guarantee that no changes are committed in case of failure.
- **Concurrent access by multiple users**.

2.2.2 Database models

Since data is a dynamic entity, the way it is stored varies. The logical structure that describes how a database is structured and used is called the database model [19]. We describe some of these models below.

Hierarchical

The hierarchical data model was introduced in the 1960s by IBM [39]. In this database model, data is arranged in a tree structure, with data at each level of the hierarchy dependent upon the data at the higher level of the hierarchy [1]. Being structured as a tree implies that each child record (data from a lower hierarchy level) only has one parent record (data from a higher hierarchy level). Parent records may have multiple children. In order to get data from such a structure, the whole tree needs to be transversed from the root [39].

Network

The network data model was developed to replace the hierarchical data model, and although similar, it allowed for children records to have multiple parents, thus forming a network. It was the first data model to introduce many-to-many relations. Due to its flexibility when compared to the hierarchical data model, DBMS that supported this type of data model became very popular in the early 1970s [39].

Object-oriented

In an object-oriented database, information and data are represented by complex data objects, that is, objects that mirror those used in object-oriented programming languages [31, 39]. Objects are instances of classes and may contain attributes and methods. An object-oriented data model also supports object-oriented design patterns such as inheritance, polymorphism, overriding, and association [31].

Relational

Due to the high adoption of relational databases, this data model is the most common [39]. In the relational data model, information is structured and stored in tables. Tables are composed of rows and columns - or tuples and attributes, respectively [2, 19]. Every row contains a unique primary key that is used to identify it. Relations between tables are also supported and are achieved through the use of foreign keys, which are columns in a row that contain a unique identifier for a given row of the other table [19].

Non-relational

Non-relational (or NoSQL) databases are the most recent from the list and are named as opposed to the relational ones. There are many non-relational databases, however, their data model usually falls within one of the following four categories [2]:

- **Key-value**: In this model, information may have any structure and is saved like in a hash table, that is, in a map that given a key, returns the corresponding data [2, 36].
- **Column-oriented**: This model is a composite approach between the relational and key-value models. The data is stored in column families. Every column family may have multiple rows, each identifiable with a row-key. Furthermore, a row may have multiple columns of different sizes, each containing information [36].
- **Document-based**: This model is the most used amongst the non-relational database models. It can store structured, semi-structured and unstructured data [36] by saving it in documents [29]. A document is a flexible way of storing data that uses field-value pairs, where the field is a string, and the value may be any other data type. Documents can be stored in formats like JSON and XML [29].
- **Graph-based**: The graph-based data model, as the name suggests, stores data in the form of a graph, which is a collection of nodes and edges. Nodes act as entities, while edges represent relationships between them [36].

2.2.3 Types of data

The type of data that needs to be stored is also a relevant consideration. Some DBMS (and respective supported data models) are better suited for storing a given type of data than others.

Data may be classified as structured or unstructured based on its structural nature. Structured data is highly specific and consistently follows a predefined structure, type and formatting. This type of data is organised and may easily be analysed and searched due to its fixed structure [19]. Some examples of this kind of data are ticket reservations and inventory systems. On the other hand, unstructured data is the broader kind among the two because it does not have a consistent structure. Its unorganised nature hinders its processing, thus making it more difficult to be searched. Some examples of this kind of data are emails, video and audio [19].

2.2.4 Relational and non-relational databases comparison

Non-relational databases were introduced as a solution for storing large amounts of unstructured data, a task that is impossible for relational databases due to their strict structured nature [36]. For this reason, non-relational databases have been gaining popularity [28], however, they do not replace relational ones, as both have advantages and disadvantages [36], depending on the use-case. Some of their distinguishing factors are:

Flexibility

Relational databases have a static schema (data structure) that should be predefined before data injection. Posterior modifications to the schema may be complex. On the other hand, non-relational databases have a dynamic schema that, although possible, does not need to be predefined and can easily accommodate posterior changes [19, 36]. Furthermore, relational databases can only

store structured data, as opposed to non-relational databases, which can store all types of data (structured, semi-structured and unstructured) [39].

Disk space

The data stored in a relational database is highly normalised [36], which avoids data redundancy and therefore occupies less disk space. On the other hand, non-relational databases rely on redundancy to improve other aspects and thus may occupy more disk space [30].

Query performance

Non-relational databases queries (data retrievals) are usually faster than relational ones. One of the reasons behind this is that in relational databases, due to its high normalisation, it is likely needed to perform table joins, and as its tables grow in size, these operations can become computationally expensive. On the other hand, non-relational databases typically do not require joins, and due to data redundancy, queries can be finished faster [2, 30].

Scalability

Relational databases are vertically scalable. This means that when the system's maximum capacity (imposed by the hardware) is reached, either the system's hardware is updated, or the database is migrated to a more capable and powerful system [2, 30, 36]. Non-relational databases, by contrast, are horizontally scalable, which means that when the system's maximum capacity is reached, more systems may be added to distribute the load [19, 30, 36].

ACID and BASE transaction models

Relational databases meet the ACID (Atomicity, Consistency, Isolation and Durability) properties [36]. "Atomicity" means that a transaction (a group of database read and write operations) only succeeds if all its operations succeed, otherwise, it is rolled back [37]. "Consistency" guarantees that changes made within a transaction are consistent with database constraints [30]. "Isolation" guarantees that concurrent transactions do not affect each other's outcomes [30]. Finally, "Durability" guarantees that the results of the transaction are permanent, even in the presence of failures [37].

On the other hand, although not all [30, 36, 37], most of the non-relational databases meet the BASE (Basically Available, Soft State, Eventually Consistent) properties instead. "Basically Available" means that data are available most of the time. A failure could cause a part of the data not to be available, but only catastrophic failures could cause a global failure [17]. "Soft State" means that databases provide a relaxed view of data in terms of consistency [17]. Due to the lack of immediate consistency, its values may change over time. "Eventually Consistent" means that data will become consistent, eventually. Until it happens, data reads are possible, although the returned values might not reflect the reality.

BASE models tend to prioritise availability over consistency[6] and its properties are much looser than ACID ones. Still, there is no direct one-for-one mapping between the two transaction models, and none of them is perfect for every situation [37].

2.3 Usability

Depending on the context, usability may have different implications and definitions. For this project, the most relevant definitions are those in the fields of human-computer interaction and software engineering.

In the field of human-computer interaction, it is widely accepted the definition proposed by the International Organization for Standardization (ISO), in its standard ISO 9241-11:2018 [6,21]. This standard defines usability as the "extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use" [21]. On the other hand, in the standard ISO/EIC 25010:2011, which proposed a product quality model for software and computer systems, usability is considered a quality characteristic of a software product [20]. In this document, usability is defined as the "degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use" [20]. Even though the fields differ, possibly due to their correlation, both definitions are identical. Three attributes can be identified from these definitions:

- **Effectiveness**: Refers to the level of accuracy and perfection performed by a user in completing a task for a specific objective [21,42].
- **Efficiency**: Refers to the relation between the resources used and the results achieved. Typical resources include time, human effort, costs and materials [21].
- **Satisfaction**: Refers to the extent to which the user's emotional response that results from interacting with the system meets their expectations and needs [21].

2.3.1 Usability measurement models

A usability measurement model (or usability evaluation method) is a procedure composed of well-defined activities for collecting usage data and identifying specific properties of the target software that contribute to achieving usability. If its methodology relies on collecting and analysing usage data from users, it is considered empirical. On the other hand, if it relies on the analysis of expert evaluators or designers, it uses an inspection method [13].

2.3.1.1 System Usability Scale

The System Usability Scale (SUS) is an evaluation tool that proved to be robust and reliable. It was created due to the necessity of a fast and cheap system usability evaluation method within an

[6] ACID and BASE transactions: `https://phoenixnap.com/kb/acid-vs-base`

industrial context. It is a Likert scale (a five-point agreement scale) composed of ten questions that give a global view of subjective assessments of usability [22]. According to [5, 22], the suggested statements are:

1. I think that I would like to use this system frequently;
2. I found the system unnecessarily complex;
3. I thought the system was easy to use;
4. I think that I would need the support of a technical person to be able to use this system;
5. I found the various functions in this system were well integrated;
6. I thought there was too much inconsistency in this system;
7. I would imagine that most people would learn to use this system very quickly;
8. I found the system very cumbersome to use;
9. I felt very confident using the system;
10. I needed to learn a lot of things before I could get going with this system.

Each item contains a statement and the respective agreement scale, and should be presented to the user as Figure 2.3 shows.

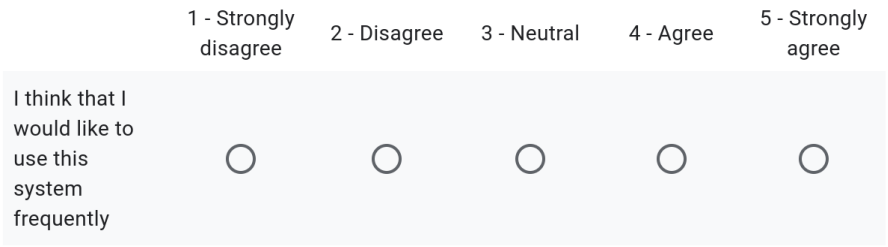


Figure 2.3: System Usability Scale - Example item with question and Likert scale.

The SUS is designed to be applied after the user has had the opportunity to utilise the target system, but before any discussion occurs. The user is supposed to give their prompt response to each of the ten items, and in the case of not being able to answer, the centre point of the scale should be selected [22].

From the application of this method results a number (between 0 and 100) representing a composite measure of the overall usability of the system being studied [22].

2.3.1.2 Other usability measurement models

Evaluation methods may be based on different definitions and interpretations of usability, thus, proposing different usability attributes to be measured. These are some examples of usability models proposed by other authors:

- **Nielsen's model** [33]: According to this model, usability is considered a property of a user interface and may be decomposed into the attributes of learnability (how easy the system is to learn), efficiency, memorability (the system should be easy to remember), errors (the system should have low error rate) and satisfaction [5, 6, 33].
- **Quality in Use Integrated Measurement** (QUIM): This model consolidates several other standards and conceptual models. According to this model, the ten usability factors are efficiency, effectiveness, productivity, satisfaction, learnability, safety, trustfulness, accessibility, universality and usefulness. These may be further decomposed into a total of 127 measurable, specific metrics [38].
- **Oztekin, Nikov and Zaim's model** [34]: Usability Website Information System (UWIS) is a methodology for usability assessment and design of web-based information systems through a checklist. In its assessment model, efficiency, effectiveness, and satisfaction are considered high-level "mostly objective" [34] measures that cannot be directly changed by the designer, but rather as a consequence of changing low-level measures. The considered low-level measures are reliability, integration of communication, navigation, controllability, assurance, responsiveness, and quality of information.

Chapter 3

State of the Art

This chapter presents some state-of-the-art frameworks and practices relevant for developing web-based applications/services and state-of-the-art database management systems.

3.1 Web development and web-based applications

Web development and web applications have been topics of high interest worldwide, and specifically in the last five years, there has been an increase in interest in such topics[1]. Furthermore, the number of applications that were previously released for the desktop (as dedicated applications and targeting a given operating system) and later converted to web-based versions is also growing [11]. These changes originated an overall increase in the requirements of this kind of applications, thus creating new challenges and problems to overcome. Nowadays, the main requirements of a web-based application are the ability for it to scale, to handle not only large amounts of data, but also a large number of concurrent requests (from multiple users) [27], and do all of the above while still being secure. In order to fulfil these requirements in a reasonable amount time, web frameworks are often used in the development process [23, 24].

3.1.1 Web frameworks

A web framework is a high-level solution framework that aims to support the development of web applications and decrease their development time by providing the developer tools and libraries that simplify common web development tasks such as accessing databases, managing sessions and page construction. These often promote the reuse of code and/or software pieces [24].

There are many examples of this type of web technology, each one of them with its advantages and shortcomings when compared to one another [23]. This means that, depending on the context and target requirements of the project, a given framework might be better than the other, thus

[1] According to Google Trends: `https://trends.google.pt/trends/explore?date=today%205-y&q=web%20development,web%20app`

making the process of choosing a framework requires some research, in order to make a well informed decision. A poor choice in this phase may lead to problems such as failing to meet the project deadlines and in more extreme cases, its requirements [24]. Some common distinction factors amongst them are: the state of development (active/inactive), the effort required to learn it (and possibly its language), performance benchmarks, the quality of the documentation, its popularity in support forums (such as Stack Overflow[2]), the availability of plugins and packages to extend its functionalities and support for testing. Another good indicator in favour of a framework is its popularity and adoption worldwide, as it is usually related to software maturity and support.

Web frameworks are often classified as front-end or back-end frameworks based on whether their primary use is in the client-side or the server-side of the web application.

Figure 3.1 shows the results of a survey[3] made by Stack Overflow, in March 2021, with 49941 responses, and it lists the 18 most used frameworks by professional web developers.

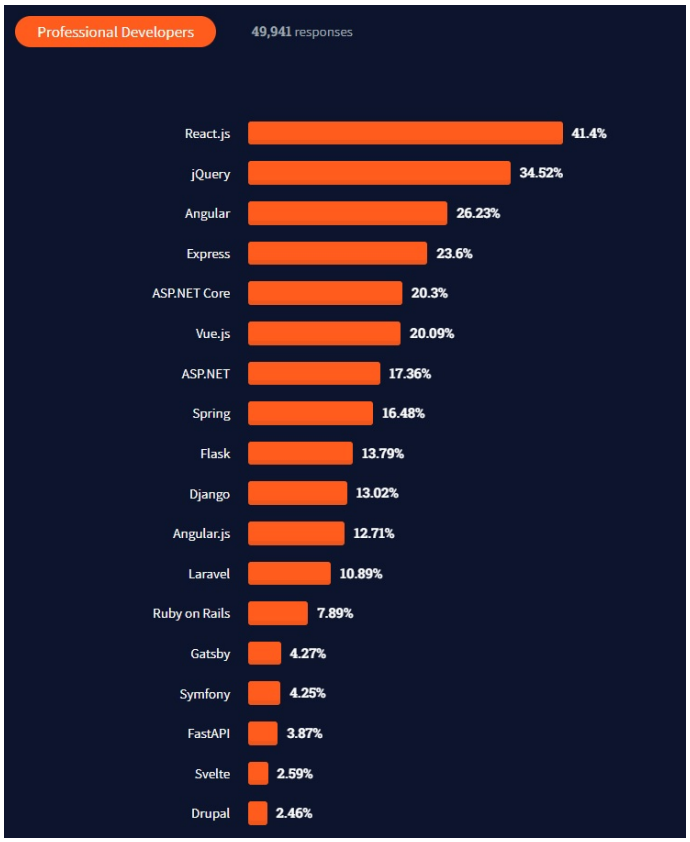


Figure 3.1: Most commonly used web frameworks, according to a survey made by Stack Overflow in March 2021.

As the popularity of the web grew, many technologies emerged. JavaScript was and still is one of the most impactful ones [11]. Many of the previously mentioned frameworks, as well

[2]Stack Overflow: `https://stackoverflow.com/`

[3]More information about the survey can be found here: `https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-web-frameworks`

as the web applications we use nowadays, depend on JavaScript and its features to operate. Its importance mainly derives from allowing web pages to be more dynamic and making page reloads less frequent [26], or in some cases, even non-existent (these used to be required every time a user had to submit data to the web server).

These changes allowed both the client-side and the server-side to evolve. In a modern web application built with, for example, React.js as the front-end framework, the client-side can generate HTML markup in real-time [26] and populate it using data requested from the server-side, all without reloading the page. On the other hand, the server-side has evolved into web services, promoting interoperability. It handles requests, interacts with the database and other services (if any). Although the physical architecture of web applications did not change, the way its components interact is now commonly done through a web API provided by the application's back-end.

3.1.2 Web services and their APIs

Web services depend on their APIs to receive requests, trigger the execution of a given task, and return its result. Technologies such as RMI (Remote Method Invocation), CORBA (Common Object Request Broker Architecture) or DCOM (Distributed Component Object Model) can make this communication possible, however, these often create security and compatibility issues [18]. Nowadays, the four main architectural styles of a web service's API are the remote procedure call (RPC), simple object access protocol (SOAP), graph query language (GraphQL) and REpresentational State Transfer (REST) [26].

RPC allows remote function calling over an HTTP API. It is primarily used in the development of high-performance micro-services and in the field of Internet Of Things (IoT) [26].

SOAP is a lightweight, platform-independent protocol for a decentralised, distributed environment. It is a World Wide Web Consortium (W3C) standard that uses XML to define the communication and HTTP to transmit the messages [18].

The graph query language (GraphQL)[4] uses JSON format and was introduced by Facebook developers. Its main use cases are in complex systems where logic encapsulation is beneficial and mobile devices with limited data transfer [26].

As REST is the most common type of API nowadays [26] and the one used in team's application, it will be explained in greater detail in the next section.

3.1.2.1 REST and RESTful web services

REST is a style of API architecture for creating web services. It was introduced in 2000 by Roy Fielding, and it supports both JSON and XML [18]. A web service that meets all the requirements and conditions of the REST architecture is called a RESTful web service [26]. Nowadays, these have become the most common type of web service for new projects at a professional level [18]. Some of their characteristics are:

[4]GraphQL: `https://graphql.org/`

- **Statelessness**: Each request contains all of the information necessary for the server to understand and process the request and is independent of any other requests that may have preceded it [14]. It removes the need to store state information, thus reducing the consumption of physical resources and improving scalability. Furthermore, it allows parallel processing of requests [14]. Listing 3.1 is an example of a request to a REST API.
- **Addressability**: All resources must be identifiable by a unique Uniform Resource Identifier (URI) [18]. A resource translates to anything that can be named. A document, an image, a temporal service (e.g., weather forecasting) are only a few examples of valid resources, according to [14]. Resources may be static or change over time. Regardless, their identifier must be static since it distinguishes one resource from another.
- **Uniform interface**: The API should be uniform and use standard HTTP methods to provide access to the resources [18]. In REST, all communications are based on the use of HTTP methods, and they define the type of request that is sent to the web server. The four main methods available are: GET - Used to request data; PUT - Used to perform data editing; POST - Used to add new information; DELETE - Used to delete data [45]. In the Listing 3.1 request example, the method GET was used.

  This characteristic allows for a consistent, application-independent interaction with the web server [18] and promotes substitutability, that is, as long as the interface does not change, the implementation of the service can be replaced or improved without impacting its clients [14].

Listings 3.1, 3.2 and 3.3 exemplify a simple interaction with free web service, through a REST API.

```
> GET /api/activity HTTP/1.1
> Host: www.boredapi.com
> User-Agent: insomnia/2021.7.1
> Accept: */*
```

Listing 3.1: Example REST API request.

```
< HTTP/1.1 200 OK
< Server: Cowboy
< Connection: keep-alive
< X-Powered-By: Express
< Access-Control-Allow-Origin: *
< Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept
< Content-Type: application/json; charset=utf-8
< Content-Length: 113
< Etag: W/"71-cSDCHznnjtYpCyTJu+F685xIFmw"
< Date: Wed, 26 Jan 2022 17:16:55 GMT
< Via: 1.1 vegur
```

Listing 3.2: Example REST API response.

```
{
  "activity": "Write a song",
  "type": "music",
  "participants": 1,
  "price": 0,
  "link": "",
  "key": "5188388",
  "accessibility": 0
}
```

Listing 3.3: Example JSON returned as response to a GET API request.

3.2 Database Management Systems

For many years, Relational Database Management Systems (RDBMS) have been the most common DBMS, however, due to the information growth in organisations and the need for increased scalability and performance, alternatives to these systems evolved [19]. Non-relational database management systems emerged as a possible solution to achieve such requirements and are thus increasingly being used, especially in Big Data and real-time web applications [2]. Nowadays, DBMSs are, in most cases, either relational or non-relational.

3.2.1 Examples and classification

Table 3.1 contains some of the most used relational and non-relational database management systems by professional developers nowadays[5], as well as their classification according to supported data models and properties. Neo4j does not belong to the list of most used and was only added to include a DMBS with a graph-based data model.

[5] Available at: `https://insights.stackoverflow.com/survey/2021#most-popular-technologies-database-prof`

Table 3.1: DBMS and their respective classification based on the data model and transaction properties [39].

DBMS Name	Data Model	Transaction Properties
Cassandra[6]	Non-relational: Column-oriented	BASE compliant
DynamoDB[7]	Non-relational: Key-value	ACID compliant[8]
MariaDB[9]	Relational	ACID compliant
Microsoft SQL Server[10]	Relational	ACID compliant
MongoDB[11]	Non-relational: Document-based	ACID compliant[12]
MySQL[13]	Relational	ACID compliant
Neo4j[14]	Non-relational: Graph-based	ACID compliant[15]
Oracle[16]	Multi model	ACID compliant[17]
PostgreSQL[18]	Object-relational	ACID compliant
Redis[19]	Non-relational: Key-value[20]	ACID achievable[21]
SQLite[22]	Relational	ACID compliant

Although most DBMS only support one data model, some of the technologies included in the list (such as Oracle and Redis) support multiple.

3.2.2 How to choose?

The choice of the DBMS affects the entire lifecycle of the product and therefore is a challenging decision [25]. In order to perform a well-informed decision about which DBMS to use, it is essential to take into consideration the purpose of the application and what its requirements are. More precisely, these are some of the technical factors that should be carefully weighed and matched against each DBMS's characteristics while making a decision [19, 25]:

1. Type and amount of data that needs to be stored;

[6]Cassandra: `https://cassandra.apache.org/_/index.html`
[7]DynamoDB: `https://aws.amazon.com/dynamodb/`
[8]Properties: `https://aws.amazon.com/blogs/aws/new-amazon-dynamodb-transactions/`
[9]MariaDB: `https://mariadb.org/`
[10]Microsoft SQL Server:`https://www.microsoft.com/pt-pt/sql-server/`
[11]MongoDB:`https://www.mongodb.com/`
[12]Properties: `https://www.mongodb.com/basics/acid-transactions`
[13]MySQL: `https://www.mysql.com/`
[14]Neo4j: `https://neo4j.com/`
[15]Properties: `https://neo4j.com/product/neo4j-graph-database/`
[16]Oracle: `https://www.oracle.com/pt/database/`
[17]Properties:
`https://docs.oracle.com/cd/E11882_01/server.112/e40540/transact.htm#CNCPT117`
[18]PostgreSQL: `https://www.postgresql.org/`
[19]Redis: `https://redis.io/`
[20]Note: Redis may support other data models when using modules.
[21]ACID is achievable with performance penalties; Not recommended:
`https://redis.com/blog/your-cloud-cant-do-that-0-5m-ops-acid-1msec-latency/`
[22]SQLite: `https://www.sqlite.org/index.html`

2. Schema characteristics;
3. Transaction amount and frequency;
4. Desired performance;
5. Cost limitations;
6. Hardware limitations.

Chapter 4

Problem and Methodology

This chapter starts by briefly explaining the architecture of the team's web application at the beginning of this project, then presents the two main problems that this dissertation aims to solve, as well as the methodology followed for each.

4.1 Application architecture

The team's web application comprises two front-end applications and a back-end one, each running inside a Docker container, deployed to a VM inside an INESC TEC server. Figure 4.1 is a diagram of the components of the team's application.

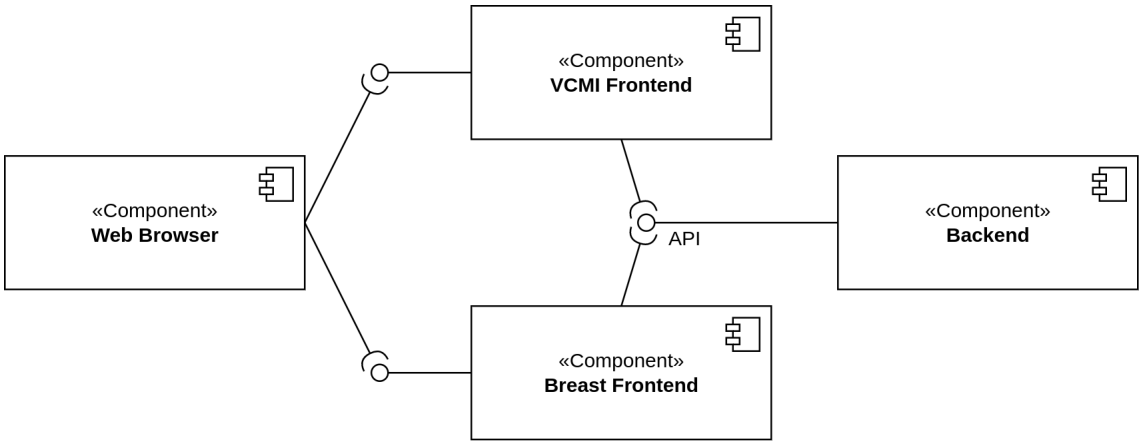


Figure 4.1: Application component diagram.

As we can see from the diagram, we have two front-end applications. The "VCMI" one is the default, while the "Breast" was created for a team's sub-group. These are very similar code-wise, the main difference being the content they fetch and display. For this reason, in some sections of this document, they are treated as one.

The team's application uses Vue.js[1] as its front-end framework and Django[2] as its back-end framework. Files (such as images) are stored in the file system, and other data, such as session information, pages, projects and news, are stored in an SQLite[3] database. Both of the described data storages occur in the back-end. The client-side uses Axios[4] to leverage server-side services through a REST API, and the server-side uses the library Django REST framework[5] to provide them.

All the components (or services) that run inside their Docker containers are independent, thus allowing each one to use its specific libraries, frameworks, and technologies. Moreover, this approach allows each service to be independently changed and redeployed without requiring the entire application to be restarted [16, 44]. Another advantage of this architecture approach is that each component may be independently scaled as needed [44]. Even though independent and technologically different, these containers communicate with each other through a REST API. Based on these factors, we can conclude that the team's application follows a microservice architecture [16].

In a typical interaction, after the client-side sends a request and the server-side receives it, Django executes the requested service, which might involve interacting with the database or even other external services, and responds. Upon reception, the data is used to populate Vue.js's components and, thus, the web page. Figure 4.2 is a diagram of the described interaction.

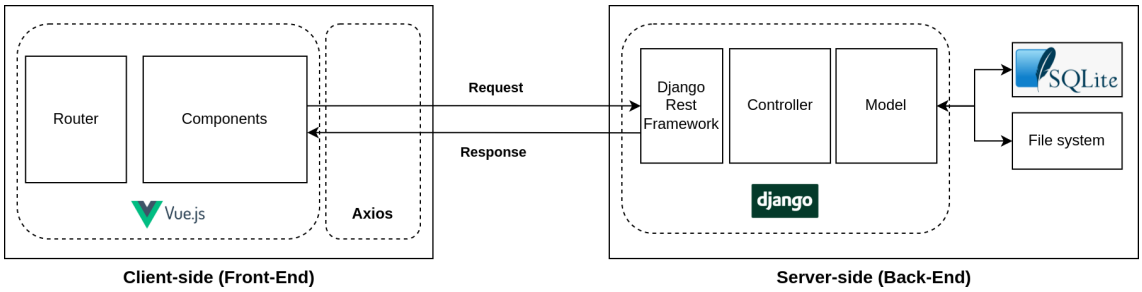


Figure 4.2: Interaction between the front-end and the back-end.

It is important to note that although the term "back-end" may refer to anything that runs on the server-side, throughout this dissertation project, when we use it, we are referring specifically to the container where Django is running.

[1] Vue.js: `https://vuejs.org/`
[2] Django: `https://www.djangoproject.com/`
[3] SQLite: `https://www.sqlite.org/index.html`
[4] Axios: `https://axios-http.com/`
[5] Django REST framework: `https://www.django-rest-framework.org/`

4.2 Problems

This dissertation's problem is twofold: on the one hand, the study and improvement of the team's web application usability, and on the other, the addition of interactive web-based algorithm demonstrations.

4.2.1 Usability improvements and study

Mainly motivated by the high page loading times and other inconsistencies, the first problem consisted of identifying the most impactful issues currently affecting the usability of the team's web application and fixing them. In order to both guide and evaluate the impact of such changes, this process was accompanied by a usability study, based on methods previously mentioned in section 2.3.1.

4.2.1.1 Methodology

The methodology we followed to solve this problem is composed of the following steps:

1. **First usability evaluation**: In order to evaluate the usability of the team's web application, an approach based on [5] was followed, where the tester was given a set of tasks to complete using the application. Effectiveness was measured by the completion rate of such tasks and efficiency based on the time taken to complete them. The user satisfaction was measured using the SUS method [5, 22].

   The first step objectively evaluated the application's usability before any changes were made so that it could be later compared with the new version. It also allowed us to identify the usability attributes that need improvement, thus helping to identify the most impactful issues to solve.

2. **Implementation**: Consisted in selecting both the issues that harm the application's usability and the feature suggestions that contribute to improving the most lacking usability characteristics identified in the previous step, and solving/implementing them. This step was the most time-consuming of this problem.

3. **Second usability evaluation**: The second usability evaluation was performed in the same conditions as the first one so that both results would be comparable. The purpose of this evaluation is to give an insight into how much each usability characteristic has improved (if at all) after the changes were implemented.

4.2.2 Interactive algorithm demonstrations

The second problem consisted of developing a solution that allows the team members to reliably and securely publish interactive demonstrations of their algorithms in their web application so that its visitors may effortlessly try them.

4.2.2.1 Methodology

The methodology followed to solve this problem was composed of the following steps:

1. **Approach study**: In order to explore multiple implementation possibilities and make an informed decision, the first step consisted of finding possible solutions for the problem, previewing the architectural changes that these imply, as well as their advantages, disadvantages, and implications.

2. **Software publishing**: In this step, a dedicated "Software" section on the application was created to allow the publishing of software and interactive demonstrations.

3. **Example demonstration**: This step consisted of following the approach decided in the first phase and implementing an interactive demonstration to validate the approach.

Chapter 5

Requirements Gathering

This chapter describes the requirements and feature suggestions extrapolated from the multiple tasks done throughout the requirement gathering phase of this dissertation project.

5.1 Application analysis

This section presents the results of three examinations of the team's initial web application. These were conducted with the intent of identifying issues and gathering requirements. The first consists of personal observations from using the web application (i.e., subjective analysis), the second of analysing the results generated by an automatic audit tool, and the third is an overview of the technologies used to build the application, as well as considerations about their adequateness to this dissertation's objectives.

5.1.1 Subjective analysis

This analysis consisted of using the web application and noting the most evident negative aspects in a non-exhaustive manner. From this analysis, three issues were identified:

- **High page loading times**: Some pages took around 7 seconds to load.
- **Visual artefacts in mobile phone view**: Some pages displayed overlapping components such as buttons and text when visited through a mobile phone (especially with a small screen).
- **Lack of HTTPS**: The connection uses HTTP instead of HTTPS. This represents a security issue, as a potential attacker can monitor and simply read requests and responses containing sensitive information [9].

5.1.2 Audit reports

This analysis consisted of using Lighthouse[1] (version 9.1.0), an open-source audit tool, to analyse the team's application. It strives to improve the quality of web pages by auditing, among others, its performance, accessibility, and best practices.

The audits were executed in the Chromium[2] web browser (version 98.0.4758.80) and using private browsing to avoid interference due to cached data or browser extensions. Each audit is relative to a single web page and was run twice, one for desktop and the other for mobile views. The reports produced a score for each parameter, ranging from 1 (worst) to 100 (best). Figures 5.1 and 5.2 contain the results obtained in every page, and Table 5.1 summarises the scores.

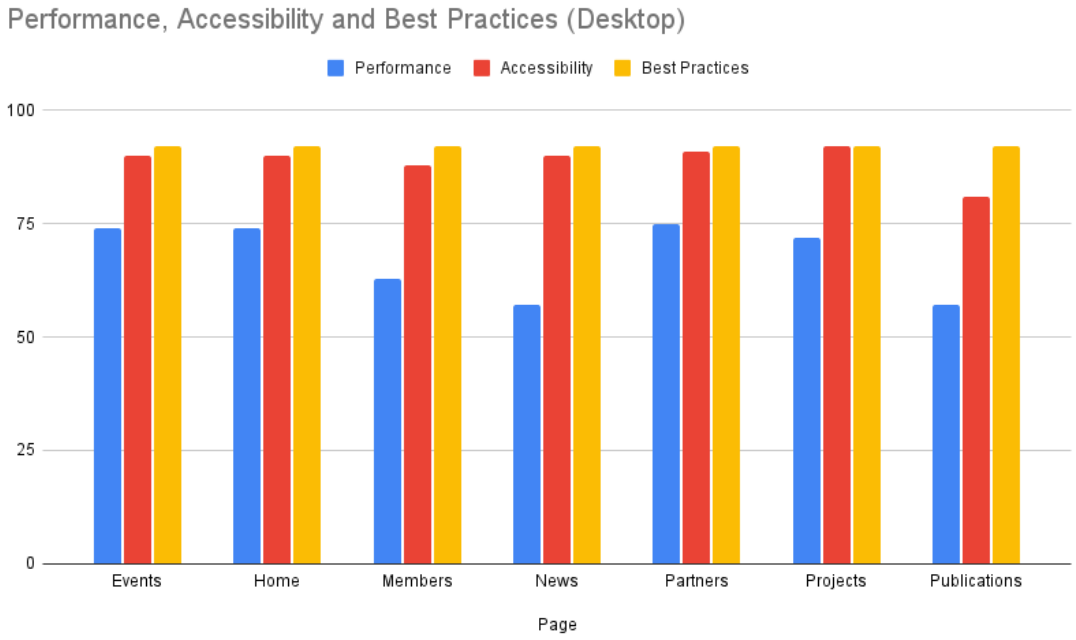


Figure 5.1: Lighthouse audit results (desktop view).

[1]Lighthouse: `https://developers.google.com/web/tools/lighthouse`
[2]Chromium: `https://www.chromium.org/chromium-projects/`

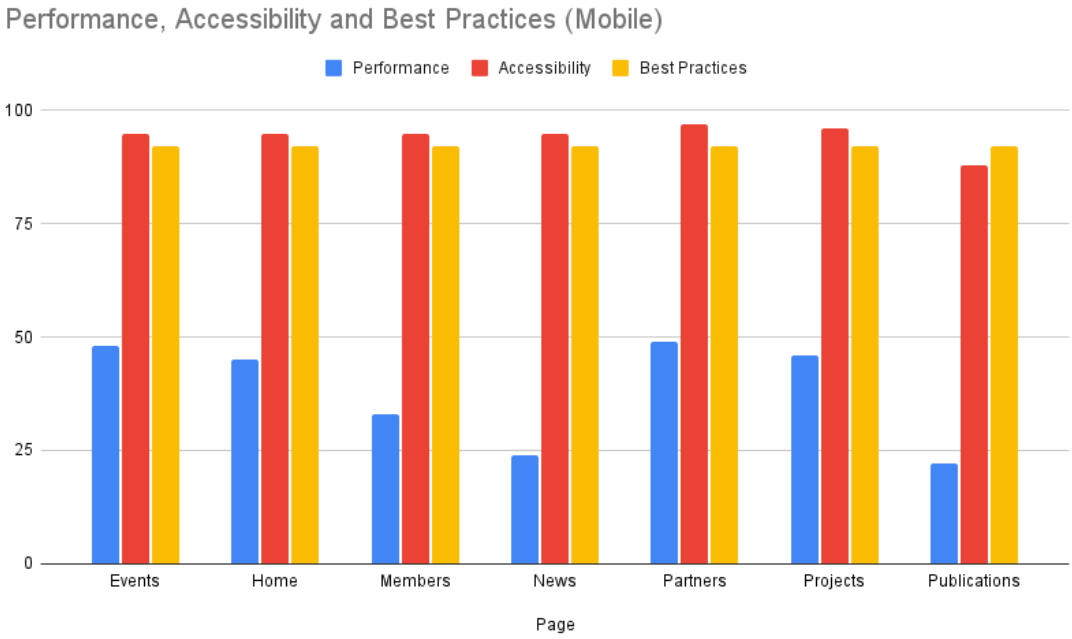


Figure 5.2: Lighthouse audit results (mobile view).

Table 5.1: Summarised audit reports.

View	Aggregation	Performance	Accessibility	Best Practices
Desktop	Min	57	81	92
	Max	75	92	92
	Average	67.43	88.86	92
Mobile	Min	22	88	92
	Max	49	97	92
	Average	38.14	94.43	92

As we can see by these results, "Performance" is the parameter with the lowest score in both views. This parameter's score is especially low in the mobile view because Lighthouse emulates a mobile network connection[3], thus aggravating the already low-performance score. Regarding "Accessibility" and "Best Practices", both views scored a high score (around 90 points), which is very good. The analysis of these results allowed us to conclude that we should focus on improving the website's performance.

Other helpful features from this tool are recommendations to improve the scores obtained. In this case, to improve the performance score, some of the suggestions were:

1. To serve images in formats such as WebP and AVIF, as these provide better compression.

[3]According to: `https://developers.google.com/speed/docs/insights/v5/about`

2. Reduce network payload sizes, as high payload sizes are correlated with long load times.

3. Use HTTP/2, as header compression reduces the size of HTTP packets, thus decreasing load times[4].

4. Increase cache lifetime, to speed up page loads after the first time.

5.1.3 Technology analysis

This section contains a very brief overview of each of the technologies being used on the team's web application and our opinion on whether it should be replaced or not, considering the objectives of this dissertation project and the team's needs.

- **NGINX**: It is open-source software that offers solutions for, among others, web serving, reverse proxying, caching and load balancing. Specifically as a web server, it aims to be the most performant amongst its competitors [32]. Since these are its use cases in the team's application, we considered this technology choice adequate for the future. Furthermore, it supports WebSockets [32], which are used in the interactive demonstrations.

- **Django**: It is an open-source Python web framework that provides solutions for common web development tasks, such as user authentication and content administration. It aims to shorten the development time of web applications while still providing various security measures against common vulnerabilities [12]. An alternative to this technology, also written in Python but with a different vision, is Flask[5]. It aims to be simple by default but extensible (with extensions) [15]. Since the team's application was already built using Django and its characteristics are adequate for this dissertation project, we did not consider changing it.

- **Vue.js**: It is a JavaScript framework for building user interfaces. Its two core features are reactivity (tracks JavaScript state changes and efficiently updates the page) and declarative rendering (supports dynamic HTML template output based on JavaScript state) [43]. Although currently we do not believe that there is any compelling reason to change this technology, based on its current use case, other front-end frameworks such as React.js[6] and Angular[7] are possible alternatives.

- **SQLite**: It is an embedded SQL database engine that, unlike other SQL databases, reads and writes directly to ordinary disk files. A complete SQLite database and its tables, indices, triggers and views are stored in a single file. SQLite is not directly comparable to other DBMS such as MySQL or PostgreSQL because its focus is not scalability, concurrency, centralisation, or control. Instead, it provides a solution to store data locally for individual

[4] HTTP1.1 and HTTP2.0 comparison: `https://www.cloudflare.com/learning/performance/http2-vs-http1.1/`
[5] Flask: `https://flask.palletsprojects.com/en/2.0.x/`
[6] React.js: `https://reactjs.org/`
[7] Angular: `https://angular.io/`

applications and devices [41]. Based on these factors, we believe there are more adequate technologies (e.g. MongoDB, PostgreSQL) for the team's application and, thus, we decided to change it.

5.2 Interviews with team members

Interviews with the team members were conducted to gather requirements and feature suggestions. The interview had three main sections: an evaluation of their perception of each of the usability properties (efficiency, effectiveness and satisfaction), problem enumeration and feature suggestions. Since the first section was a preliminary evaluation of the application's usability that relied entirely on the participants' opinion, it is very subjective and, therefore, will not be considered in this phase.

There were a total of 8 responses, 5 of which were filled in along the respective individual interview and 3 of which were posteriorly answered through a form link. The full interview can be conferred in appendix A.1.

5.2.1 Problem enumeration

In the problem section, interviewees were asked to enumerate factors that negatively affect the visitors' experience and other user interface issues. It was also requested for page administrators to point out back-end-related problems.

Regarding the first question, five out of six answers stated that "slowness", that is, the high page loading times was the most impactful issue to the visitors' experience. Other problems were also pointed out, such as the lack of visual feedback while loading and the home page layout.

As for the second question, the answers were more diverse. The most common issue brought up by multiple administrators was the necessity to reboot the application in order to fetch the new team members' publications from Authenticus[8], as opposed to periodically updating them, as it was initially intended. Another aspect pointed out was that the database was simply a file (SQLite database), whose limitations make it inadequate for the team's use case. Yet another negative characteristic appointed was the need to recur to the administration panel in order for team members to change their personal information and credentials. Finally, it was also appointed some difficulties that the administration panel poses to the usual page administration workflow.

The whole problem enumeration section, as well as its results, can be consulted in appendix A.1.2.2.

5.2.2 Feature suggestions

In the feature suggestion section, interviewees were asked to suggest new functionalities they deemed relevant to the application. Among the many answers were ordering and filtering in the

[8] Authenticus: `https://www.authenticus.pt/pt`

news page, publication searching, implementing a personal area for the team members, and reformulating the team's newsletter. The complete list can be conferred in appendix A.15.

5.3 Interactive demonstrations

Since computer vision and pattern recognition algorithms tend to be computationally intensive, and not every visitor's device may be able to run them properly, the execution should occur on the server-side, and its output(s) should be transmitted to the client-side with low delay. Finally, the user should also be able to interact with the algorithm parameters through a user interface in order to be able to play and test the algorithm as they may desire.

In order to better understand the types of inputs and outputs the team's application should support, we developed a small questionnaire for the team members to answer (see appendix A.2.1). The questionnaire asked the respondent to answer based on experience from previously developed projects. There were a total of ten responses.

The obtained results (see appendix A.2.2) show that images are the most important types of files to support, followed by text and videos, both for inputs and outputs. One of the ten respondents also revealed some interest in supporting audio and structured data (e.g. CSV files). Finally, 90% of the respondents considered it relevant to display the execution time in the interface.

5.4 Summary

Throughout this chapter, many problems and suggestions were appointed. Due to the large variety of results from the various requirement gathering moments previously described, we had to select the most pertinent ones, considering both the duration of this dissertation project and the purpose of the team's web application. The following enumeration contains the requirements we deemed to be the most relevant, organised by their respective objective:

Switch to an adequate DBMS

1. Change the DBMS to a more adequate one, capable of serving multiple concurrent website visitors.

Improve usability

1. Reduce page loading times so that the visitors' experience and efficiency while performing a given action are not impaired.
2. Add visual feedback to the website's pages so visitors can undoubtedly perceive that their interaction with the interface had an effect.

3. Enhance the homepage layout so that this section's initial space gets better employed with the most recent news about the team members.
4. Add a way to search by publications so that a visitor may quickly find a publication by a given team member.
5. Allow the filtering and ordering of news.

Interactive algorithm demonstrations

1. Execute algorithms on the server-side so that the algorithm's execution does not depend on the visitor's device.
2. Transmit execution to the client-side.
3. Support images, text, and videos both as inputs and outputs to properly showcase most of the algorithms developed by the team members.
4. List all available algorithm demonstrations in the team's application.
5. Allow visitors to tweak the algorithm's execution parameters through the user interface so that they can test it as they may desire.
6. Display the algorithm's execution time so visitors may have a rough estimation of its performance.

Chapter 6

Development and Implementation

This chapter describes the process of implementation of the requirements of this dissertation. It includes not only the implementation results, but also the reasoning behind the most critical decisions taken, as well as critical reflections about the solutions applied.

6.1 Switching DBMS

This section illustrates how the DBMS used in the team's web application was changed (see subsection 5.1.3). It starts by detailing how the new DBMS was chosen, then explains how it was implemented and how the data from the previous database was transferred into the new one. Finally, it concludes with considerations of this process and the resulting system architecture.

6.1.1 Choice process

In order to make an informed decision about the most appropriate DBMS to switch to, we took into consideration a variety of factors, some described in subsection 3.2.2, and others that were raised during the implementation:

Type and amount of data

The team's application data was composed of 4.3 MB of structured data (total size of the SQLite database) and 64.9 MB of images (and a few documents). Because images are stored in the server's filesystem, only the database size may affect our choice. Since the amount of data in the database is small, it currently imposes no restrictions on the type of database to use [19]. Even considering an overestimated annual growth ten times greater than the one observed since the first release in August 2019 ($10 * \frac{4.3\ MB}{2.5\ years} = 17.2$ MB / per year) would not represent a significant amount of data enough to affect our decision, for many years. On the other hand, when we consider the entire database schema (see appendix A.3), we can see that it has a considerable amount of relationships among the various tables, which is undesirable for some non-relational DBMS [19]. Choosing a

non-relational DBMS would still be possible, but in that case, it would be advisable to refactor the database schema.

Transaction amount and frequency

A grand majority of the application's use cases consist of consulting information and, therefore, most of the database operations are read operations. Furthermore, database updates and insertions are not frequent (only when an administrator member wants to add content) and are usually done sequentially. Another factor we considered was that, unlike a bank's database, whose data must be consistent all the time and operations should be atomic to avoid partial executions on failure, our data is non-critical. Therefore, we concluded that a DBMS with ACID transaction support could be helpful, but it is not an absolute necessity.

Cost and hardware limitations

While meeting with the application administrators, it was decided that we should keep the team's application costs as low as possible.

Since the new version of the application is supposed to replace the current one, we already had a place to deploy it to. The application runs on a server, inside a virtual machine, whose allocated hardware is specified in Table 6.1.

Table 6.1: Virtual machine characteristics.

CPU	Intel(R) Xeon(R) CPU E5-2643 v4 @3.40GHz (4 cores available)
Disk Space	64 GB
RAM	4 GB

Considering these hardware characteristics, we recognise that 64 GB is not much space, especially considering the number of images the application uses (that will only increase over time due to the expository nature of the application). However, since we are working with a virtual machine, we can simply allocate more space (provided that the server has available disk space) for the application, so we did not weight this space limitation in our decision.

Another implication of that decision is that the chosen DBMS should be free of charge, thus excluding Oracle and other paid proprietary DBMS that require payment to be used.

Other restrictions

Some of the software that composes the team's application, namely Django and Wagtail[1], only support some DBMS, thus restricting our possibilities. Django has official support for SQLite,

[1] Wagtail: `https://wagtail.org/`

MySQL, PostgreSQL and Oracle[2], and, through the use of third-party backends, may also work with other DBMS such as MongoDB[3] and Microsoft SQL Server[4], among others.

On the other hand, Wagtail, the content management system used by the team members to manage their application's database, is more restrictive and only supports PostgreSQL, MySQL and SQLite[5].

Since changing any of these components is not practicable due to requiring a significant code refactor and being out of scope for this dissertation project, we had to cope with these imposed constraints.

Performance

The last and, in this case, the most relevant distinction factor amongst the DBMS is its performance. The term "performance" is rather vague in the sense that it might refer to different measures such as average throughput (operations per second) or average time taken in a given operation (Read, Write, Update, Delete), however, since one of the objectives of this dissertation project is to improve page loading times, we decided to measure "performance" as the average time taken to complete the queries required to load the main website's pages.

Initially, we considered using a benchmarking tool such as benchbase[6] or YCSB[7] to evaluate each of the possible DBMS. By choosing a workload that relies mostly on Read operations (similar to the normal usage of the website), we could estimate the capabilities of said DBMS for our use case. However, thanks to the level of abstraction that Django provides when working with databases, we were able to easily try every DBMS not excluded up to this point (PostgreSQL, MySQL) and therefore measure the time taken to complete the pages' queries. Since MySQL had two supported versions (v5.7 and v8.0), considered interesting to include a comparison of the performance of both. The application was running locally, and the measurements were carried out in triplicate for each of the team's main pages, using django-silk[8] (profiling tool for Django). Tables 6.2 and 6.3 show, respectively, the average time used querying the database and the average overall time to fulfil a request (back-end processing time included), for the PostgreSQL, MySQL5.7 and MySQL8.0 DBMS. Some values on the MySQL5.7 column are missing because django-silk did not measure the request times on some pages.

[2]Django officially supported databases: `https://docs.djangoproject.com/en/2.1/ref/databases/`
[3]Djongo: `https://github.com/doableware/djongo`
[4]Other third-party database backends: `https://docs.djangoproject.com/en/2.1/ref/databases/#using-a-3rd-party-database-backend`
[5]Wagtail compatibility: `https://github.com/wagtail/wagtail#compatibility`
[6]BenchBase: `https://github.com/cmu-db/benchbase`
[7]YCSB: `https://ycsb.site`
[8]Django-silk: `https://github.com/jazzband/django-silk`

Table 6.2: DBMS comparison - Query time (best result in bold).

Query Time (ms)	**DBMS**	MySQL 8.0	MySQL 5.7	PostgreSQL
Page				
VCMI Home		**77.67**	95.00	85.67
Members		741.67	-	**711.00**
Projects		21.67	**17.67**	**17.67**
Publications		1,752.33	-	**1,702.33**
Partners		24.67	23.33	**20.67**
News		676.33	-	**664.67**
Events		10.33	**9.00**	**9.00**

Regarding the time used on queries, we can observe that MySQL8.0 was the fastest on the Home page, MySQL5.7 and PostgreSQL on the Projects and Events pages and PostgreSQL on the rest of the pages' queries, thus making it the best performant DBMS, in our use case. In most requests, the query times did not significantly deviate from the other DBMS, except on the Publications and Members pages, where PostgreSQL is approximately 50ms and 30ms, respectively, faster than the second-best DBMS. One factor that impacted these results is the fact that Django (or its database adapter package) would generate slightly fewer queries for the same page requests, when compared to MySQL.

Table 6.3: DBMS comparison - Overall time (best result in bold).

Overall Time (ms)	**DBMS**	MySQL 8.0	MySQL 5.7	PostgreSQL
Page				
VCMI Home		300.00	339.67	**241.33**
Members		2,396.33	-	**1,917.33**
Projects		131.33	102.00	**59.00**
Publications		5,488.00	-	**4,618.33**
Partners		141.67	83.67	**65.00**
News		2,087.67	-	**1,797.00**
Events		49.67	37.33	**34.33**

Regarding the overall time used to fulfil the page requests, which includes both the database queries and the data processing on the back-end, we can see that the team's application using PostgreSQL as its DBMS presents the lowest average overall time, regardless of the page. Here we see a more significant difference among implementations that, in some cases, reached approximately 870ms. Since, in every implementation, the schema, data and indexes were kept the same, the only reason we found for these deviations is how Django uses the required database adapters ("psycopg2" for PostgreSQL and "mysqlclient" for MySQL).

Based on these results, we concluded that PostgreSQL's implementation is the one that offers the best performance while using Django and for our use case.

Once we analysed all of the described factors and considerations, we concluded that PostgreSQL is the most appropriate DBMS for our use case. Although it will increase the cost of running the application slightly (due to being a separate server process constantly running, as opposed to a file), any other DBMS running on the server would have the same implication. Another downside of our solution is that the whole database (not just the application's data) now occupies 108.1 MB (SQLite database was 4.3 MB), thus increasing the disk space use. On the other hand, choosing this DBMS brought us some advantages: storing all the structured data while not requiring changes to the original data structure, Django and Wagtail support and the best performance compared to the other possible DBMS.

6.1.2 Implementation and data transfer

The first step of this conversion process was to export all the original database's data. Django provides a command to export data as a JSON file. Listing 6.1 contains a simplification of the command used to export the database's data and Listing 6.2 a portion of the resulting file.

```
django-admin.py dumpdata --natural-foreign --indent 6  > data.json
```

Listing 6.1: Export data command.

```
{
     "model": "pages.partnerpage",
     "pk": 208,
     "fields": {
          "name": "Faculdade de Engenharia da Universidade do Porto",
          "external_url": "https://sigarra.up.pt/feup/en/WEB_PAGE.INICIAL",
          "partner_type": 205,
          "vcmi": true,
          "breast": false
     }
},
```

Listing 6.2: Exported data portion.

Once the data was exported, we proceeded to create the PostgreSQL database. Like other services of the application (front-ends and webserver), we created a Dockerfile based on the official PostgreSQL image[9] and configured a service using it. Listing 6.3 shows the service configuration added to the docker-compose file.

[9]Postgres on docker hub: `https://hub.docker.com/_/postgres`

```
db:
    build:
      context: ./postgresql
      dockerfile: Dockerfile
      args:
        ROOT_USER_PASSWORD: $ROOT_USER_PASSWORD
        VCMIDB_USER_PASSWORD: $VCMIDB_USER_PASSWORD
    container_name: vcmi_postgres_db
    networks:
      - network-dev
    ports:
      - '5432:5432'
    environment:
      POSTGRES_PASSWORD: $POSTGRES_PASSWORD
      POSTGRES_USER: $POSTGRES_USER
      POSTGRES_DB: $POSTGRES_DB
      PGPASSWORD: $PGPASSWORD
    volumes:
      - ./postgresql/data:/var/lib/postgresql/data
      - ./postgresql/vcmibreast-initdb.sql:/docker-entrypoint-initdb.d/vcmibreast-
          initdb.sql
```

Listing 6.3: PostgreSQL service (db) added to docker-compose.yml.

As we can see by the above listing, the "db" service image is built using the Dockerfile in the "./postgresql" directory, with two arguments whose values come from an environment file. Once the image is instanced, the resulting container is named "vcmi_postgres_db". This container's "/var/lib/postgresql/data" directory is mounted to the "./postgresql/data" directory of the host machine, and changes made to any of these reflect in the other. This configuration is used so that all the databases' data (written from inside the container) persists in the host filesystem and therefore is preserved, even when the container is removed. The four environment variables described above, which are used to configure PostgreSQL, are set inside the container, and their values come from the same environment file as the arguments. The container is part of the network "network-dev", like the rest of the application's services, so that they can communicate with each other, and the port 5432 inside the container is mapped to the port 5432 on the host machine. This mapping is required in order for the container's application to be accessible to the host machine through this port.

The next step consisted of installing the database adapter package "psycopg2" and changing Django's default database from SQLite to Postgres, as shown in Listing 6.4.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'vcmibreast',
        'USER': '<User>',
        'PASSWORD': '<Password>',
        'HOST': 'db',
        'PORT': '5432',
    }
}
```

Listing 6.4: Django's default database configuration.

After both of these requirements were met, we started the application. Django automatically created the database tables according to the models defined in the various Python files. At the same time, Wagtail generated some page data, which proved problematic for loading the original database's data. After clearing the data from the "page" table, we loaded the previously exported JSON file using the command in Listing 6.5.

```
django-admin.py loaddata data.json
```

Listing 6.5: Load data command.

After the loading process was finished, the team's application was fully functional, and thus our implementation was complete.

6.1.3 Database benchmark

To estimate the performance of our DBMS with the current hardware, we used pgbench[10], a program for running benchmark tests on PostgreSQL. It runs multiple concurrent SQL commands and calculates the average transaction rate (average number of transactions executed per second). The commands executed depend on the script used. Pgbench has three built-in scripts: "tpcb-like", "simple-update" and "select-only". The first consists of three "Update" operations, one "Select" operation and one "Insert" operation. The second executes one of each operation, and the last one only executes "Select" statements.

Before running the scripts, pgbench requires initialisation of the database, i.e., creating the required tables and populating them with data. We created a new, dedicated database named "benchmark" to preserve the other data. We used a scale of 64, which leads to a total database size of around 1GB. Listing 6.6 shows the command executed for this purpose.

```
pgbench -U test -i -s 64 benchmark
```

Listing 6.6: Pgbench initialisation command.

[10]Pgbench: `https://www.postgresql.org/docs/current/pgbench.html`

We chose to run the "tpcb-like" script because it is the default and it has the most diverse set of operations among the available ones, and the "select-only" script, as it is the most similar to the predominant use-case of the team's application (consulting information). Listing 6.7 displays the issued commands to run the benchmarks.

```
1 pgbench -U test -c <N_Clients> -j <N_Threads> -b select-only -C -T 60 benchmark
2 pgbench -U test -c <N_Clients> -j <N_Threads> -b tpcb-like -C -T 60 benchmark
```

Listing 6.7: Pgbench benchmark commands.

The "c" flag we can see in the previous commands is the number of simulated clients, i.e., the number o concurrent database sessions, and the "j" flag refers to the number of worker threads used by pgbench to split the clients. We ran these commands for multiple client and thread values. It is also worth noting that the "C" makes it so that a connection is not reused, i.e., it is re-established for each transaction executed rather than once per client. Since this dramatically impacts the results (high connection overhead), we ran both commands with and without it. Table 6.4 contains the best scores obtained for each script, with and without connection reuse, in transactions per second (TPS).

Table 6.4: Database benchmark results (TPS).

Script	With connection reuse	Without connection reuse
select-only	27522.28	811.51
tpcb-like	3519.47	581.01

These results represent an estimation of our implementation's capabilities. More specifically for our most common use case, we can expect our DBMS, with the hardware described in Table 6.1 to be able to resolve up to 27522 "Select" operations per second across multiple database sessions.

6.1.4 Conclusions and considerations

This transition concludes the second objective of this dissertation project. We studied both the requirements and restrictions of our project, successfully switched the DBMS to PostgreSQL, and proved it can handle multiple concurrent requests.

We believe that one of the advantages of keeping Django as the back-end framework of the application was shown during this phase, as it not only allowed us to quickly try multiple DBMS and test them, but also facilitated the migration of all the application data to the new database.

One factor to keep in mind about our implementation is the fact that PostgreSQL is an object-relational DBMS, thus, vertically scalable. In our case, this implies that in the event of a sufficiently high and consistent load increase (which could be due to a significant popularity increase

of the team's application), the virtual machine's allocated hardware (or even the servers') must be upgraded.

The structural relationship amongst the components of the application after this phase is described in the component diagram (see Figure 6.1).

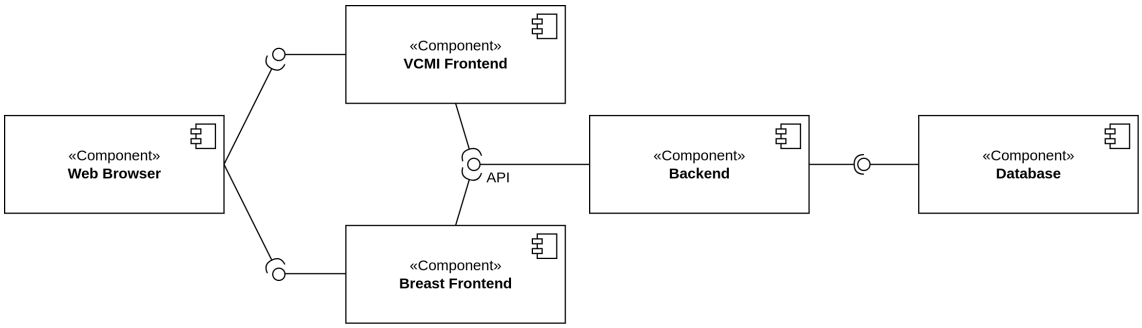


Figure 6.1: Application's component diagram after database implementation.

6.2 Usability study

This section follows the methodology described in subsection 4.2.1. It starts by analysing the results of the first usability evaluation and then proceeds to describe the implemented solutions for the gathered usability-related requirements. Finally, it compares the results of the second usability evaluation to the first one, analysing the impact of the implemented changes on the website's usability.

6.2.1 First usability evaluation

The usability evaluation performed consisted of two phases. In the first phase, eight simple tasks were given to the tester for them to complete. Both their success (or lack of) in completing a given task, and the time taken to do it were noted to assess the effectiveness and efficiency. The tasks given to the testers were chosen to be diverse and cover the most common use-cases of the team's website. The complete list of tasks was the following:

1. View the list of partners;
2. Read the most recent news;
3. Learn more about the event "MICCAI Hackathon 2021";
4. Find out how to contact the team;
5. View the team's concluded projects and read more about one of them;
6. Go to one of the team members' personal web page and view their publications in the year 2020;

7. Read the conference article “Automatic Augmentation by Hill Climbing”, published in 2019, using the “Publications” tab;

8. Open the page you would go to in order to log in.

In order to assess the usability’s remaining and most subjective attribute - user satisfaction - the SUS method was followed: After finishing the first phase’s tasks, the testers were asked to classify each of the ten SUS’s items (see subsection 2.3.1) in a five-point agreement scale (similarly to a Likert scale) according to what they felt while using the team’s application. The outcome of this method is a score (a number between 0 and 100) calculated by adding the individual scores of the ten questionnaire items and multiplying the obtained value by 2.5. The corresponding score of the odd-numbered items is the scale position (from 1 to 5) minus 1. As for the even-numbered items, their score is 5 minus the selected scale position [22].

There were a total of six participants: three interns, who did never visit the team’s website, and three team members. Although these last three knew about the existence of the VCMI’s website, they stated that they had only used it very punctually and thus were still considered valid testers. It is also worth noting that all of the participants in this first evaluation were either young students or young graduated researchers and therefore are used to navigating through websites daily. The results obtained are summarised in Tables 6.5, 6.6 and 6.7.

Effectiveness

Table 6.5 shows whether or not a tester was able to complete each of the evaluation tasks they were given. A value of 1 means the tester successfully finished the task, and a value of 0 means the tester could not complete it and gave up.

Table 6.5: Task completion, by tester.

Task Completion	Task	1	2	3	4	5	6	7	8
Tester									
1		1	1	1	1	1	1	1	0
2		1	1	1	1	1	1	1	0
3		1	1	1	1	1	1	1	0
4		1	1	1	1	1	1	1	0
5		1	1	1	1	1	1	1	0
6		1	1	1	1	1	1	1	0

Success Rate (%)		100	100	100	100	100	100	100	0

By analysing these results, we can verify that the testers successfully concluded all but the eighth task, thus making the overall effectiveness 87.5%.

The eighth task consisted of opening the login page. Once logged in, a team member may, amongst others, request changes to the website's content. However, at the time of this evaluation, there was no way to access that page through the user interface, except by changing the Uniform Resource Locator (URL). We knew that this task would be difficult to complete without prior knowledge, however, since it is a crucial step to make use of some important functionalities of the team's application, we kept it. From the lack of success in its completion, we decided to add the following usability requirement: "Add a way to access the login page through the user interface".

Efficiency

Table 6.6 exhibits the time taken, in seconds, by every tester in each task. The timer was started right after the tester finished reading the task and stopped either once the task was completed or after they gave up. In case of withdrawal, the time taken is displayed inside parenthesis.

Table 6.6: Time taken in each usability task, by tester.

Time taken (s)	Task	1	2	3	4	5	6	7	8
Tester									
1		2.90	6.07	9.25	3.24	11.37	17.83	41.08	(46.34)
2		4.69	7.72	28.09	4.25	12.36	56.91	29.22	(25.46)
3		2.50	13.97	12.91	2.59	10.90	21.56	37.69	(50.86)
4		5.25	8.82	10.81	83.17	19.31	45.04	37.93	(98.99)
5		3.12	5.70	20.48	30.45	8.98	23.51	41.12	(124.85)
6		4.01	9.79	8.80	65.89	9.56	23.77	50.22	(36.23)
Average		3.75	8.68	15.06	31.60	12.08	31.44	39.54	(63.79)

As we can see by the results obtained, Task 1 had an average completion time of 3.75 seconds, Task 2 of 8.68 seconds, Task 3 of 15.06 seconds, Task 4 of 31.60 seconds, Task 5 of 12.08 seconds, Task 6 of 31.44 seconds and Task 7 of 39.54 seconds. Task 8 was not considered for the efficiency evaluation because none of the testers could complete it. Thus, we observe an average task completion time of approximately 20 seconds.

Most of the completion times are similar across testers in a given task, except for the discrepancy in Task 4, in which half of the testers took much longer than the rest. Faced with this observation, we decided to add the following usability requirement: "Highlight the contacts section so that a visitor can easily find how to contact the VCMI team".

User satisfaction

The summary of all the answers from the SUS's questionnaire can be consulted in appendix A.4. By analysing the testers' perspective on the system, we concluded that, overall, they found

the system simple and easy to use and understand. The answers were mostly favourable, however, one of the six testers pointed out that they found the system inconsistent. We considered that in the next phase's improvements.

Through the usage of the SUS method, we could more objectively analyse the user satisfaction. Table 6.7 summarises the scores calculated from the questionnaire answers.

Table 6.7: User satisfaction evaluation - SUS scores.

Scores	Item	1	2	3	4	5	6	7	8	9	10	Total	SUS Score
Tester													
1		2	2	3	3	2	3	3	3	2	4	27	67.5
2		3	3	3	4	3	4	3	3	3	3	32	80
3		3	3	3	3	2	3	3	3	3	3	29	72.5
4		2	3	3	4	3	4	3	4	3	3	32	80
5		3	4	4	4	3	4	3	3	3	4	35	87.5
6		3	4	3	4	2	1	4	4	3	4	32	80

Final Score (Average)
77.92

As we can observe, the overall SUS score in the first evaluation is 77.92 (out of 100). Even though this is already a good score, we expected to observe an improvement in user satisfaction in the second evaluation.

6.2.2 Usability improvement

In this phase of the usability study, many tasks were conducted in order to fulfil the usability requirements. Their implementation is briefly described below.

6.2.2.1 Reduce page loading times

Improving queries and serializers

One of the most noticeable issues while using the team's application used to be the high page loading times. This issue has been present since the application was first built, but it was not evident for a long time. However, as time went by and the amount of data grew, it became gradually worse until it became apparent that something was wrong.

Utilizing the browser's DevTools (in this case, Firefox's) allowed us to verify that the GET request for the page data often would take a very long time to be fulfilled (in some cases, almost 8 seconds). Furthermore, on pages whose content has images (e.g. "News", "Members", and "Projects"), once the page data was retrieved, multiple subsequent requests would be sent (one for each image on the page) to fetch the image URL in the database (used to display the image to the user). Another observation was the excessive amount of data being returned as response, i.e.,

a part of the data obtained was not being used by the front-end and, therefore, was making this process slower without any reason. All these issues negatively impacted the efficiency of every task that involved these pages.

In order to understand the cause of our first realization, we analysed these requests using django-silk[8]. With this tool, we could view the number of queries that the API request originated and the SQL code executed in each one. Table 6.8 contains the number of queries that were being executed as a consequence of a single GET request to retrieve the page data for each of the main pages.

Table 6.8: Number queries executed to fetch page data (before changes).

	Home	**Members**	**Projects**	**Publications**	**Partners**	**News**	**Events**
Number queries	94	991	31	2314	31	840	12

Based on these results, and considering the simple nature of these requests, we can see that an excessive number of queries were being executed. By analysing the queries' SQL code, we observed that many were identical, with the difference being the id of the page being selected, i.e., the page's items (or sub-pages) were being retrieved one by one. Furthermore, if these items were related to other tables (such as the image table), they would generate even more queries and thus flood the DBMS with requests. The cause of this undesirable behaviour was in the way that said data was being fetched.

Django's object-relational mapper (ORM) maps its model classes to the database and mediates queries[11]. A model contains the essential fields and behaviours of the data we want to store in our database[12]. Once a model is created, Django allows us to treat database tables (and rows) as objects by providing a database-abstraction API that lets us perform insert, read, update and delete operations. In order to retrieve (read) data from the database, a QuerySet must be constructed[13]. A QuerySet represents a collection of objects from the database, and operations such as filtering and slicing may be applied to it. One crucial detail about this structure is that it aims to reduce the number of queries executed by only requesting and retrieving the information from the database once the QuerySet is evaluated (e.g. once it is iterated, sliced, converted to a list)[14]. The improper creation and use of QuerySets led us to experience the exact opposite effect and therefore was the root of most of our performance problems. Listing 6.8 shows the QuerySet that was being used to fetch all the team members (required to load the "Members" page), and Listing 6.9 is the member serializer whose function is to prepare the members' data to be returned as a JSON.

[11] Django ORM: `https://docs.djangoproject.com/en/2.1/releases/1.0/#an-improved-orm`
[12] Model: `https://docs.djangoproject.com/en/2.1/topics/db/models/#module-django.db.models`
[13] Making queries with Django: `https://docs.djangoproject.com/en/2.1/topics/db/queries/#making-queries`
[14] QuerySet evaluation: `https://docs.djangoproject.com/en/2.1/ref/models/querysets/#when-querysets-are-evaluated`

```
MemberPage.objects.live().child_of(self)
```

Listing 6.8: Members QuerySet.

```
1  class MemberSerializedField(Field):
2      """A custom serializer used in Wagtails v2 API."""
3
4      def to_representation(self, members):
5          """Return member data"""
6          data = []
7          for member in members:
8              data.append(
9                  {
10                     "id": member.slug,
11                     "name": member.name,
12                     "photo": member.photo.id,
13                     "type": member.member_type.description,
14                     "years": member.filter_years,
15                     "position": member.position,
16                     "vcmi": member.vcmi,
17                     "breast": member.breast,
18                 }
19             )
20         return data
```

Listing 6.9: Members serializer.

As we can see in this code section, a group of members is iterated. Even though every member is fetched in a single query, this code also accesses the "photo", "member_type", and "filter_years" properties (lines 12 to 14), whose values depend on other tables, thus triggering three other queries for every member iterated. Furthermore, since members and publications have a many-to-many relationship (through an intermediate relationship table), even more queries are issued to build the "years" list.

The solution we found for this issue consisted of improving the QuerySets so that they select the related data when they are executed (see Listing 6.10).

```
MemberPage.objects.live().child_of(self).select_related("member_type", "photo")
```

Listing 6.10: Members QuerySet (optimised).

By calling "select_related"[15] in a QuerySet, we get a single, more complex query that will pre-populate all the member's data and related fields ("photo" and "member_type"), thus eliminating

[15] Select related:
https://docs.djangoproject.com/en/2.1/ref/models/querysets/#select-related

the need for more queries to follow the foreign keys. Particularly in the case of the "Members" page, this performance booster brought the number of queries from 991 down to 12. Although very useful, this method has the limitation of not working for many-to-many relationships, therefore, for these cases, we found another solution: the "prefetch_related"[16] method. This method, just like the previous one, aims to reduce the number o queries that reach the database, however, instead of joining the SQL queries, this one does a separate lookup for each relationship and performs the joins in Python. Once the data is prefetched, it gets cached, thus eliminating the need to perform more queries to access related-object data.

By carefully analysing each of the page's QuerySets, applying the most appropriate of these methods, and some other small logic changes, we reduced the number of queries being executed significantly. Table 6.9 shows the number of queries required to load each of the main pages after these changes.

Table 6.9: Number queries executed to fetch page data (after changes).

	Home	**Members**	**Projects**	**Publications**	**Partners**	**News**	**Events**
Number queries	19	12	12	14	12	14	12

As we can see, these QuerySet optimisations had a massive impact on the number of queries hitting the database (in the case of the "Publications" page, 2300 fewer queries). After these changes, the team's application became noticeably faster and more responsive.

The next step in our process of improving the database queries was related to images. As previously stated, after a page's data was retrieved, multiple independent queries to fetch its images by their id were being spawned by each of the front-end's components. This behaviour is especially wasteful because the query executed to fetch the page's data already performed a join with the image table to return the image id.

Our solution consisted of updating the page serializers so that they return not only the image's id but also their URL. This way, we eliminated the subsequent requests simply by passing the returned URLs to the image components (on the front-end), thus allowing them to display the images without extra overhead.

Finally, regarding the issue of the unused data, we analysed every serializer (back-end) and the data used by the respective page components (front-end) and, with these, we determined what fields were not being used and removed them from the serializer. Since some serializers were being shared amongst multiple pages, we had to create new dedicated ones for specific pages.

Once all of these issues were addressed, every page request became more efficient and returned only the required data to display it.

Serving images in the WebP format

WebP is a method of compression released by Google that can be used on images. This modern

[16]Prefetch related:
`https://docs.djangoproject.com/en/2.1/ref/models/querysets/#prefetch-related`

open-source image format aims to make the web faster by being smaller and better-looking than other standard image formats, such as Portable Network Graphics (PNG) and Joint Photographic Experts Group (JPEG)[17].

As most of the application's images were either PNG or JPEG files, we saw an opportunity to reduce the disk space occupied by them and make some pages (those that have images) load slightly faster (due to less data being transferred). This change would also be beneficial for visitors with a poor internet connection.

To make this transition, we used Converseen[18], a free batch image processor to convert all the application's images (served by the back-end) to the WebP format. Although we converted all the images in the filesystem, the image paths stored in the database were still relative to previous ones and had to be updated accordingly. Since the new images' paths simply have a different extension, we created and used the SQL script in Listing 6.11 to perform the path update on all the image rows. We ran it once for JPEG files and another for PNG ones. After this step, the application was fully functional and served all images in the WebP format.

```
1  UPDATE
2      wagtailimages_image
3  SET
4      file = oldfiles.filename
5  FROM
6      (
7          SELECT
8              id,
9              file,
10             concat(
11                 substring(file, 0, position('.JPEG' in file)),
12                 '.webp'
13             ) filename
14         FROM
15             wagtailimages_image
16         where
17             file LIKE '%.JPEG'
18     ) AS oldfiles
19 WHERE
20     wagtailimages_image.id = oldfiles.id;
```

Listing 6.11: Script used to rename the images.

Text compression and image caching

Compressing text responses reduces the size of the transmitted data and, therefore, may make transmitting the necessary data quicker. Although compressing occurs at runtime, thus adding

[17] WebP format: https://developers.google.com/speed/webp/faq#what_is_webp_why_should_i_use_it

[18] Converseen: https://converseen.fasterland.net/

some extra processing overhead, it may still be advantageous[19], especially for visitors with a poor internet connection or using mobile data.

In order to start compressing text, we simply had to modify NGINX's configuration file so that it compresses the types of files we specify. Listing 6.12 contains the configuration we used.

```
1  gzip on;
2  gzip_comp_level 5;
3  gzip_min_length 1000;
4  gzip_types
5      text/css
6      text/javascript
7      application/javascript
8      application/json
9      text/xml
10     text/plain;
```

Listing 6.12: NGINX text compression configuration.

As we can see, we enabled Gzip[20] compression for Cascading Style Sheets (CSS), JavaScript, JSON, XML and plain text files whose length is more than 1000 bytes. The compression level was set to 5 (levels range from 1 to 9) as it is a middle ground between compression and processing power (a higher compression level leads to more overhead).

Another minor tweak for improving the performance of our application was serving static assets with an efficient cache policy. It simply consists of informing the browser, through a header, to keep a given resource cached for a given amount of time[21]. This way, subsequent requests use the locally stored copy, thus reducing overhead and loading faster.

To implement such a change, we only had to add the line 10 in Listing 6.13 to NGINX's configuration file, more specifically, add it to the "/media" location block as it defines the rules for accessing every image stored in the "media" folder, on the back-end. This line makes it so that cached images are valid for up to 1 year.

```
1  location /media {
2      proxy_pass http://backend:8000/media;
3      proxy_http_version 1.1;
4      proxy_set_header Upgrade $http_upgrade;
5      proxy_set_header Connection 'upgrade';
6      proxy_set_header Host $host;
7      proxy_cache_bypass $http_upgrade;
8
9      # Cache media files
10     expires 1y;
11 }
```

Listing 6.13: NGINX image caching configuration.

[19] Text compression: `https://docs.nginx.com/nginx/admin-guide/web-server/compression/#introduction`

[20] Gzip: `https://www.gnu.org/software/gzip/`

[21] Caching policy `https://web.dev/uses-long-cache-ttl/`

Publication and article pagination

A majority of the team's application main pages have the function of listing content. This content may be members, projects, publications, partners, articles (i.e., the "News" page) or even events. Although some have only a few items, others have more content, such as the "News" and "Publications" pages. Since these pages would always load and display every entry, they would become vertically extensive, thus making it cumbersome to reach the end. Although still not noticeable at this scale, this behaviour is also problematic because as the number of articles and publications grows, their respective pages would require progressively more time to load and consume more resources on the visitors' machine. To solve these problems, we decided to paginate the "News" and "Publications" pages.

We believe that visitors rarely want to view all of the articles ever published on the application, but rather the most recent ones, therefore, only loading some of the most recent ones by default seemed fitting. As for the "Publications" page, we realised that it is more common for a visitor to access older publications than to access older news. However, we still think that loading 500+ publications (as of the time of writing) by default would be wasteful, especially if the visitor simply wants to check if there are new publications. Since publication searching was planned, we still decided to paginate by default, and if the visitor were to search or apply a filter, the complete list of results would be shown (not paginated).

Initially, we tried to implement this item organisation method by using Wagtail's API[22] to fetch content in a paginated way. However, these requests would, once again, spawn several unoptimised database queries, thus taking a considerable amount of time to be completed. Faced with this limitation, we opted to create a new dedicated API route to fetch content and paginate it using Django REST framework (DRF). This approach allowed us to define the queries executed and choose the pagination method and size for each case.

DRF supports three types of pagination[23]: "PageNumberPagination", "LimitOffsetPagination", and "CursorPagination". The first takes a page number from the request's query parameters and returns the requested page of items (of predefined size). The second requires a "limit" value that defines the number of items requested (or page size) and an "offset" value that determines the number of items to be skipped. The last one uses a "cursor" indicator (sequence of characters) that may be used to page through the result set. It only allows moving forward or back on the results list (no page skipping). On the other hand, the two first ones tend to slow down as the size of the dataset (number of rows in the table) increases.

Since the "News" and "Publications" tables are the ones with the most number of rows and the ones that grow the most in the team's application, we decided to use the third pagination type, cursor-based pagination[24]. This pagination type is the most future-proof of the three because its performance does not decline as the dataset grows. Although advantageous from a long term

[22]Fetch content using Wagtail's API:
`https://docs.wagtail.org/en/v2.7.1/advanced_topics/api/v2/usage.html#pagination`
[23]DRF Pagination: `https://www.django-rest-framework.org/api-guide/pagination/`
[24]Cursor-based pagination:
`https://www.django-rest-framework.org/api-guide/pagination/#cursorpagination`

performance standpoint, this pagination method also requires items to be ordered by a unique, unchanging field.

The first step of our implementation consisted of setting the default type of pagination and pagination size for DRF inside our Django configuration file, as Listing 6.14 shows.

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.CursorPagination',
    'PAGE_SIZE': 20,
}
```

Listing 6.14: DRF default pagination configuration.

Next, we started creating our API endpoints (APIViews and routes). First, we created classes inheriting from "CursorPagination" to better configure pagination for each case (one for articles and the other for publications). Listing 6.15 shows the configuration we used for the "News" page's items.

```
class NewsIndexPagination(CursorPagination):
    cursor_query_param = 'c'
    ordering = ('-news_date', '-first_published_at', '-id')
```

Listing 6.15: Individual pagination configuration.

As we can see, in this configuration, we are setting the cursor name to "c" and setting the pagination's ordering to be descendant and prioritise the "news_date" field, followed by the "first_published_at" field, and finally, the article "id" field. In order to implement the APIView (in charge of processing and handling the API requests), we created classes inheriting from the class "ListAPIView". We specified the class serializer and the pagination class and overrode the "get" method inside these classes. By overriding the "get" method, we can define which code is executed upon receiving a GET request. Listing 6.16 contains a simplified version of the "PaginatedNews" APIView.

```python
class PaginatedNews(ListAPIView):
    serializer_class = ArticleListSerializedField()
    pagination_class = NewsIndexPagination

    def get(self, request):
        # Request Validation (...)

        # There is only one NewsIndexPage
        index_page = NewsIndexPage.objects.first()

        NewsPage.objects.child_of(index_page).live() \
            .filter(vcmi=request.query_params.get('vcmi')) \
            .prefetch_related(
                "news_image", "associated_project", "
                    article_member_relationship__member"
            )

        page = self.paginate_queryset(queryset)
        serialized_data = self.get_serializer_class().to_representation(page)
        return self.get_paginated_response(serialized_data)
```

Listing 6.16: Simplified "PaginatedNews" APIView.

Once these were concluded, we assigned a unique URI to the created APIViews, so that once a request is issued to the corresponding URI, it is routed to the right APIView. In this case, only GET requests are available because we only overrode the "get" method. If it passes the validation, the response obtained should be similarly structured to Listing 6.17.

```
{
  "next": "http://localhost/api/v2/paginations/news/?c=bz0xJnA9MjAyMi0wNC0xMQ%3D%3D
      &vcmi=1",
  "previous": null,
  "results": [(20 results...)]
}
```

Listing 6.17: Example paginated data API response.

As we can see, the response contains the most recent 20 articles (omitted to save space), accessible through the "results" property. Furthermore, property "next" contains the URI to access the following page. Note that this contains the "cursor" value of the following page specified next to the query parameter "c". Finally, the "previous" value is null because we requested the first page, however, it follows the same logic as the "next" property and is used to view the previous page of articles.

Since the back-end implementation was complete, we only had to apply this pagination logic to the respective front-end pages. We changed the API calls so that requests used the newly created

endpoints and updated the page construction logic to use the new response structure. At this point, the "News" page would display the 20 newest articles. To allow visitors to view more articles, we added a button that requests the next page using the URI provided by the "next" property. Upon receiving the response, the new items are appended to the page. This process may be repeated until there are no more articles.

6.2.2.2 Team's suggestions

Searching, sorting and filtering

When visitors of the team's application wanted to read a given publication, they would need to open the "Publications" page, and either use the browser's "find in page" functionality or scroll through it until they found the one they wanted to consult. This process would sometimes be lengthy, thus leading us to implement publication searching to improve efficiency.

Our first approach to implementing this feature used PostgreSQL's Full Text Search functionality. This searching method provides the capability of identifying documents containing a given search term and sorting them by similarity or relevance[25]. A document can be one or multiple concatenated text fields. In our implementation, publications were the documents, and their "title" field represented them for search purposes (although more fields could be added). We did not commit to this implementation due to two significant drawbacks we observed: worse performance and lack of partial matches. When we compared this implementation's performance to the one in an implementation using a regular text search, we observed that the last was significantly better. We even tried to create indexes to improve the full text search[26] process, however, due to the page title being a field defined in the parent "Page" model (predefined by Wagtail), we could not do it through Django. The second drawback was due to the lack of Django's support for partial full text search matches. No partial matches would mean that a visitor would have to write a whole word of the publication's title to get it as a result of the search.

Based on these results, we decided to proceed using the regular text search. Our implementation started, in the back-end, by creating a new dedicated API endpoint for publication searching. Inside the respective APIView, we validate the user's input, and if it is valid, we build the search QuerySet with those inputs in the "get_queryset" method. Listing 6.18 is a simplified version the QuerySet we used.

[25] Full Text Search: `https://www.postgresql.org/docs/current/textsearch-intro.html`
[26] Indexes for full text search:
`https://docs.djangoproject.com/en/2.1/ref/contrib/postgres/search/#performance`

```
1  def get_queryset(self, search_text, desired_year, desired_type):
2      index_page = PublicationIndexPage.objects.first()  # There is only one
3
4      query = PublicationPage.objects.child_of(index_page).live()
5
6      if search_text != None:
7          query = query.filter(title__icontains=search_text)
8
9      if desired_year != None:
10         query = query.filter(year=desired_year)
11
12     if desired_type != None:
13         query = query.filter(pubtype=desired_type)
14
15     return query
```

Listing 6.18: Publication searching QuerySet.

This QuerySet starts by representing all the publication pages and gets progressively filtered according to the request's values (if set). Line 7 is the statement that will originate the query to search by the publication's title (it is case-insensitive due to "icontains"). In the following lines, we can also observe two more filter statements: one for years and the other for publication type. In addition to searching, we also added publication filtering, which can considerably facilitate finding a publication in a long list. Once we assigned a URI to the APIView, our back-end implementation was completed.

We had to make a few adjustments to the display logic in the front-end. At this point, publications were paginated, so we added a search bar, two dropdowns to apply both the year and publication type filter, and a "search" button. Upon entering some text or filtering any of the fields, the search button unlocks, thus allowing the visitors to perform the search request. Once received, all the results are displayed, and the paginated structure is ignored. Figure 6.2 shows the "Publications" page after this implementation was complete.



Figure 6.2: Publications page after changes.

As a way to improve efficiency on tasks using the "News" page, we also implemented sorting and filtering articles by date. This required us to modify the "PaginatedNews" class (see Listing 6.16) to accept a "year" parameter and build the QuerySet using a filter for that year. When a visitor selects a year in the filter dropdown on the front-end, unlike the "Publications" page, an API request is immediately sent to fetch the new parameters. Once received, the page's content gets updated. As for our sorting implementation, it happens on the client-side only, therefore, changing the article order will only change the order of the already-loaded articles instead of generating a new request. Figure 6.3 shows the "News" page after this implementation was complete.

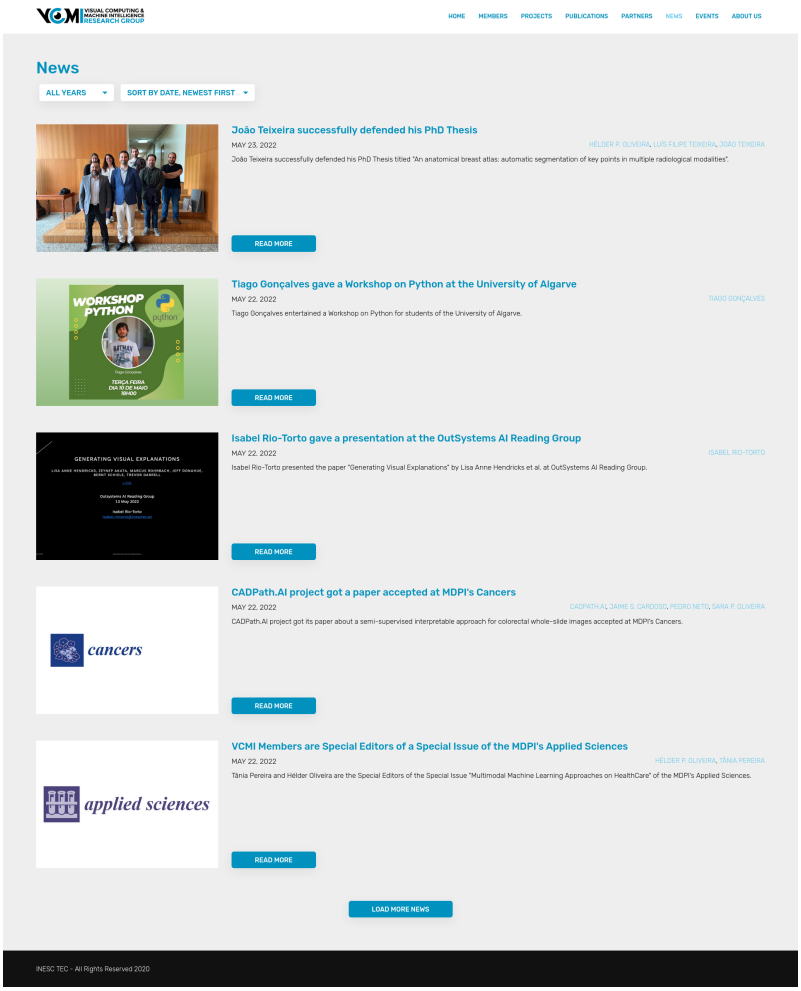


Figure 6.3: News page after changes.

Visual feedback

Most of the application's web pages use a subset of the database's data and, therefore, before the page can be fully displayed, said data has to be requested, fetched, and sent back as a response. This process may be time-consuming, depending on the amount of data requested and the complexity of the queries executed. Concretely when opening the "Publications", "News", and "Members" pages, which took a long time (especially before the performance improvements), the lack of a visual indicator would very often confuse the visitors, making them doubt whether or not they had clicked on a given component.

In order to solve this issue, we added a custom loading animation of the VCMI's icon (see Figure 6.4) that appears every time a visitor switches between pages while the next loads. We also added this loading indicator in search fields and other components that load more data into the page and whose operations tend to be lengthy. Furthermore, we added field state indicators that show whether a field is valid or invalid (see Figure 6.5) and floating error and success messages (see Figure 6.6). We expected these changes to improve effectiveness, efficiency and user satisfaction.

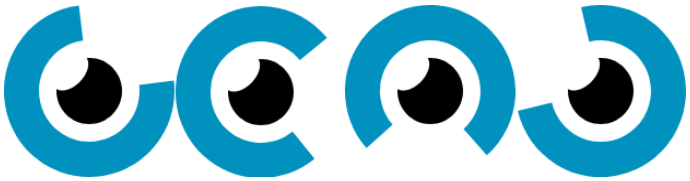


Figure 6.4: Loading animation (frames).

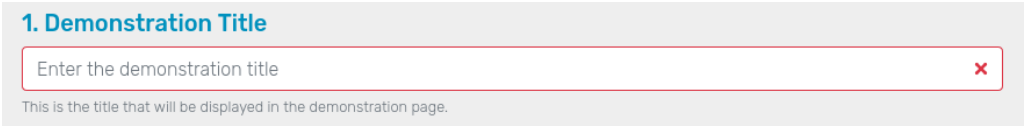


Figure 6.5: Validated field example.

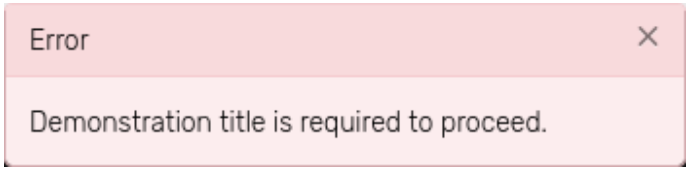


Figure 6.6: Floating message example.

Homepage layout change

The application's homepage contains some of the team's projects, some news, a description of the team and its purpose, and contact information. However, it also had an image banner at the top that occupied the whole screen, thus making it necessary to scroll down in order to access the other components. As this banner was simply an image with a phrase and therefore had no use, we decided to reorganise this page.

We figured that the component that would be more interesting and relevant to promote on the application's homepage would be the most recent news. Until this point, the news displayed on this page were being manually selected through the admin's interface and, therefore, sometimes were not on par with the most recent ones. This factor led us to create a new dedicated API endpoint to fetch the most recent news. Once this step was concluded, we created a carousel to display and automatically cycle through the five most recent news (or articles) and moved it to the beginning of the page so that it gets highlighted. Some other minor adjustments were made to the rest of the page components to maintain consistency. Figure 6.7 shows the home page layout before and after the changes (note that in the first one some news were omitted to shorten the image height).

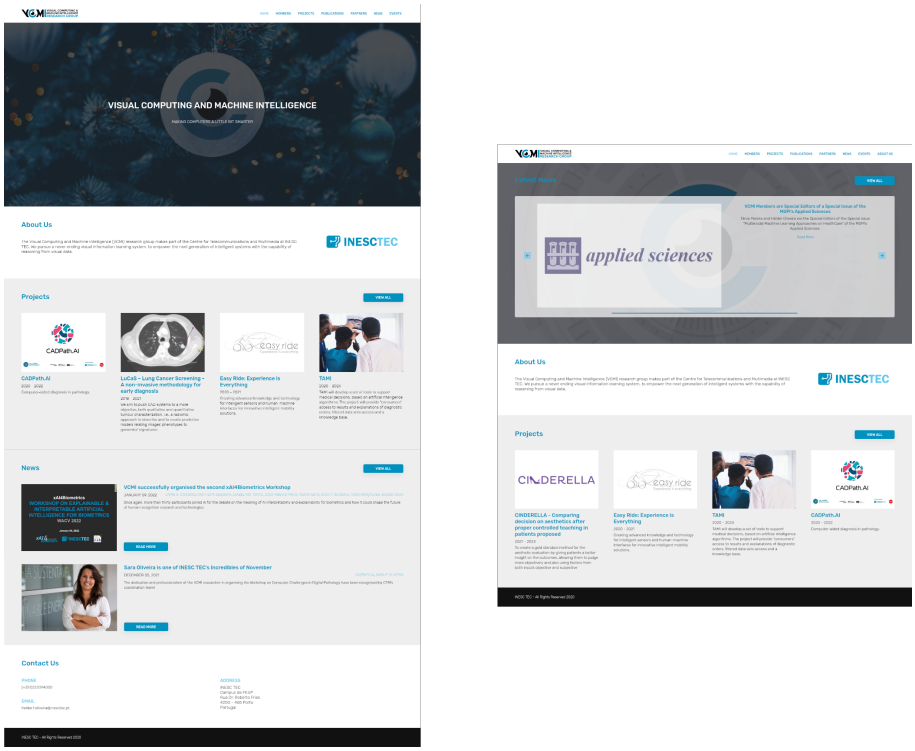


Figure 6.7: Homepage layout before and after the changes.

6.2.2.3 Requirements from the first usability evaluation

Login button

For a team member to make changes to their personal information (e.g. its picture or description), they must use their credentials to log in to the administration page, make the desired change, submit it, and wait for an administrator's approval. However, the page is not accessible through the user interface, i.e. it requires appending "/admin" to the website's URL. Furthermore, since these types of operations are rare, most of the team members end up forgetting how to access the administration page and, therefore, do not know how to submit changes to the application administrators.

We implemented a simple solution for this problem: a button on the "Members" tab that redirects the visitor to the administration page, where they can log in. Usually, such a button is placed in the header to be accessible throughout the entire application, however, since it is only relevant to the team members, we decided to put it inside the "Members" tab only. We expected this solution to improve the application's effectiveness.

Dedicated tab for the team's description and contacts

As shown by the results of the first usability evaluation, 50% of the participants had some difficulty

finding how to contact the team. Based on this observation, we decided to highlight that information by creating a new, dedicated tab on the application's navigation bar for it. Upon clicking on this button, a page with both the team's presentation and contacts is displayed.

In order to add the new header-bar link, we added the components in Listing 6.19 to our "Header" component, and made the displaying text "About us".

```
<b-nav-item>
    <router-link class="nav-route" to="/about-us"> About Us </router-link>
</b-nav-item>
```

Listing 6.19: Adding a new header bar link.

Due to the application's database schema (heavily influenced by Wagtail's requirements), we had to create a new "AboutPage" model, which extends Wagtail's "Page" model so that we could add a new page. This model, in which we specified the data fields we wanted to store, is then interpreted by Django and originates a new Database table. The data fields are converted to the most appropriate types supported by the database. Listing 6.20 contains the model creation and one of the fields added.

```
class AboutPage(Page):
    about_us_text = models.TextField(
        "About us extended text",
        max_length=600,
        blank=False,
        help_text="This should be a descriptive description of the team"
    )
```

Listing 6.20: Adding a new page model.

In this case, the "about_us_text" text field, which can not be null and has a maximum length of 600 characters, was converted to a "text" data type on PostgreSQL. The resulting table Django generated from the full model can be verified in Listing 6.21.

```
CREATE TABLE public.pages_aboutpage (
    page_ptr_id integer NOT NULL,
    about_us_text text NOT NULL,
    phone_number character varying(20) NOT NULL,
    email character varying(50) NOT NULL,
    address text NOT NULL,
    logo_id integer
);
```

Listing 6.21: Resulting SQL table generated from Django's model.

The only missing step was to specify which field should be returned by the API once a specific "About us" page was requested. The way to achieve this is by adding an "APIField" object to a list called "api_fields", as Listing 6.22 shows.

```
api_fields = [
        APIField("about_us_text"),
        APIField("logo", serializer=IndividualImageSerializedField()),
        APIField("phone_number"),
        APIField("email"),
        APIField("address", serializer=RichTextSerializedField),
    ]
```

Listing 6.22: Fields of "AboutPage" to be returned by API.

As we can see, some of the fields have a "serializer" argument being passed. It is used to control how a given field is outputted[27].

After this step was concluded, we created a row in the "pages_aboutpage" table and filled its fields with the required information through the administration panel.

Finally, we built a page (for both the VCMI and breast front-ends) that requests the page's data and displays it. After this step, our implementation was concluded. Visitors were now able to visit the new "About us" page.

6.2.2.4 Others

These small tasks are not part of the previously defined usability requirements but were still accomplished, as we believe these improve the visitors' experience and, therefore, may also improve user satisfaction. Some were selected from the administration's improvements list, and some came as a consequence of using the application.

In order to improve the consistency while navigating through the various web pages, we forced the opening of external links in a new tab. Unfortunately, some inline links (primarily present in an event's description) could not be changed. Even considering this limitation, we believe that visitors will have a more consistent experience using the application with this modification.

While visiting a member's page, all the publications they were involved in were displayed. This design decision was posing a problem, especially in the pages of the oldest members of the team, which have more publications, as it would make the page unnecessarily long, thus increasingly difficult for a visitor to reach its end. Faced with this, we decided to show by default only the latest five publications, and if the visitor desired to view more, they would click on a "View all" button or filter the member's publications by a given year.

On the "Partners" page, while using a small view (visiting with a mobile phone or small browser window), all the partner names that would take more than one line would overlap, thus

[27] Serializers: https://www.django-rest-framework.org/api-guide/serializers/

making it unreadable. This issue got fixed by adjusting the "overflow" and "padding" properties of the partner list.

Finally, motivated by Lighthouse's improvement suggestions, we decided to enhance the accessibility of the various application pages. In the recently modified pages, we added an "alt" attribute to all the "img" tags so that every image has an adequate description if it is not possible to display it. Another step towards this objective was adding the "aria-label" attribute to the elements with no visible text on the screen (in this case, that relied on icons to denote its purpose) so that screen readers (mainly used by people with impaired vision capabilities) may provide a text alternative to them[28].

6.2.3 Second Lighthouse audit

After all the improvements were finished, we reran Lighthouse's audit for each of the main pages in order to quantify the improvements observed according to its evaluation parameters. Figures 6.8 and 6.9 contain the results obtained for the desktop and mobile views, respectively. Table 6.10 summarises these results.

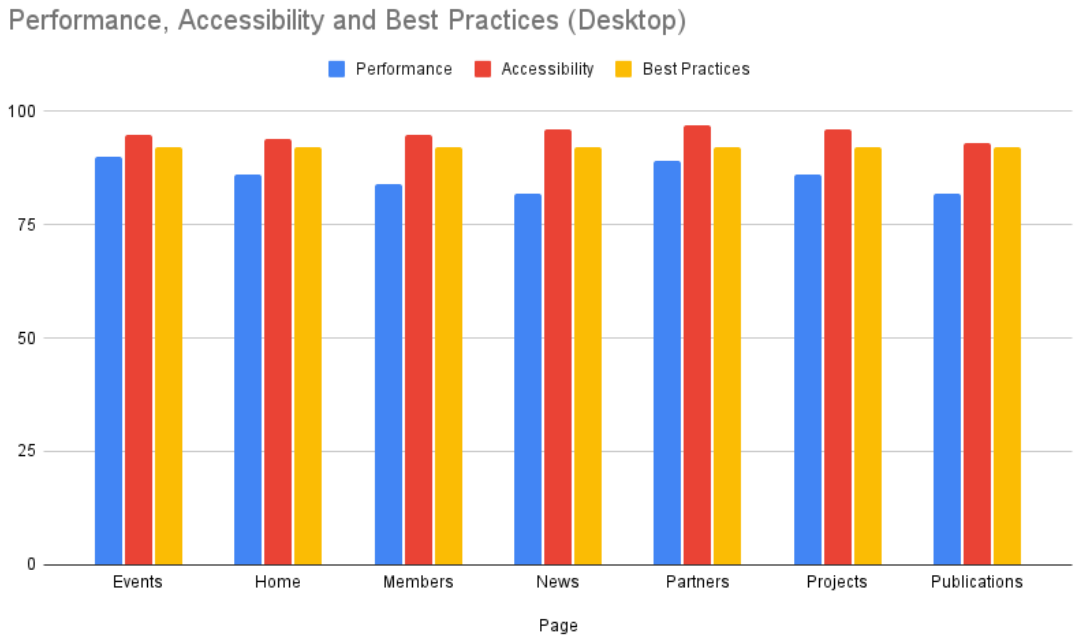


Figure 6.8: Second Lighthouse audit results (desktop view).

[28] Aria-label: `https://www.aditus.io/aria/aria-label/`

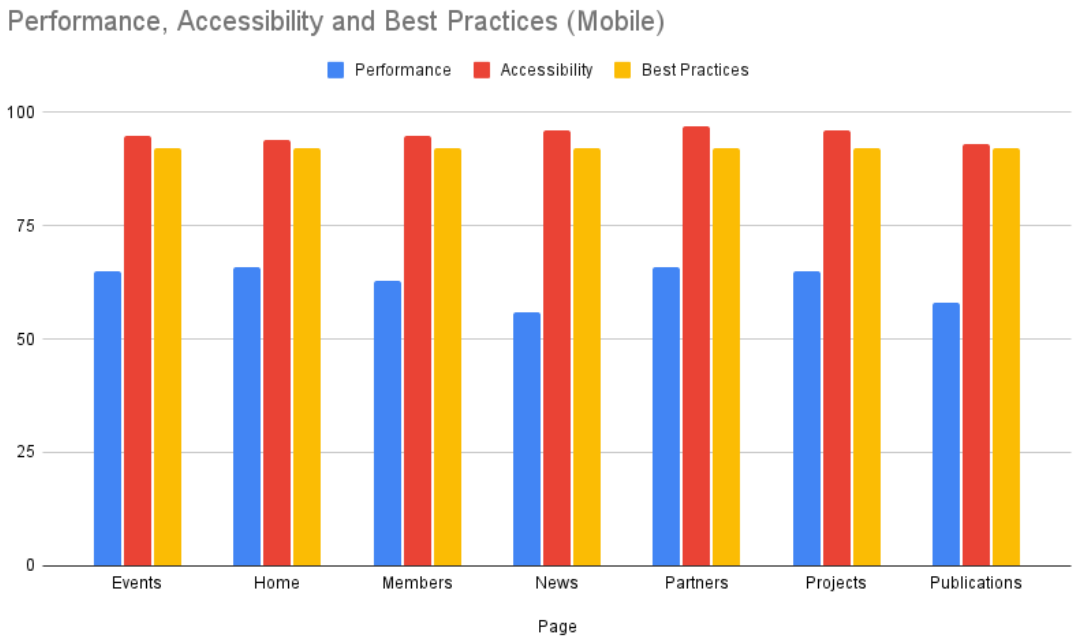


Figure 6.9: Second Lighthouse audit results (mobile view).

Table 6.10: Summarized audit reports.

View	Aggregation	Performance	Accessibility	Best Practices
Desktop	Min	82	93	92
	Max	90	97	92
	Average	85.57	95.14	92
Mobile	Min	56	93	92
	Max	66	97	92
	Average	62.71	95.14	92

When we compare these audit results with the values of the first (see Table 5.1), we verify improvements in performance and accessibility and a constant "Best Practices" score. Particularly in the desktop view, we observe an average performance increase of 18.14 points (from 67.43 to 85.57) and an average accessibility score increase of 6.28 points (from 88.86 to 95.14).

In the mobile view, we observe a slight increase in the average accessibility score of 0.71 points (from 94.43 to 95.14). On the other hand, comparing the performance score, we see a significant increase of 24.57 points (from 38.14 to 62.71). This is both the most significant increase and the lowest overall score value. The scores are lower in the mobile view because Lighthouse emulates a mobile device and throttles its CPU. Furthermore, it also simulates a slow (significantly throttled) 4G mobile connection, with a throughput of 1638.4 Kb per second. Another factor that affected

the mobile view's score is that we are not serving smaller images in this view, which would reduce the total data required to load the page and, therefore, the page loading time.

6.2.4 Second usability evaluation

This second usability evaluation used the same methodology described in the first one (see subsection 6.2.1). The tester's tasks were kept unchanged in order to be comparable. Since one of the implemented changes moved the most recent news to the home page, and the second task consisted of reading them, we required participants to open the "News" page (as in the first evaluation) to consider it completed.

In this second round of tests, there were five different participants: four interns and one team member. Neither of the testers ever used the team's application, but everyone is used to navigating through websites daily. The results obtained are summarised in tables 6.11, 6.12 and 6.13.

Effectiveness

Table 6.11: Task completion, by tester (second evaluation).

Task Completion	Task	1	2	3	4	5	6	7	8
Tester									
1		1	1	1	1	1	1	1	1
2		1	1	1	1	1	1	1	1
3		1	1	1	1	1	1	1	1
4		1	1	1	1	1	1	1	1
5		1	1	1	1	1	1	1	1

Success Rate (%)		100	100	100	100	100	100	100	100

From the analysis of this table, we see that every tester successfully completed every task.

Compared to the first evaluation, we can verify an improvement in the effectiveness of the testers performing the eighth task. This fact is an indication that the addition of the "Login" button on the "Members" page was an appropriate solution to the problem raised in the first evaluation.

Efficiency

Table 6.12: Time taken in each usability task, by tester (second evaluation).

Time taken (s)	Task	1	2	3	4	5	6	7	8
Tester									
1		7.84	3.86	7.35	4.49	5.92	9.84	8.09	12.20
2		3.53	4.02	10.62	3.41	8.10	8.53	20.31	20.09
3		11.18	2.08	35.93	3.52	8.44	14.66	10.63	10.46
4		4.22	3.44	9.54	4.16	15.28	12.91	19.71	17.97
5		2.65	2.80	8.73	3.96	8.47	17.45	21.68	14.88
Average		5.88	3.24	14.43	3.91	9.24	12.68	16.08	15.12

In this second evaluation, we observed an average task completion time of approximately 10 seconds, with the minimum being 3.24 seconds (second task) and the maximum being 16.08 seconds (seventh task).

When we compare these results with those of the first evaluation, we verify a reduction of around 10 seconds of the overall average task completion time (half of the time). This improvement shows us that our efforts to improve efficiency were successful. On the other hand, when we analyse the tasks individually, results suggest that: in the first task, unlike the rest, we observe an increase in the mean task completion time of 2.13 seconds. We believe that the observed deviation is mainly due to testers 1 and 3, whose times are significantly higher than the rest and would be less influential if we had more than five participants. Since the first task consisted of viewing the list of partners, and we did not make any changes that affected its process, we cannot verify any improvement or regression regarding this task. In the second, fifth and sixth tasks, there was a reduction in the average task completion time of 5.44, 2.84 and 18.76 seconds, respectively, which we think are primarily due to the page loading time improvements. Like the first task, the third was not significantly impacted by our optimisations. In this task, a slight time reduction of 0.63 seconds was verified, which we believe is not meaningful. In the fourth task, where the testers had to find the team's contacts, we observed our most significant improvement of 27.69 seconds. Faced with this fact, we concluded that highlighting the team's information by moving it to a dedicated section of the header bar was an effective solution. Regarding the seventh task, which required testers to find and read a specific publication, we verified a completion time reduction of 23.46 seconds. Furthermore, since most participants used the newly implemented search functionality, we concluded that our solution was appropriate. Finally, in the eighth task, we cannot make comparisons regarding efficiency because, in the first evaluation, it was not completed by any of the testers.

User satisfaction

Table 6.13: User satisfaction evaluation - SUS scores (second evaluation).

Scores	Item	1	2	3	4	5	6	7	8	9	10	Total	SUS Score
Tester													
1		2	3	3	4	4	4	2	3	2	4	31	77.5
2		3	3	3	4	3	3	3	3	2	3	30	75
3		3	4	4	4	4	3	4	4	4	4	38	95
4		4	4	4	4	4	4	4	4	4	4	40	100
5		4	4	4	4	4	4	4	4	4	4	40	100

Final Score (Average)
89.5

The overall SUS score obtained in this second usability evaluation was 89.5, representing a considerable increase compared to the first. Although there were no modifications explicitly dedicated to improving user satisfaction, we consider that some of them (e.g. the addition of visual feedback and the overall increased responsiveness) ended up indirectly affecting this evaluation component positively.

6.2.5 Conclusion, considerations and limitations

The second usability test confirms the accomplishment of the third objective of this dissertation project. By relying on user feedback and analysing the results of the first usability evaluation, we were able to identify the most relevant issues, and by solving them, we were able to enhance effectiveness, efficiency and user satisfaction, therefore improving the application's usability.

We believe the implementation phase of this usability study is a prime example of the advantages and disadvantages of functionality abstraction: On the one hand, thanks to Django's ORM, querying the database for the required data was quick and highly simplified. On the other hand, not fully comprehending how QuerySets work made the developers of the application's initial version introduce an unintended behaviour that would constitute a severe performance and usability issue a few years later.

One limitation we identified of our implementation, specifically introduced by the conversion of the served image format to WebP, was that some out-of-date browsers could not display them. Although this image format is supported by most web browsers[29], there will be rare cases where the visitors will not be able to view the application's images. In the future, we could also serve an alternative image format (such as PNG or JPEG) so that these rare cases still display the images (by using the fallback format), even if it takes slightly longer to load.

[29] WebP global support: `https://caniuse.com/?search=webp`

6.3 Interactive demonstrations

This section starts by describing two alternative solutions to add interactive demonstrations to the team's application and then proceeds to compare them, culminating with our decision. After that, the implementation of a new software tab is described, followed by the implementation of an example demo. Finally, we introduce and explain the demo creation helper, a form to help team members build new demonstrations.

6.3.1 Implementation approaches

Producer/consumer custom solution

Our first plan to implement interactive demonstrations on the team's application consisted of a producer/consumer container-based approach where demo requests would be placed in a queue (produced once a visitor loads a given demo's page) and consumed by (one of) the respective demonstration container(s) running. Upon consuming a request, a container would establish a connection to the client's web browser through a WebSocket and send all the demonstration data that would be displayed on the demo's page (e.g. video, images, text) and receive the visitor's input. In Figure 6.10, we can see a preliminary version of the application's component diagram with most of the required components for this approach.

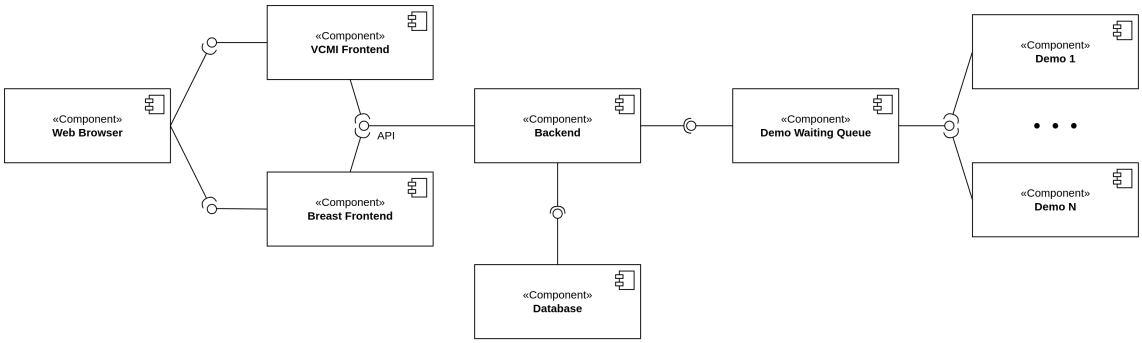


Figure 6.10: Preliminary component diagram, assuming producer/consumer-based solution.

Using this approach, a team member would use a template Dockerfile as a base and build a Docker image of their algorithm by adding their code and respective dependencies. Furthermore, they would have to adapt their code to make it not only constantly consume requests from the queue and send its outputs through a WebSocket but also update the execution parameters according to the inputs forms in the demo's page. Although this last step may be attenuated by providing a dedicated library for such processes, considerable code adaptations would still be required.

This kind of solution is scalable because it allows us to increase the number of consumers by replicating a given demonstration container, thus increasing the system's capacity to handle that specific demonstration (useful in situations where a high load is predicted). It also is interoperable because it is independent of what each demonstration container is running, as long as it follows the same communication protocol (through WebSockets). This last property would allow us to

incorporate demonstrations using different programming languages and environments. Another positive aspect is that we would have complete control of the communication protocol, input, and output details, thus allowing us to optimise them for our specific use cases.

Assuming that the Docker image creation process can be simplified, this approach's main downside is that a considerable amount of code would have to be adapted to follow the demonstration's communication protocol.

Streamlit-based solution

Streamlit[30] is an open-source Python library that aims to make the process of creating web apps to showcase machine learning and data science applications as easy as possible[31]. It was created in January 2018 and has been in active development ever since[32]. This library is also well documented[33], and has community forums that may be used to troubleshoot issues[34].

Our second plan consisted of incorporating containerised Streamlit applications into our system. Since Streamlit applications are served by their own web server, we would simply have to redirect the visitors to the URL of the requested demonstration. Figure 6.11 displays the application's component diagram that this approach would lead to.

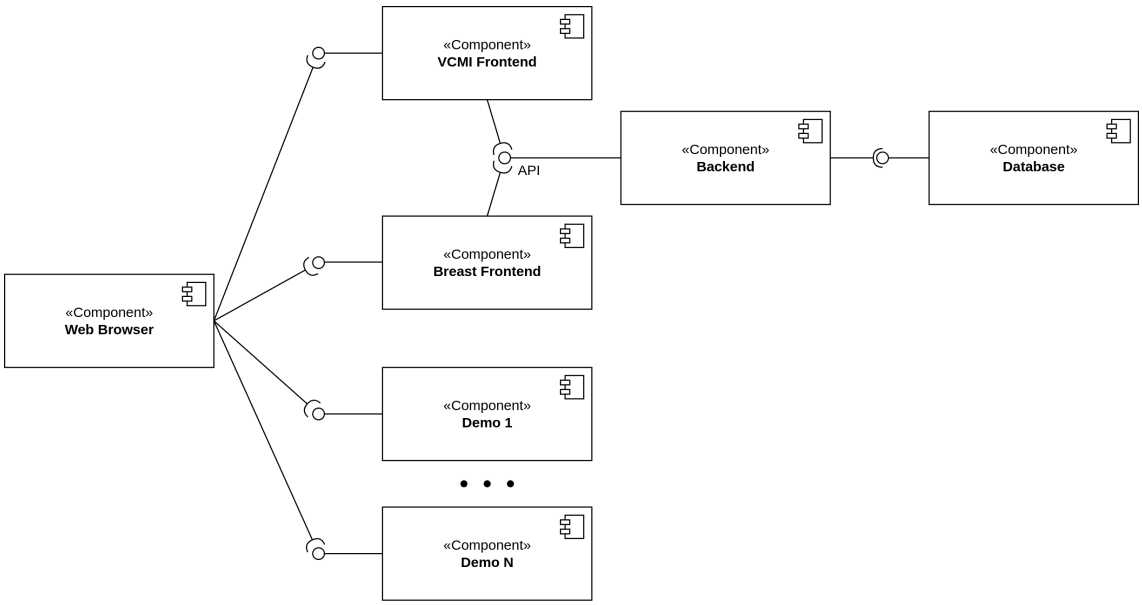


Figure 6.11: Component diagram, assuming Streamlit-based solution.

Using this approach, a team member would need to make a few adaptations to their code to use Streamlit's components. Then, they would need to build a Docker image based on a template

[30]Streamlit: `https://streamlit.io/`
[31]Introduction to Streamlit: `https://docs.streamlit.io/`
[32]Streamlit's GitHub commit history:
`https://github.com/streamlit/streamlit/graphs/commit-activity`
[33]Streamlit's documentation:`https://docs.streamlit.io/`
[34]Streamlit's community forums:`https://discuss.streamlit.io/`

Dockerfile by adding their code and respective dependencies. Once the new demonstration was functioning as intended, it would be deployed and accessible through a URL.

Applications using Streamlit are written just like a regular Python script[35]. Considering that most of the team's algorithms are written in Python, most members are familiar with it, which would also be a favourable characteristic. Another positive characteristic, besides simplicity, that would favour a solution using this library is the wide variety of input widgets (elements that the visitor may interact with) and display elements (elements that display content to the visitor) it supports. Regarding display elements, Streamlit supports, among others, text fields, images, image streams, tables, charts, audio and video. Moreover, when it comes to input widgets, checkboxes, sliders, text inputs, date pickers, and file uploaders are supported[36]. As we can see, these would cover not only all of the input and output requirements suggested by the team members but would also add many other useful ones.

On the other hand, a possible complication with a solution using Streamlit is its unique data flow: when something must be updated on the screen (e.g. due to a visitor updating an input widget's value), Streamlit reruns its entire script[35], which may be challenging to understand correctly. Another possible difficulty of this approach is the step of building a Dockerfile, which can sometimes be challenging, even using a template Dockerfile.

This solution also has a few limitations: when compared to a custom one, we end up losing customizability and control in favour of functionality abstraction, i.e., even though Streamlit has a wide variety of functionalities for multiple purposes, by implementing a custom solution, we could develop more specific and optimised ones, for our use cases. Another downside is that it would also be limited to Python algorithms and environments.

Final decision

After pondering these two approaches, we decided to proceed with the second one. We believe that the Streamlit-based solution is the one that will better fit the team members' needs due to its simplicity, good documentation and the large variety of functionalities and input/output components it provides. Although a custom solution would indeed allow the development of components dedicated to the team's needs, we could never match all the functionalities that this library supports and the documentation it has, especially in the amount of time we had left. Furthermore, considering that most of the team's algorithms are written in Python, the fact that Streamlit only works with this language does not represent a significant drawback to us. This decision also implies that we must cope with Streamlit's potential bugs and limitations.

[35] Streamlit data flow:
`https://docs.streamlit.io/library/get-started/main-concepts#data-flow`

[36] Streamlit supported input widgets and display elements:
`https://docs.streamlit.io/library/api-reference`

6.3.2 Software publishing implementation

Software section

Team members showed interest in having a dedicated section of the application for publishing some of the software they developed. In this section, a visitor would be presented with a list of software entries and their respective short description, from which they would be able to read more about it. Furthermore, every software entry would optionally have a download link that leads the visitor to the software's repository and a link to try it live (interactive demonstration). Since not every software published will be downloadable or have an interactive demonstration associated.

Our implementation process started by creating Django's models for the individual "Software" page and then for the software index page. The last one only has the fields of the class it inherits from (the default Wagtail "Page" model) and is parent to every other individual "Software" page. The software page model's fields are displayed in Table 6.14.

Table 6.14: Software page model fields.

Field name	Field type
software_image	ForeignKey (wagtailimages)
summary	TextField
description	RichTextField
vcmi	BooleanField
breast	BooleanField
repo_url	URLField
interactive_demo	ForeignKey (Demo)

Since there may be many software entries in the future, we decided to paginate the "Software" index page. Once the models were built, we created a new route and API endpoint to get the paginated software list, ordered by the first publication date, just like the "News" page (see section 6.2.2.1). The QuerySet used to fetch these items can be verified in Listing 6.23.

```
1  def get_queryset(self, is_breast):
2      index_page = SoftwareIndexPage.objects.first()  # There is only one
3
4      query = SoftwarePage.objects.child_of(index_page).live().select_related("
           software_image", "interactive_demo")
5
6      if is_breast:
7          query = query.filter(breast=True)
8      else:
9          query = query.filter(vcmi=True)
10
11     return query
```

Listing 6.23: Software pages QuerySet.

Once the back-end expansions were complete, we moved to the front-end, where we created the software index page (that lists all the software in a paginated way) by requesting them to the API route we created. On this page, each software entry contains its title, summary and a button to read more about it. Optionally, it may also contain an image, a link to download it, and a link to a demonstration. The second page we created was the individual software page, where a visitor may read the full description and access all related links of that particular software. Figures 6.12 and 6.13 show the aspect of the software index page and the individual software page, respectively.

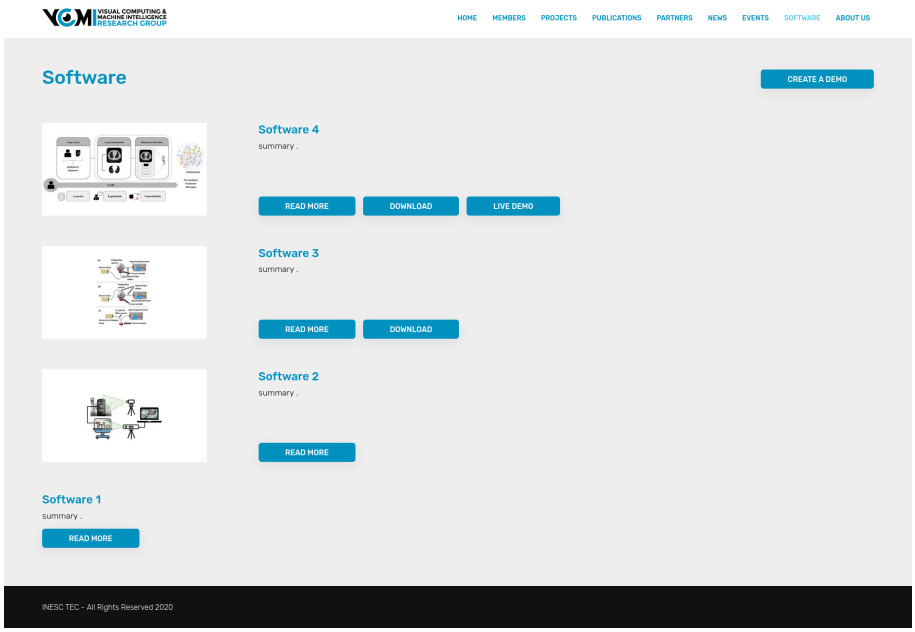


Figure 6.12: Software index page.

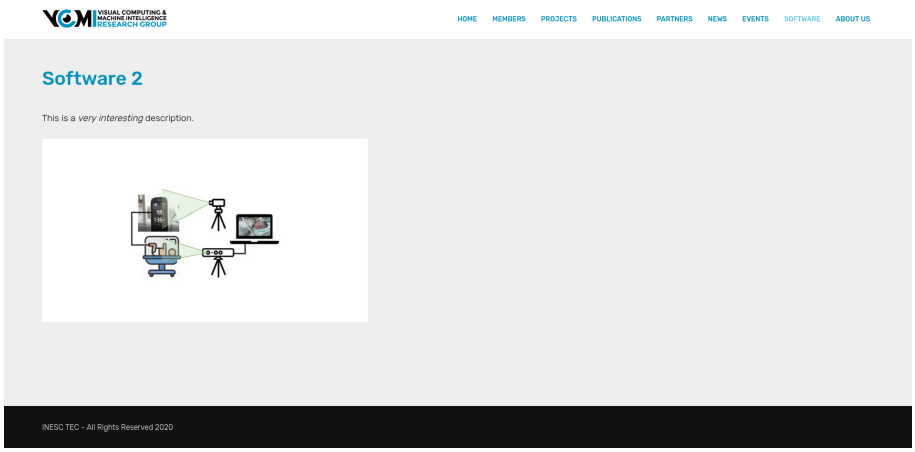


Figure 6.13: Software page.

Containerising example demo

As proof of the concept of our solution, we created an example demonstration. Since team members may use this demo as a reference for implementing their own in the future, we decided to make ours out of an algorithm that outputs a sequence of images (image stream). We think this type of output is the most complex one (out of the ones suggested by the team members) to implement using Streamlit, as there is no specific component for it, therefore, making it a valuable example.

This process started by developing the base algorithm of our demonstration, which can be seen in Listing 6.24.

```
def convert_video(video_file_path):
    if video_file_path:
        video_capture = cv2.VideoCapture(video_file_path)

        while video_capture.isOpened():
            succeeded, frame = video_capture.read()

            if not succeeded:
                print("Error reading... Ending")
                break

            grey = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
            cv2.imshow("Grey video", grey)
```

Listing 6.24: Example demo - initial algorithm.

This simple algorithm uses the OpenCV[37] library, and, simply put, it reads a video file, frame by frame, converting it [the frame] to greyscale, and displaying it on the screen (on a new window). Since a video contains multiple frames, this generates a sequence of images (image stream).

With the algorithm finalised, we began creating the demo by making a basic Streamlit application with a title. For this purpose, we created a new Python script in which we imported the "streamlit" package and set the application's page title, icon and layout as Listing 6.25 shows.

```
streamlit.set_page_config(
    page_title="Video Example Demo",
    page_icon="/app/favicon.ico",
    layout= "centered",
)
```

Listing 6.25: Example demo - page configuration.

We then proceeded to put the team's logo and a header on our page, as Listing 6.26 displays.

[37] OpenCV: `https://opencv.org/`

```
col1, col2 = streamlit.columns([1, 5])

with col1:
    streamlit.image("/app/favicon.ico")

with col2:
    streamlit.title("Video Example Demo")
```

Listing 6.26: Example demo - page header.

This code creates a two-column container where the second column's width is five times greater than the first's and puts an "image" component in the first column and the title in the last.

At this point, our application simply had a header and the team's icon and was still missing the input and output components. Because our base algorithm requires a video as input, we decided to add a file upload component. Moreover, to display its output, we added another component: a single element empty container. This placeholder may be filled by any other component[38] and will only display something when it is. Listing 6.27 illustrates how these components were created.

```
# Inputs
uploaded_video = streamlit.file_uploader('Choose a Video', type=['mp4'])

# Outputs
stframe = streamlit.empty()
```

Listing 6.27: Example demo - input and output components.

These few lines allowed us to cover both the inputs and outputs of our algorithm. It is worth noting that components are added from the top to the bottom of the page by order of creation, therefore, the file uploader would appear below the header and icon we previously added, and the image stream would be below the file uploader (once written). Another note is that once an input component is updated (like the file uploader), the whole application script is rerun.

Finally, and although not required, we added a button that appears once a file is uploaded. This button starts the demonstration by calling our algorithm and passing the uploaded file and our placeholder component, "stframe". Its creation code is displayed in Listing 6.28. Note that it only is created once the variable "uploaded_video" is set (once a video has been loaded).

```
def run():
    convert_video(uploaded_video, stframe)

if uploaded_video:
    streamlit.button("Run!", on_click=run)
```

Listing 6.28: Example demo - run button.

[38] Streamlit's empty component:
https://docs.streamlit.io/library/api-reference/layout/st.empty

This concluded the development of our page, however, at this point, our demonstration still was not finished, as we still needed to adapt our algorithm to make it use our video file and write the result to our placeholder component. Listing 6.29 shows the adapted code.

```
1  def convert_video(uploaded_video, stframe):
2      if uploaded_video:
3          tfile = tempfile.NamedTemporaryFile(delete=False)
4          tfile.write(uploaded_video.read())
5          video_capture = cv2.VideoCapture(tfile.name)
6
7          last_frame_timestamp = time.time()
8
9          while video_capture.isOpened():
10             ret, frame = video_capture.read()
11
12             if not ret:
13                 print("Error reading... Ending")
14                 break
15
16             gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
17             stframe.image(gray)
18
19             last_frame_timestamp = time.time()
20             time_left_sleep = (last_frame_timestamp +
21                                (1/FRAME_RATE)) - last_frame_timestamp
22             time.sleep(time_left_sleep if time_left_sleep > 0 else 0)
```

Listing 6.29: Example demo - adapted algorithm.

Compared to the initial version, we find the main differences at the start (lines 3 and 4), where we create a temporary file (from the "uploaded_video" bytes) in the filesystem in order for OpenCV to be able to read it. Moreover, in line 17, instead of displaying the frames in a window, we write them to our "stframe" placeholder, using the "image" function. Note that lines 7, 19, 20, 21 and 22 were added only to limit how fast the frames were being displayed.

With the demonstration working as intended, we generated the list of dependencies (requirements.txt) using "pipreqs"[39] tool and then proceeded to create a Docker image. The first step to creating a Docker image is making a Dockerfile. The one we developed can be verified in Listing 6.30.

[39]pipreqs package: `https://pypi.org/project/pipreqs/`

```
FROM python:3.10

COPY . /app

WORKDIR /app

RUN pip install -r requirements.txt

EXPOSE 8502

CMD ["streamlit", "run", "main.py"]
```

Listing 6.30: Example demo - Dockerfile.

We chose to use “python:3.10” as a base image because it is an image that contains a recent version of python preinstalled (3.10). As we can also see, we are copying every file in the directory of the Dockerfile (application script, our algorithm and its dependency list) to the “/app” directory inside the container. Then, we change the working directory to “/app”, install the dependencies with pip, and make the container listen to port 8502. Finally, once the image is instantiated and the resulting container is started, our application will be initiated with the command “streamlit run main.py”, thus concluding our implementation. Figure 6.14 illustrates the resulting application page while running.

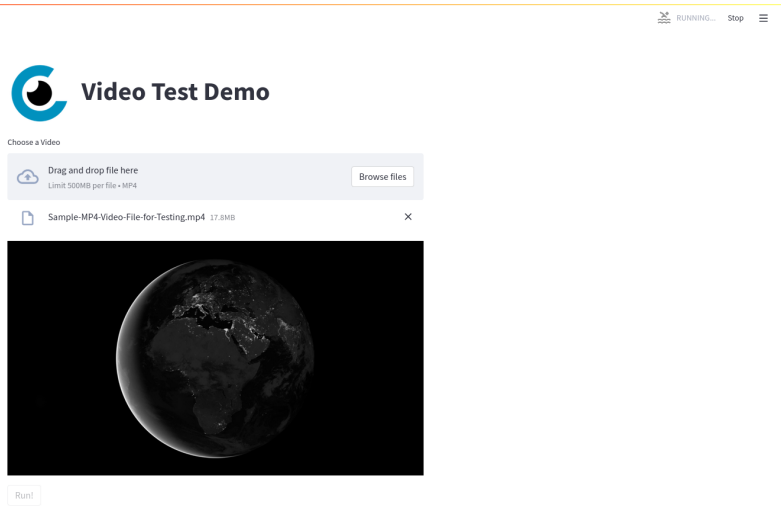


Figure 6.14: Example demo page.

It is worth noting that Streamlit supports multipage applications[40], which we could leverage to put multiple demos in the same application, however, we decided to create a container for each

[40]Streamlit multipage applications: `https://blog.streamlit.io/introducing-multipage-apps/`

demonstration, as it avoids dependency conflicts.

6.3.3 Demo creation helper

For a new demonstration to be deployed into the team's application, it first must be tested by the administrators. This strategy would require the team member who created the demonstration first to send its project folder to the administrators somehow. In order to ease this process as well as to speed up the demo creation and containerisation process, we decided to create a page (on the front-end) to walk team members through the steps required to create an interactive demonstration.

This demo creation helper is a four-step form that aims to simplify the demo creation process, specifically the creation of the Streamlit application, by allowing the download of a template project whose code already has the correct demonstration page layout, as well as the input and output components the team member specified in the form. Furthermore, this also removes the need for them to create a Dockerfile for their project, as it is created automatically. It also has the advantage of allowing project submissions on the last page, thus eliminating the need to send them to the page administrators.

Implementation

Our implementation started by finding a way to generate and return the template project from an API endpoint. In order to make the template project with the user's form data, we used a template-based code generation approach. We used the chevron[41] library, which is a Python implementation of the mustache[42] templating language. For each template project file, we made a template based on the example demonstration (see section 6.3.2). Listing 6.31 shows the template for the Dockerfile used.

```
1  FROM {{docker.python_base_image}}
2
3  COPY . /app
4
5  ADD ./config.toml /root/.streamlit/config.toml
6
7  WORKDIR /app
8
9  RUN pip install -r requirements.txt
10
11 EXPOSE 8502
12
13 CMD ["streamlit", "run", "main.py"]
```

Listing 6.31: Dockerfile mustache template.

[41] chevron: `https://pypi.org/project/chevron/`
[42] mustache: `https://mustache.github.io/`

By providing a dictionary with the value “docker.python_base_image” set, we can generate a Dockerfile whose content is just like the template’s, but with the section between curly brackets replaced by the value of “docker.python_base_image”.

Once every template was assembled, we made a new API endpoint to return a zip file containing a “main.py” script (generated with a template), the team’s icon, and a “Code” sub-directory. The “main.py” script is based on the example demo’s application (see subsection 6.3.2), only requiring the team member to adjust and call their algorithm. This API endpoint starts by validating the user’s input and, if everything is valid, proceeds to build the template project. Listing 6.32 contains a simplified version of the function used to build the template project.

```
1  def buildMain(self, data):
2      with open('demo_templates/main.mustache', 'r') as f:
3          generated_main = chevron.render(f, data)
4
5      with tempfile.TemporaryFile() as tempFile:
6          with zipfile.ZipFile(tempFile, mode='w', compression=zipfile.ZIP_DEFLATED)
               as zipedFile:
7
8              # Code folder
9              zipedFile.writestr("Code/", "")
10
11             # main.py
12             zipedFile.writestr('main.py', generated_main)
13
14             # favicon.ico
15             zipedFile.write('demo_templates/favicon.ico', 'favicon.ico')
16
17         return True, tempFile.read()
```

Listing 6.32: Template project build function.

In lines 2 and 3, we are opening the “main.mustache” template and generating the code for “main.py” with the user input stored in “data”. In lines 5 to 15, we create a temporary zip file and add the “Code” sub-directory, the team’s icon and “main.py”. Finally, in line 17, we return the zip file’s bytes. Since we are returning a file instead of a JSON response, we had to specify the content type (application/x-zip-compressed) and the file’s name (TemplateProject.zip), as Listing 6.33 shows.

```
1  response = HttpResponse(proj_bytes, content_type='application/x-zip-compressed')
2  response['Content-Disposition'] = 'attachment; filename=TemplateProject.zip'
3
4  return response
```

Listing 6.33: Template project API response.

Before we started the second API endpoint, we decided to create a new model to store an interactive demonstration's state and unique service name. Table 6.15 contains the fields we used. We also added a new "Demos" tab on the administration panel so that page administrators may easily verify how many demonstrations were submitted and are waiting to be tested.

Table 6.15: Demo model fields.

Field name	Field type
title	CharField
service_name	CharField
waiting_review	BooleanField

The second API endpoint we developed was to submit and store the demonstration project. Firstly, we made the corresponding APIView validate the submitted form data and project structure. Secondly, we made it generate the rest of the files (Dockerfile and Streamlit configuration), write the dependency list (requirements.txt) with the submitted form data and save the resulting project inside a folder on the back-end. Finally, if all the previous steps were successful, a new demonstration is added to the "Demo" table, with the state of "waiting for review".

Since this route accepts files and stores them, it also creates a way for people to fill up the server's disk. A hypothetical attacker may exploit this route to submit multiple folders to fill up the VM's disk space or even to submit malicious code that, without being previously analysed by the administrators, may end up being executed in the VM. As a measure to prevent these situations, we made both of the API endpoints use session authentication. This implies that for any API endpoints to be used, a valid session cookie must be sent, meaning a team member must log in before using this helper.

With our back-end work completed, we proceeded to the front-end, where we began making the page and the form fields: On the form's first page (Figure 6.15), we explained how the helper works and put a (required) text field for the demonstration title.

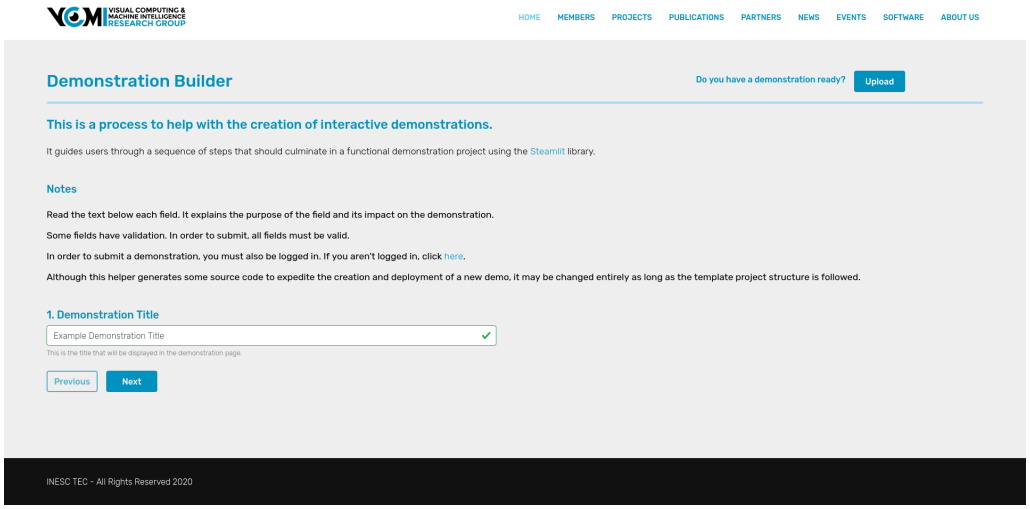


Figure 6.15: Demo helper's first page.

On the second page (Figure 6.16), we added a way for team members to (optionally) add input widgets and display elements to its demonstration. Depending on the type of element selected, a different sub-form appears with the fields required to generate its code, along with some useful configuration options, in some cases. The input widgets supported by the helper are the file uploader, text input (small text field), text area (large text field), slider, checkbox, radio and multi-select. Since, in our approach, display elements only generate an empty component (to be later written), these do not require any extra fields.

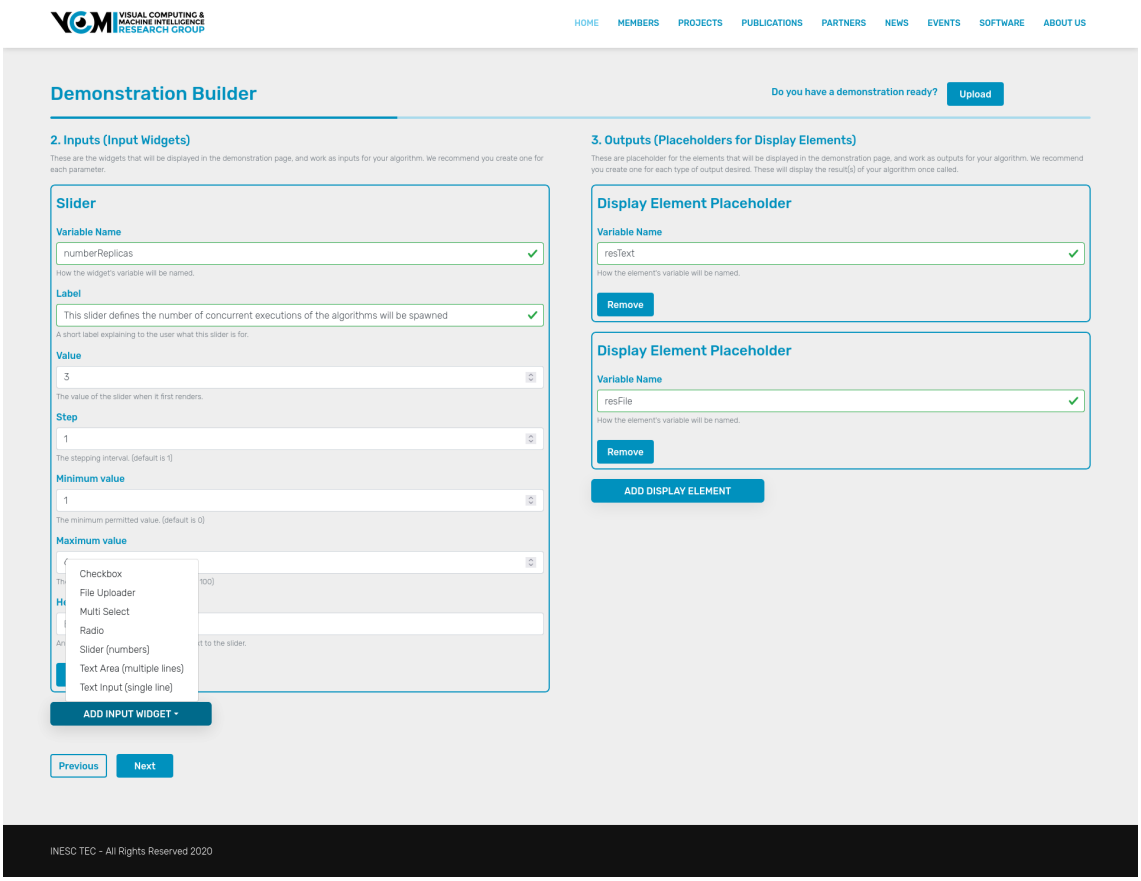


Figure 6.16: Demo helper's second page.

On the third form's page (Figure 6.17), we added a button that sends the form information (of the first two pages) to the first API endpoint we created, where a template project is built and sent back to the client. Still on this page, we added a way to upload the demonstration project (once ready).

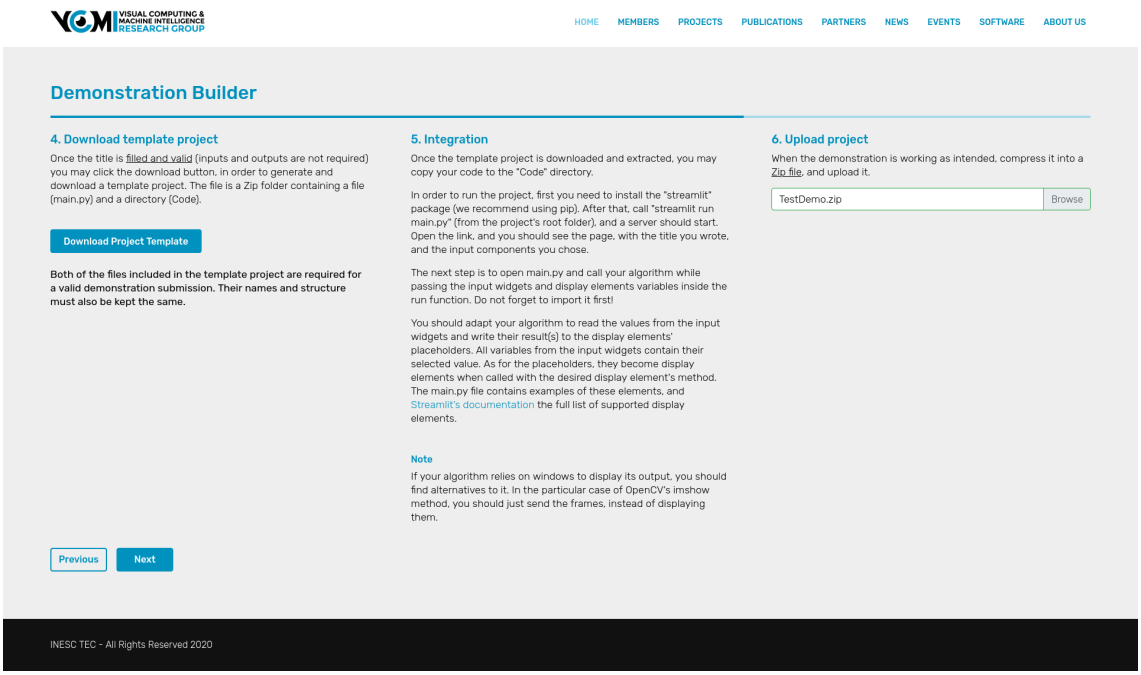


Figure 6.17: Demo helper's third page.

The final form's page (Figure 6.18) is related to the containerisation process. On this page, we added a (required) field for the demonstration's service name, which will define the Docker service name and the URL to access it once deployed. We also added a text area field for the demonstration requirements and a dropdown to select the desired Python version to run inside the container (which defines the version of the "Python" Docker image used). Once the form is valid, it may be submitted, thus sending its fields along with the uploaded project for administration approval (second API endpoint).

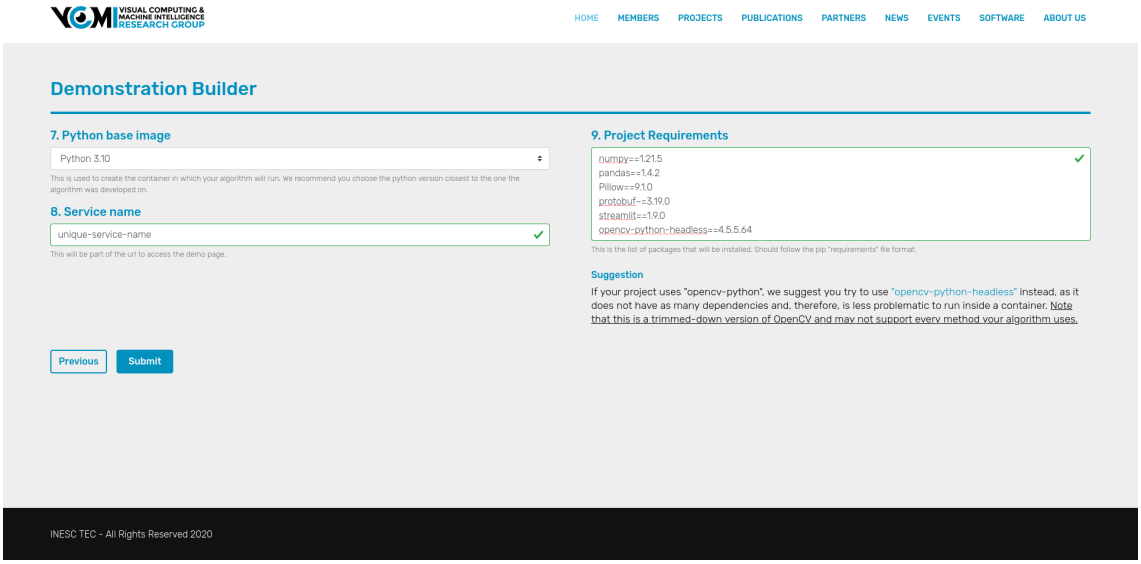


Figure 6.18: Demo helper's fourth page.

6.3.4 Conclusions, considerations and criticisms

We consider that both the new software section and the example demo prove the accomplishment of the fourth objective of this dissertation project, allowing visitors to find more about the team's software as well as interactively try some of it.

An aspect to consider is that as the number of interactive demonstrations deployed in the team's VM grows, so will the resource usage. Depending on the scale and load, the memory, disk space, and even CPU may require an upgrade.

Another consideration, more specifically regarding the demo creation helper, is that while releasing the need for a team member to create a Dockerfile is positive, we also end up losing the assurance that the submitted project will work inside the Docker container, as there was no previous testing in that environment. This may lead to situations where the administrators need to tweak the automatically-generated Dockerfile to make the submitted demo work as intended.

We think that the addition of the creation helper aggravated the security issue that the lack of HTTPS represents. Considering that, while using the creation helper, entire projects will be transferred and that these projects may contain sensitive data and/or not publicly available code, the possibility of an attacker intercepting and reading this data is concerning.

6.4 HTTPS, production environment, automatic publication fetching, and other helpful periodic jobs

This section is composed of other varied improvements made to the team's application throughout the development phase that do not fit the previous sections' subject. In some cases, these changes were introduced to solve some of the limitations of the previous implementations. We start by explaining the process of activating HTTPS and its implications, and then we proceed to describe some periodic jobs that automate several of the administrators' tasks.

6.4.1 HTTPS

The lack of encryption on the team's application represented a security issue, especially since user credentials (including the admin's) could be intercepted and used maliciously. Furthermore, with the addition of the demo creation helper (which increases the frequency of sensitive data transfer), we think the severity of this issue was aggravated. For these reasons, we decided to prioritise HTTP encryption with TLS over other improvements.

In order to upgrade from HTTP to HTTPS, we needed an SSL/TLS certificate, obtainable from a certificate authority. A certificate authority is a trusted third party (an outside organisation) that can issue and digitally sign a certificate (with its private key). These certificates contain the domain name and organisation it was issued to, which certificate authority issued it and its signature, issue and expiration dates, and the application's public key[43]. We started by creating a certificate signing

[43] SSL certificate: `https://www.cloudflare.com/learning/ssl/what-is-an-ssl-certificate/`

request file using the command from Listing 6.34. From this progress, we also obtained a private key.

```
openssl req -out vcmi_inesctec_pt.csr -new -newkey rsa:4096 -keyout
    vcmi_inesctec_pt.key
```

Listing 6.34: Openssl certificate signing request and private key creation.

After we received the signed certificate (.pem file) for the application's domain (vcmi.inesctec.pt), we proceeded to upgrade our connection by configuring our reverse proxy, NGINX.

In our NGINX production configuration file, we changed the port NGINX listens to from 80 (default HTTP port) to 443 (default HTTPS port), we enabled the SSL mode for every connection on that port, and we allowed the acceptance of HTTP/2 connections[44]. Finally, we specified the path of both the signed certificate and private key (we got when generating the certificate), as Listing 6.35 shows.

```
# listen 80 default_server; # Old port
listen 443 ssl http2 default_server;
ssl_certificate     /etc/ssl/vcmi_inesctec_pt.pem;
ssl_certificate_key /etc/ssl/vcmi_inesctec_pt.key;
```

Listing 6.35: NGINX default server configuration changes to allow HTTPS.

Furthermore, we made it so that NGINX also listens to port 80 so that, in case someone tries to access the application through HTTP, they get redirected to use HTTPS. Listing 6.36 shows how it was achieved.

```
server {
     listen 80 default_server;
     return 301 https://$host$request_uri;
}
```

Listing 6.36: NGINX redirect to HTTPS configuration.

6.4.2 Production webserver and environment

When we were deploying the new application's version to its VM (with the implementation of HTTPS), we verified that up to that point, the application's backend had been served on Django's development server, which is not the recommended way to deploy a Django application. Furthermore, some configuration options were set for a development environment, not production. This was especially impactful because Django's development server does not support HTTPS, thus, requiring us to find a solution.

[44]NGINX listen: `https://nginx.org/en/docs/http/ngx_http_core_module.html#listen`

Django's primary deployment platform is the Web Server Gateway Interface (WSGI)[45], a specification that describes how a web server communicates with web applications[46], and it is the Python standard for web servers and applications. The compliant webserver we chose to fill this role was Gunicorn[47]. This pure-Python WSGI server has no dependencies, is easy to install and use, and is also suggested by Django's documentation[48].

In order to use Gunicorn, we simply had to add it to the list of dependencies and modify the back-end start script, making it call Gunicorn instead of Django's development server when in production (defined by a given environment variable). Listing 6.37 demonstrates the command we used.

```
gunicorn --bind 0.0.0.0:8000 wagtail_vue.wsgi
```

Listing 6.37: Start back-end with Gunicorn.

After this change, we could initialise the application, however, static resources such as images and the admin's page were not being served. We solved this issue by serving them with NGINX.

Finally, we also refactored Django's configuration files so that we could set specific settings for development and production environments. Specifically, in the production configuration file, we started by enabling "SECURE_SSL_REDIRECT", which redirects all non-HTTPS requests to HTTPS[49]. Then, we disabled the "DEBUG" mode, which displays a detailed log in case of errors, thus, possibly exposing too much information[50]. Lastly, we enabled cross-origin requests but only allowed those originating from our application's domain (vcmi.inesctec.pt) to improve security.

6.4.3 Periodic publication fetching

The team member's publications shown and stored in the application are fetched from Authenticus[51], a national repository of publications metadata. These were being fetched when the back-end container was started by calling a custom Django command[52] that requests the team's publications through Authenticus' API.

To avoid manual back-end restarts to get the most recent publications, we decided to periodically call this command so that, from time to time, the publication list gets automatically updated.

[45] How to deploy a Django application with WSGI:
`https://docs.djangoproject.com/en/2.1/howto/deployment/wsgi/`
[46] WSGI: `https://wsgi.readthedocs.io/en/latest/what.html`
[47] Gunicorn: `https://gunicorn.org/`
[48] Django with Gunicorn:
`https://docs.djangoproject.com/en/2.1/howto/deployment/wsgi/gunicorn/`
[49] Secure SSL redirect:
`https://docs.djangoproject.com/en/2.1/ref/settings/#secure-ssl-redirect`
[50] Debug mode: `https://docs.djangoproject.com/en/2.1/ref/settings/#debug`
[51] Authenticus: `https://www.authenticus.pt/`
[52] django-admin commands:
`https://docs.djangoproject.com/en/2.1/howto/custom-management-commands/`

We started by initialising the "cron"[53], a daemon to execute scheduled commands, inside the back-end container. Since we already had a script that is executed when the container is started, we simply had to add the line is Listing 6.38 to achieve this.

```
/etc/init.d/cron start
```

Listing 6.38: Start "cron" daemon command.

A scheduled command, or job, is specified in a crontab file, where we may also dictate, besides the command, how often it is executed and the user executing it. The "cron" daemon searches for these files in the "/etc/cron.d/" directory (among others), so we placed our crontab file inside this directory (inside the container). The job we defined in it is shown in Listing 6.39.

```
20 2 * * * root python3 /app/website/wagtail_vue/manage.py fetch_authenticus_api >/
    proc/1/fd/1 2>/proc/1/fd/2
```

Listing 6.39: Cron-job definition.

Our job executes the "fetch_authenticus_api" Django custom command as the root user. The "20 2 * * *" section of the job definition means that we want it to be executed every day at 02:20, and the ">/proc/1/fd/1 2>/proc/1/fd/2" portion is used to redirect the job's output messages to Docker's.

6.4.4 Periodic database backups

The application's database data was being stored in the INESC TEC GitLab. The repository is private; therefore, only a limited number of people can access it. Since these backups were manual and only used to happen from time to time, there is a possibility of losing data when the latest backup is dated and the server's hardware fails. To circumvent this problem, we developed a method to periodically back up the database's data.

Our solution consists of two periodic jobs (in the database container): one to create a database dump (script to rebuild the database) and one to send it, using rsync[54]. It also included the development of a Docker container whose purpose is to keep stored a given number of files and remove the most dated ones if that number is exceeded. In an ideal situation, this container would be deployed to another server with redundancy so that no data would be lost in the event of a disk failure or a disaster on the application's server.

We started by creating a Docker image that, once instantiated, will store the database backups. We used "alpine:3.15" as the base image and installed OpenSSH[55] and rsync. Once the container

[53] cron: `https://man7.org/linux/man-pages/man8/cron.8.html`
[54] rsync: `https://rsync.samba.org/`
[55] OpenSSH: `https://www.openssh.com/`

is started, we initialise the ssh service, "sshd" (required to use rsync), and "cron", the daemon to execute scheduled commands.

With our container almost complete, we generated an ssh key pair for our database container using OpenSSH, through the command in Listing 6.40.

```
ssh-keygen -b 4096
```

Listing 6.40: Generate ssh key pair.

The SSH protocol uses public-key cryptography for authentication[56], therefore, this command generates a 4096-bit RSA key pair (public and private keys). With these keys created and added to the database container, we started the ssh service and added the private key to the ssh-agent, thus concluding the required configuration in this container.

Back in the "backups" image, we had to modify its creation Dockerfile. In order to allow the database container to connect through ssh to this container, we had to add its public key to the "authorized_keys" file in the ".ssh" directory. We also force the execution of rrsync[57] (a script to restrict how rsync is used via ssh) by prepending the public key with the command, as Listing 6.41 shows.

```
command="/usr/bin/rrsync /home/vcmi/vcmi-database/ -wo -no-del" <public key>
```

Listing 6.41: Authorized-keys file configuration.

As we can see, we only allow writing to the directory "/home/vcmi/vcmi-database/" and block deletions. Because the "backups" container only accepts file writes from the database container (no reads), even when an attacker compromises the application's VM, they should not be able to read the saved backups.

Finally, we made the scripts to dump the database's data, send files with rsync, and remove excessive backups. The first two were used on the database container, and the last, on the "backups" container. Jobs were also created for each one of these scripts. Listing 6.42 shows the command used to dump the database data, while Listing 6.43 demonstrates how they were sent.

```
pg_dump <Postgres DB Name> -U <Postgres User> --file=/backups/backup.sql
```

Listing 6.42: The database dump command.

```
rsync -r --remove-source-files /backups/ root@<Backups container IP>:
```

Listing 6.43: The rsync command.

[56] SSH authentication: https://www.ssh.com/academy/ssh/keygen#ssh-keys-and-public-key-authentication

[57] rrsync: https://download.samba.org/pub/rsync/rrsync.1

6.4.5 Automatic image conversion

Shortly after we converted all the application images to WebP and deployed this change, it became apparent to us that it would be hard to maintain all images in this format, mainly because there are multiple administrators, and they may not know how to convert an image to this not so common format.

In order to solve this problem, we decided to implement a cron-job that automatically tries to convert all non-WebP images into this format and updates the database if it succeeds.

We started by creating a custom Django command based on the one we used to fetch publications. This command receives a "filename" argument, which is the path of a non-WebP image, and uses it to find the corresponding row in the "Image" table of the database. Once found, it updates its extension to ".webp" and saves the changes to the database. Listing 6.44 is a simplified version of our implementation (does not have error handling).

```
1  class Command(BaseCommand):
2      def add_arguments(self, parser):
3          parser.add_argument('filename', nargs=1, type=str)
4
5      def handle(self, *args, **options):
6          image_by_filename = Image.objects.get(file="original_images/" + options['
               filename'][0])
7
8          format_index = image_by_filename.file.name.find(".")
9
10         image_by_filename.file.name = image_by_filename.file.name[0:format_index] +
               '.webp'
11         image_by_filename.save(update_fields=['file'])
```

Listing 6.44: Django-admin command to rename image path.

In order to convert images to WebP, we installed libwebp[58] in the back-end (since it is where images are stored). Once installed, we could convert images using the command in Listing 6.45.

```
cwebp image.png -o image.webp
```

Listing 6.45: Convert image to WebP.

With both of these steps complete, we made a script that iterates through all non-WebP images and tries to convert them to this format. If the image gets converted successfully, it calls the custom django-admin command we created to rename it in the database and then removes the old image. In case of failure, it proceeds to the next image on the directory.

Finally, to make this script execute every Sunday at 4:30, we defined the cron-job in Listing 6.46 and added it to our "crontab" file.

[58] libwebp: `https://developers.google.com/speed/webp/download`

```
30 4 * * 0 root /app/scripts/ConvertAllWebp.sh /app/website/wagtail_vue/media/
    original_images/ >/proc/1/fd/1 2>/proc/1/fd/2
```

Listing 6.46: Image conversion cron-job definition.

Chapter 7

Conclusion and Future Work

7.1 Conclusion and main contributions

In this dissertation project, we studied and successfully improved the already-existing team's web application, guided by the objectives described in chapter 1 and the requirements in chapter 5. While keeping future growth, performance and user feedback in mind, we built and deployed a functional new application version. The most pertinent contributions of this dissertation project to the team's application were:

- **New DBMS** We changed the application's DBMS from SQLite to the more capable and adequate PostgreSQL.
- **Usability study and improvement** We improved the application's usability by identifying, studying and fixing its problems, emphasising efficiency significantly.
- **Interactive demonstrations** We developed a solution that allows adding interactive demonstrations to the team's application.
- **Software publishing** We added a section for team members to publish the software they developed and their respective demonstrations.
- **Demo creation helper** We implemented a helper guide to facilitate and walk team members through the creation of an interactive demonstration.
- **Improved security in production** HTTPS is now being used by default in production. Some other important settings were set for this environment.
- **Administration tasks automated** Common administration tasks such as publication fetching, image conversion and database backups were automated.

We think these modifications brought advantages for every user involved: On the one hand, visitors will have a more satisfactory and responsive experience navigating through the application and may even interactively try some of the team's work without installing or downloading

anything. On the other hand, team members now have a safer and more usable application, where they can interactively share their software and possibly arouse the visitors' interest in their work.

7.2 Future work

Despite the successful completion of this dissertation's objectives, we recognise that improvements could still be made to the team's application to enhance the solutions applied and make it better fit the members' needs and increase its usage. With these objectives in mind, we think that the most pertinent improvements to be considered in a future continuation of this work are:

Improve image serving Serving appropriate-sized images dependent on the screen size of the visitors' device and fallback images of different formats should be considered in a future continuation of this work. The first would reduce image sizes on smaller devices, thus making pages that have several images lighter and load faster. The second would solve the incorrect rendering of images in outdated browsers that lack support for some image formats, such as WebP.

Update application frameworks and libraries Many of the libraries that the application is built upon are outdated, and several have known vulnerabilities. A complete update may be a very time-consuming process, however, it will patch such vulnerabilities, thus making the application more secure, and it also will bring new features.

Deploy container to store backups Due to the lack of availability of another server with redundant storage, the container developed to store the database dumps was never deployed, thus leaving them stored in the application's VM. In a future continuation of this work, it would be favourable to deploy it, so that database backups get stored in a separate machine, thus reducing the chance of data loss in case of a hardware failure.

Feature suggestions that were not implemented Some of the features suggested by the team members described in chapter 5 and appendix A.15 that were not implemented, such as the incorporation of the team's newsletter or the implementation of a personal area for each team member with the nearest incoming events, are functionalities that would have a significant impact in making the team's application more used by team members and therefore should be considered in a future continuation of this work.

References

[1] IBM . IBM Docs - introduction to IMS. https://prod.ibmdocs-production-dal-6099123ce774e592a519d7c33db8265e-0000.us-south.containers.appdomain.cloud/docs/en/zos-basic-skills?topic=product-ims-database-manager, 2021.

[2] Wajid Ali, Muhammad Usman Shafique, Muhammad Arslan Majeed, and Ali Raza. Comparison between SQL and NoSQL databases and their relationship with big data analytics. *Asian Journal of Research in Computer Science*, pages 1–10, 2019.

[3] An overview of HTTP - HTTP | MDN. https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview.

[4] Charles Anderson. Docker [Software engineering]. *IEEE Software*, 32(3):102–c3, May 2015. Conference Name: IEEE Software.

[5] I K R Arthana, I M A Pradnyana, and G R Dantes. Usability testing on website wadaya based on ISO 9241-11. 1165:012012, 2019.

[6] Khaoula Benmoussa, Majida Laaziri, Samira Khoulji, Mohamed Larbi Kerkeb, and Abir El Yamami. Evaluating the usability of a moroccan university research management web platform. 32:1008–1016, 2019.

[7] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. The world-wide web. *Communications of the ACM*, 37(8):76–82, 1994. Publisher: ACM New York, NY, USA.

[8] David Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, September 2014. Conference Name: IEEE Cloud Computing.

[9] cloudflare. Why is HTTP not secure? | HTTP vs. HTTPS. https://www.cloudflare.com/learning/ssl/why-is-http-not-secure/.

[10] Francisco Curbera, William Nagy, and Sanjiva Weerawarana. Web services: Why and how. In *Workshop on Object-Oriented Web Services-OOPSLA*, volume 2001. Citeseer, 2001.

[11] Sanja Delcev and Drazen Draskovic. Modern JavaScript frameworks: A Survey Study. In *2018 Zooming Innovation in Consumer Technologies Conference (ZINC)*, pages 106–109, May 2018.

[12] Django. Django overview | django. https://www.djangoproject.com/start/overview/.

[13] Adrian Fernandez, Emilio Insfran, and Silvia Abrahão. Usability evaluation methods for the web: A systematic mapping study. 53(8):789–817, 2011.

[14] R.T. Fielding and R.N. Taylor. Principled design of the modern web architecture. 2(2):115–150, 2002.

[15] Flask. Foreword — flask documentation (2.0.x). `https://flask.palletsprojects.com/en/2.0.x/foreword/#what-does-micro-mean`.

[16] Martin Fowler and James Lewis. Microservices. 2014.

[17] Deka Ganesh Chandra. BASE analysis of NoSQL database. *Future Generation Computer Systems*, 52:13–21, 2015.

[18] Festim Halili and Erenis Ramadani. Web Services: A Comparison of Soap and Rest Services. *Modern Applied Science*, 12(3):175, February 2018.

[19] Mohammad A. Hassan. Relational and NoSQL databases: The appropriate database model choice. In *2021 22nd International Arab Conference on Information Technology (ACIT)*, pages 1–6, 2021.

[20] ISO. ISO/IEC 25010:2011(en), systems and software engineering — systems and software quality requirements and evaluation (SQuaRE) — system and software quality models. Standard, International Organization for Standardization, Geneva, CH, March 2011.

[21] ISO. Ergonomics of human-system interaction — Part 11: Usability: Definitions and concepts, ISO 9241-11:2018. Standard, International Organization for Standardization, Geneva, CH, March 2018.

[22] Patrick W. Jordan, B. Thomas, Ian Lyall McClelland, and Bernard Weerdmeester. *Usability Evaluation In Industry*. CRC Press, 1996. Google-Books-ID: IfUsRmzAqvEC.

[23] Marin Kaluža, Marijana Kalanj, and Bernard Vukelić. A Comparison of Back-end Frameworks for Web Application Development. *Zbornik Veleučilišta u Rijeci*, 7(1):317–332, May 2019. Publisher: Veleučilište u Rijeci.

[24] Mohammad Ayoub Khan, Sanjay Gairola, Bhola Jha, and Pushkar Praveen. *Smart Computing: Proceedings of the 1st International Conference on Smart Machine Intelligence and Real-Time Computing (SmartCom 2020), 26-27 June 2020, Pauri, Garhwal, Uttarakhand, India*. CRC Press, June 2021. Google-Books-ID: PCs0EAAAQBAJ.

[25] Kamil Kolonko. Performance comparison of the most popular relational and non-relational database management systems. page 71, 2018.

[26] D. V. Kornienko, S. V. Mishina, and M. O. Melnikov. The single page application architecture when developing secure web services. 2091(1):012065, 2021. Publisher: IOP Publishing.

[27] Kai Lei, Yining Ma, and Zhi Tan. Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js. In *2014 IEEE 17th International Conference on Computational Science and Engineering*, pages 661–668, December 2014.

[28] Yishan Li and Sathiamoorthy Manoharan. A performance comparison of SQL and NoSQL databases. In *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pages 15–19. ISSN: 2154-5952.

[29] MongoDB. Document database - NoSQL. https://www.mongodb.com/document-databases.

[30] MongoDB. NoSQL vs SQL databases. https://www.mongodb.com/nosql-explained/nosql-vs-sql.

[31] MongoDB. What is an object-oriented database? https://www.mongodb.com/databases/what-is-an-object-oriented-database.

[32] NGINX. What is NGINX? https://www.nginx.com/resources/glossary/nginx/.

[33] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1994. Google-Books-ID: 95As2OF67f0C.

[34] Asil Oztekin, Alexander Nikov, and Selim Zaim. UWIS: An assessment methodology for usability of web-based information systems. 82(12):2038–2050, 2009.

[35] Amit M Potdar, Narayan D g, Shivaraj Kengond, and Mohammed Moin Mulla. Performance Evaluation of Docker Container and Virtual Machine. *Procedia Computer Science*, 171:1419–1428, January 2020.

[36] Kosovare Sahatqija, Jaumin Ajdari, Xhemal Zenuni, Bujar Raufi, and Florije Ismaili. Comparison between relational and NOSQL databases. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 0216–0221, 2018.

[37] Bryce Sasaki. Graph databases for beginners: ACID vs. BASE explained.

[38] Ahmed Seffah, Mohammad Donyaee, Rex B. Kline, and Harkirat K. Padda. Usability measurement and metrics: A consolidated model. 14(2):159–178, 2006.

[39] Anushka Sharma, Aman Karamchandani, Devam Dave, Arush Patel, and Nishant Doshi. Database management systems—an efficient, effective, and augmented approach for organizations. In Tomonobu Senjyu, Parikshit N. Mahalle, Thinagaran Perumal, and Amit Joshi, editors, *ICT with Intelligent Applications*, Smart Innovation, Systems and Technologies, pages 465–478. Springer, 2022.

[40] Leon Shklar and Rich Rosen. *Web application architecture: principles, protocols, and practices*. John Wiley, Chichester, England ; Hoboken, NJ, 2003.

[41] SQLite. Appropriate uses for SQLite. https://www.sqlite.org/whentouse.html.

[42] Tony Dwi Susanto, Anisa Ingesti Prasetyo, and Hanim Maria Astuti. Web usability evaluation on BloobIS website by using hallway usability testing method and ISO 9241:11. 974:012043, 2018.

[43] Vue. Introduction | vue.js. https://vuejs.org/guide/introduction.html#what-is-vue.

[44] Xili Wan, Xinjie Guan, Tianjing Wang, Guangwei Bai, and Baek-Yong Choi. Application deployment using Microservice and Docker containers: Framework and optimization. *Journal of Network and Computer Applications*, 119:97–109, October 2018.

[45] What is a REST API? examples, uses, and challenges. https://blog.postman.com/rest-api-examples/.

[46] What is the difference between webpage, website, web server, and search engine? - Learn web development | MDN. https://developer.mozilla.org/en-US/docs/Learn/Common_questions/Pages_sites_servers_and_search_engines.

Appendix A

A.1 Interview with team members

A.1.1 Questions

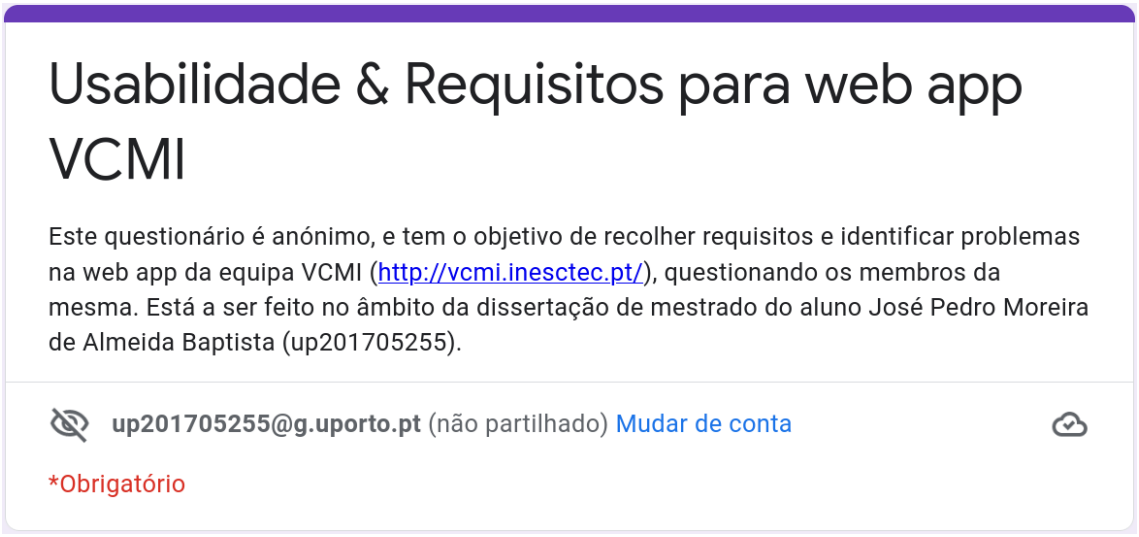


Figure A.1: Interview questions - Introduction.

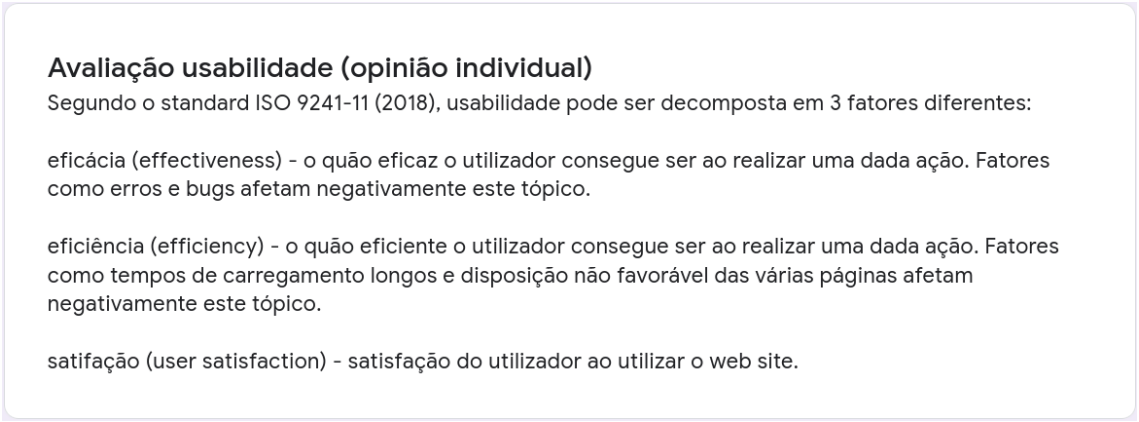


Figure A.2: Interview questions - Usability Introduction.

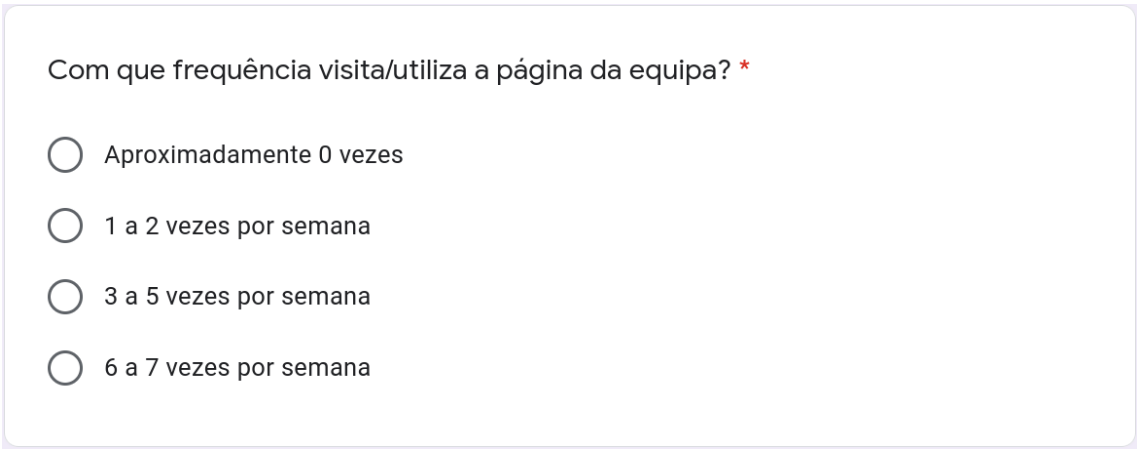


Figure A.3: Interview questions - Web application usage.

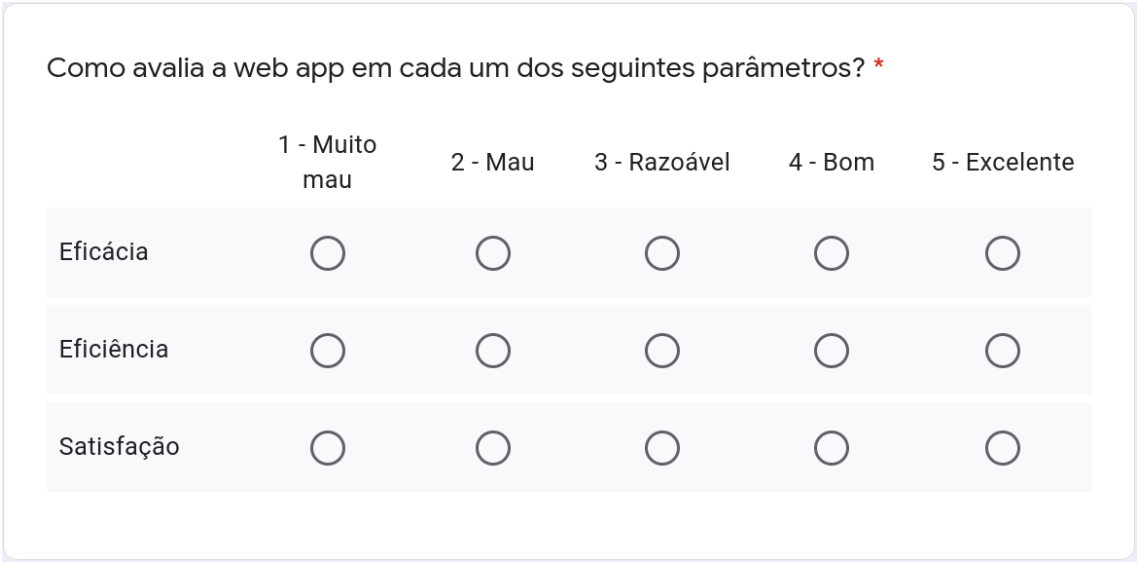


Figure A.4: Interview questions - Usability evaluation.

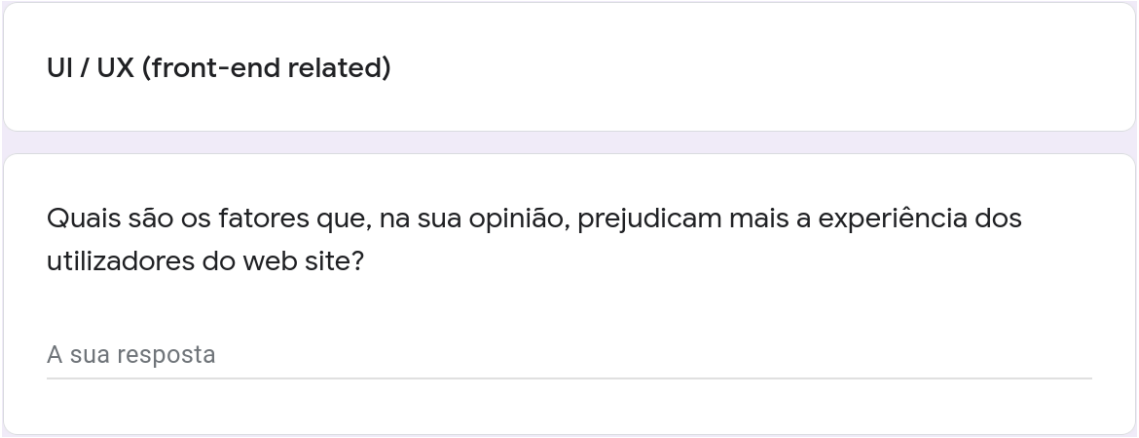


Figure A.5: Interview questions - Issues affecting users' experience.

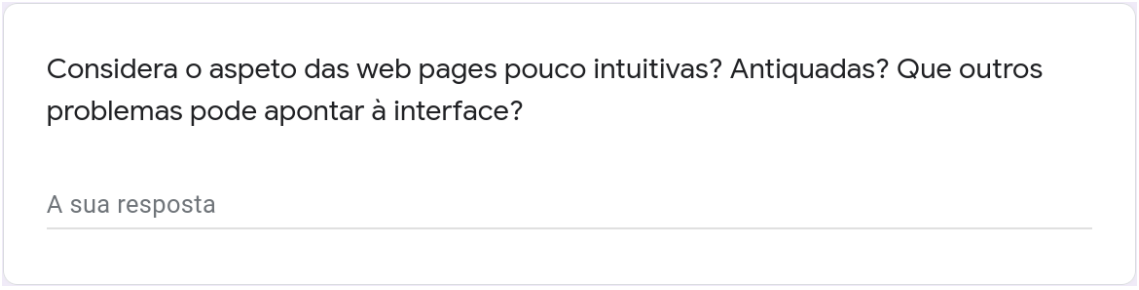


Figure A.6: Interview questions - User interface issues.

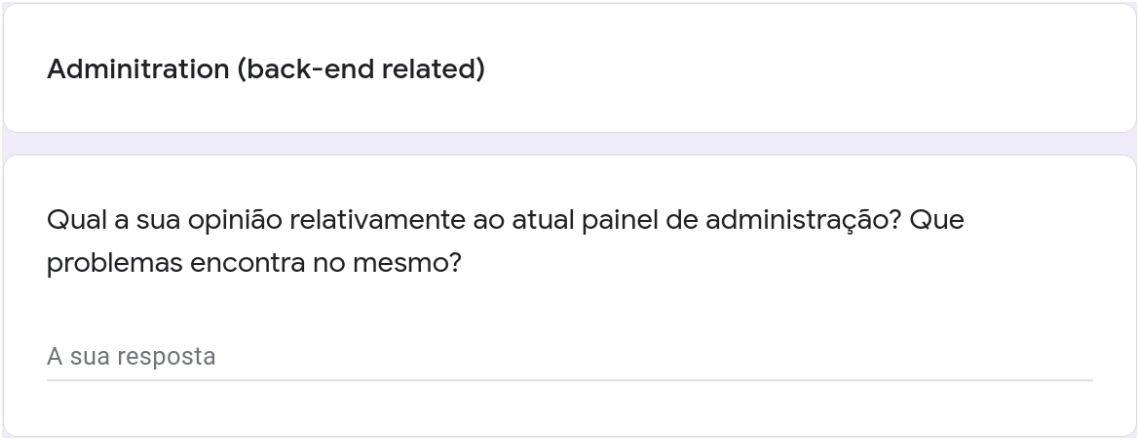


Figure A.7: Interview questions - Admin panel issues.



Figure A.8: Interview questions - Back-end issues.

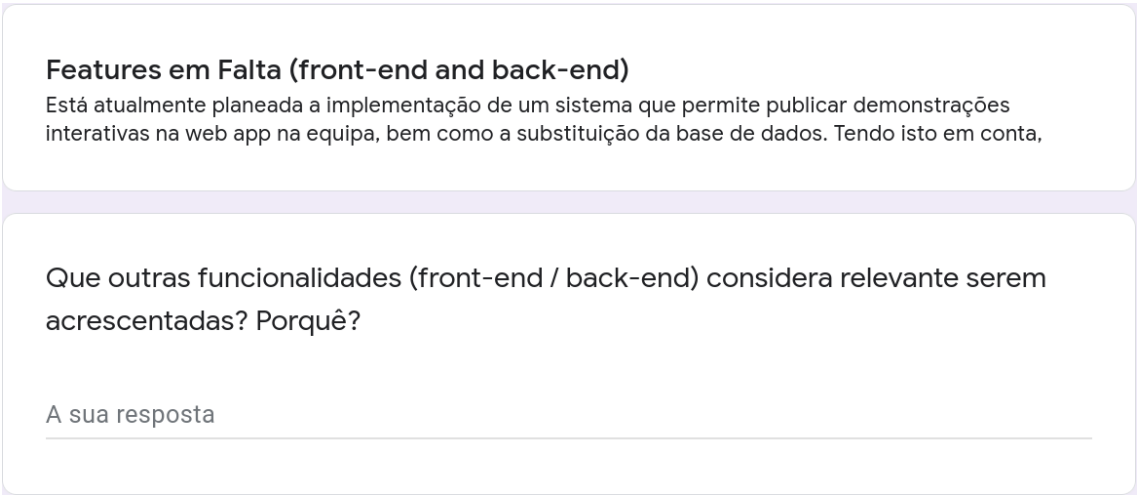


Figure A.9: Interview questions - Missing features.

A.1.2 Results

A.1.2.1 Usability related

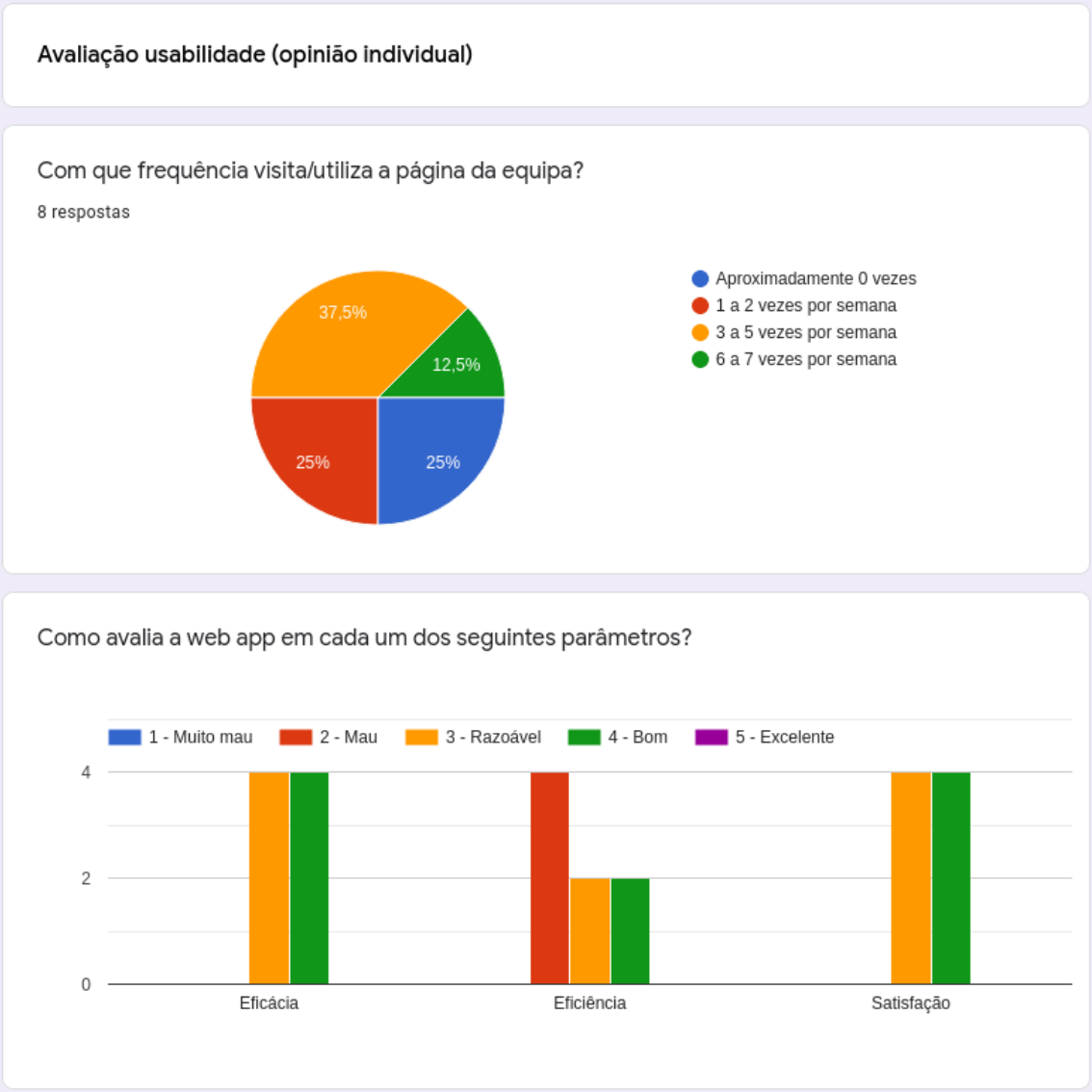


Figure A.10: Usability interview section.

A.1.2.2 Issue enumeration

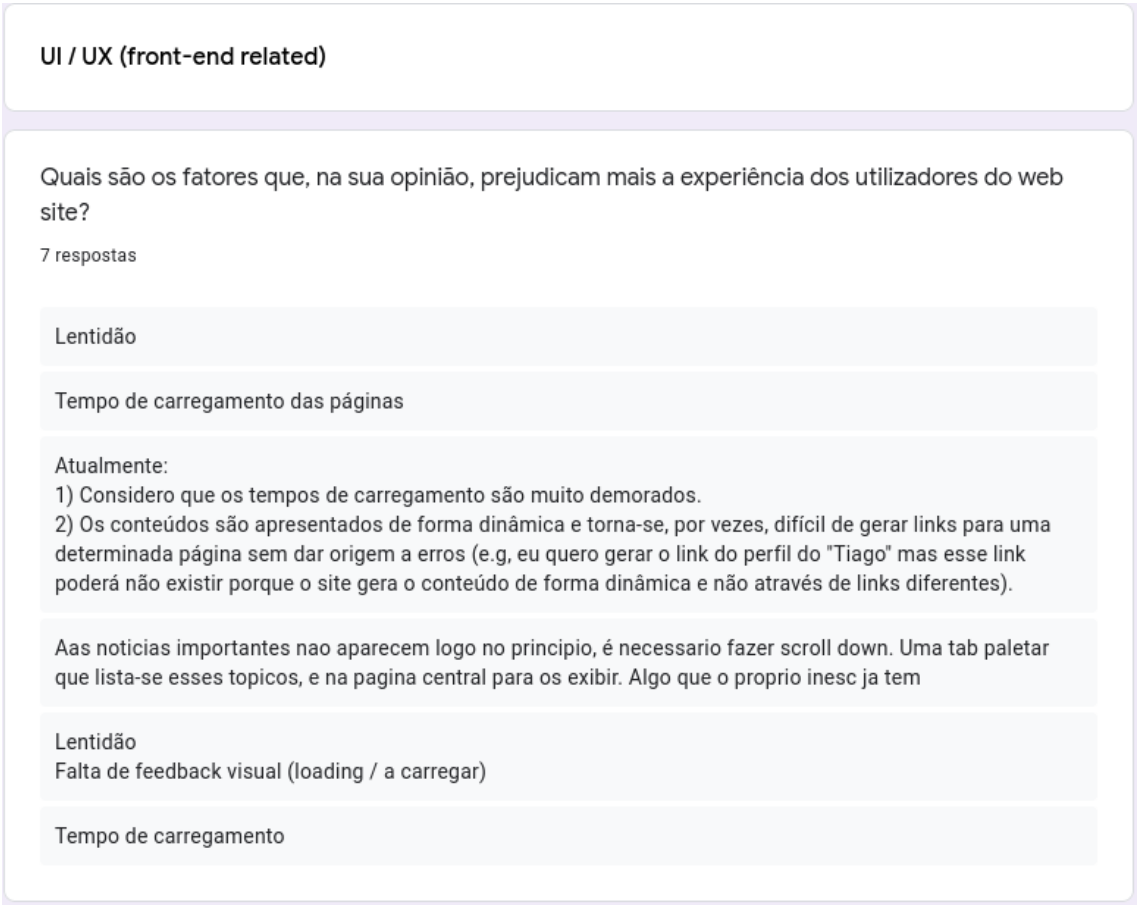


Figure A.11: Front-end issue enumeration 1.

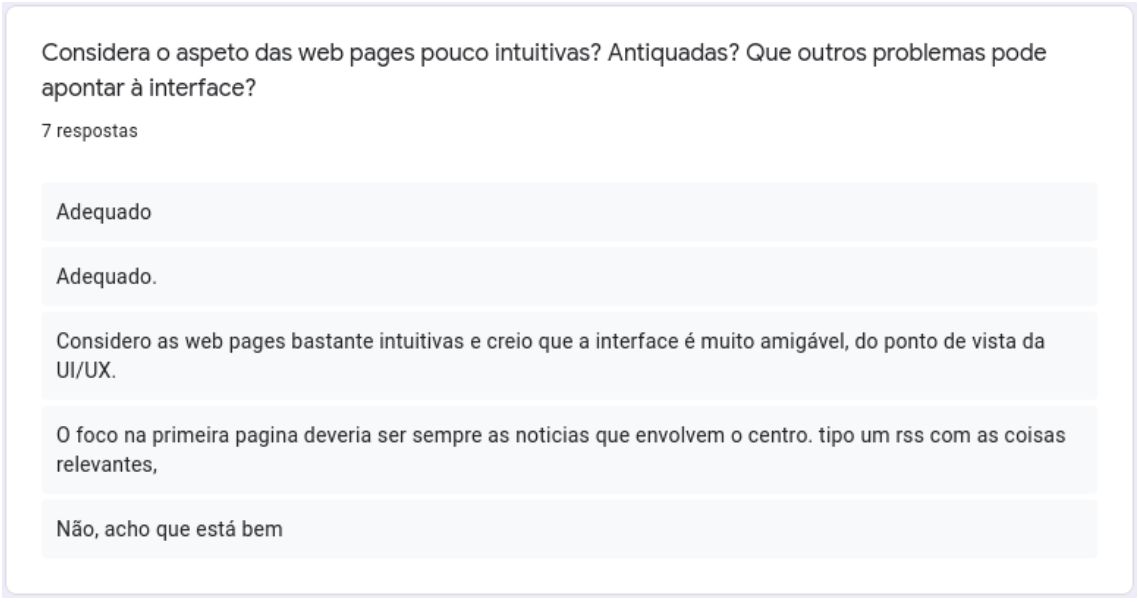


Figure A.12: Front-end issue enumeration 2.

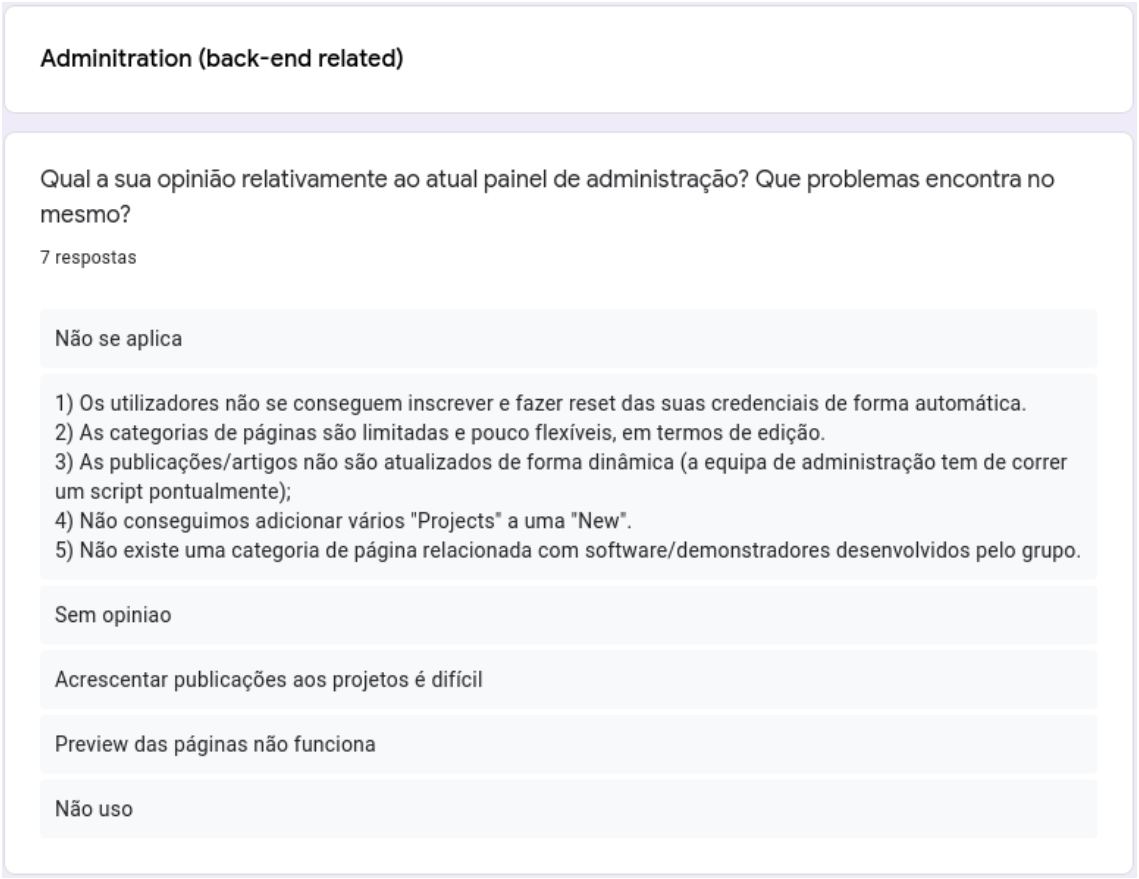


Figure A.13: Back-end issue enumeration 1.

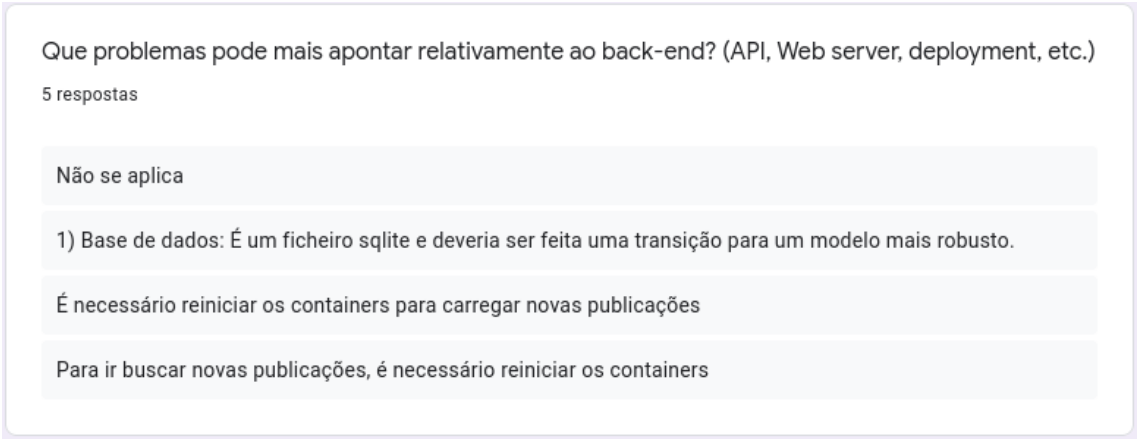


Figure A.14: Back-end issue enumeration 2.

A.1.2.3 Feature suggestions



Figure A.15: Feature suggestion section.

A.2 Interactive demonstration questionnaire

A.2.1 Questions

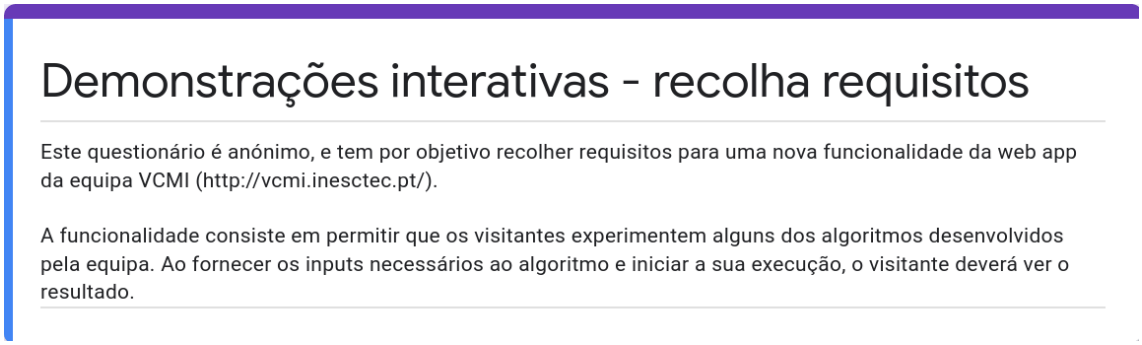


Figure A.16: Interactive demo questionnaire - Introduction.

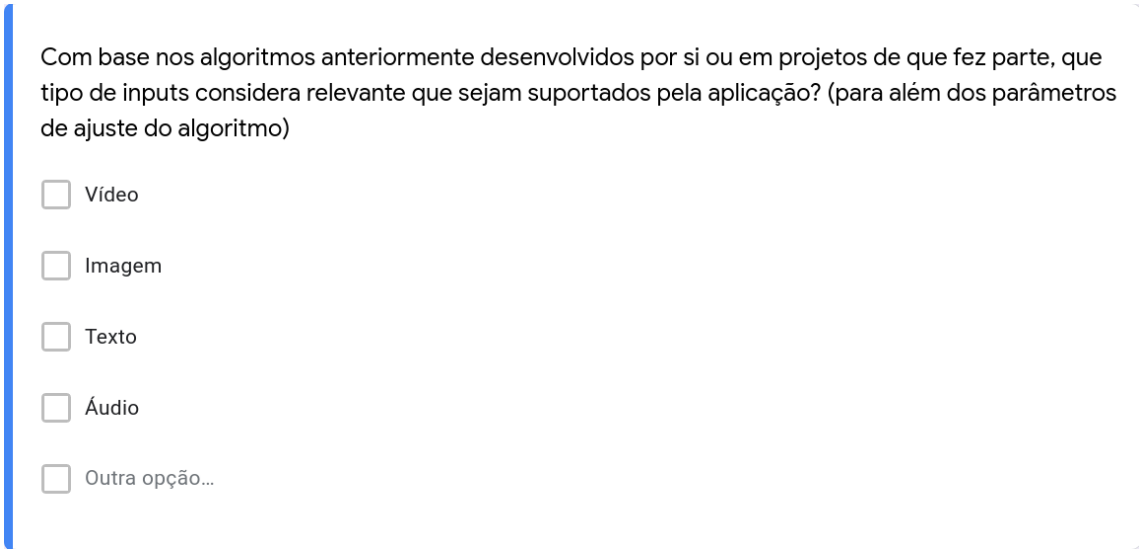


Figure A.17: Interactive demo questionnaire - Question 1.

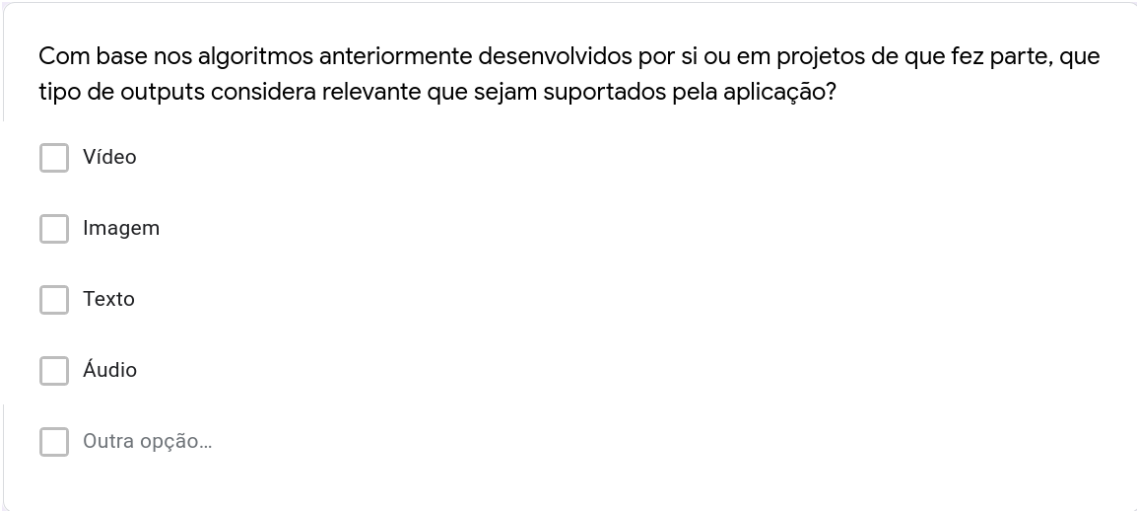


Figure A.18: Interactive demo questionnaire - Question 2.

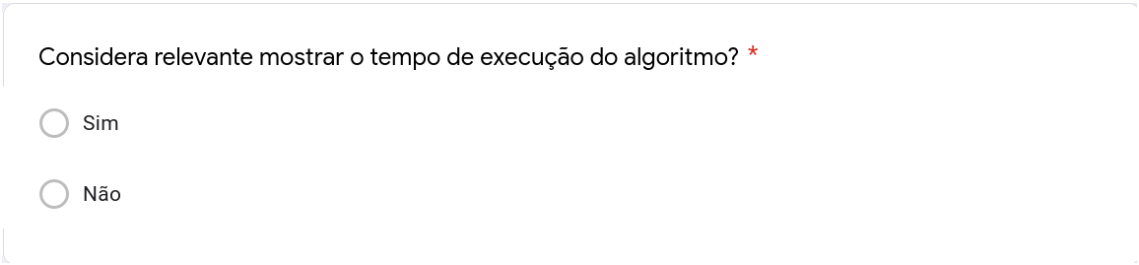


Figure A.19: Interactive demo questionnaire - Question 3.

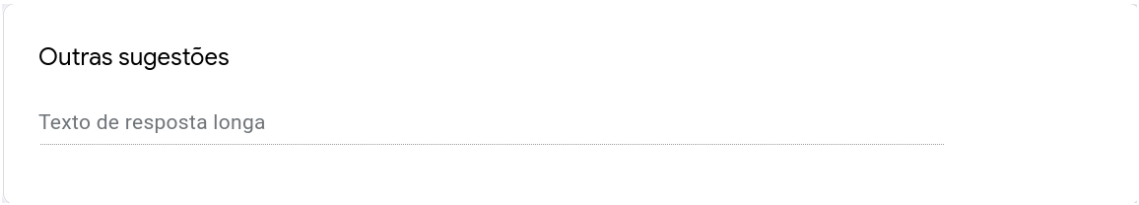


Figure A.20: Interactive demo questionnaire - Question 4.

A.2.2 Results

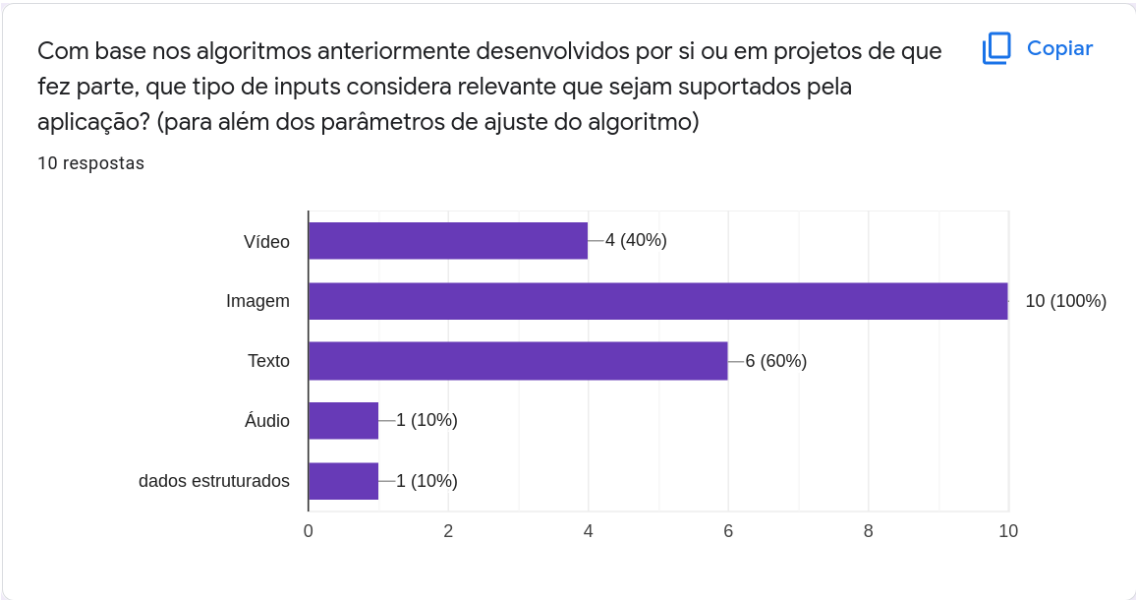


Figure A.21: Interactive demo questionnaire - Question 1 results.

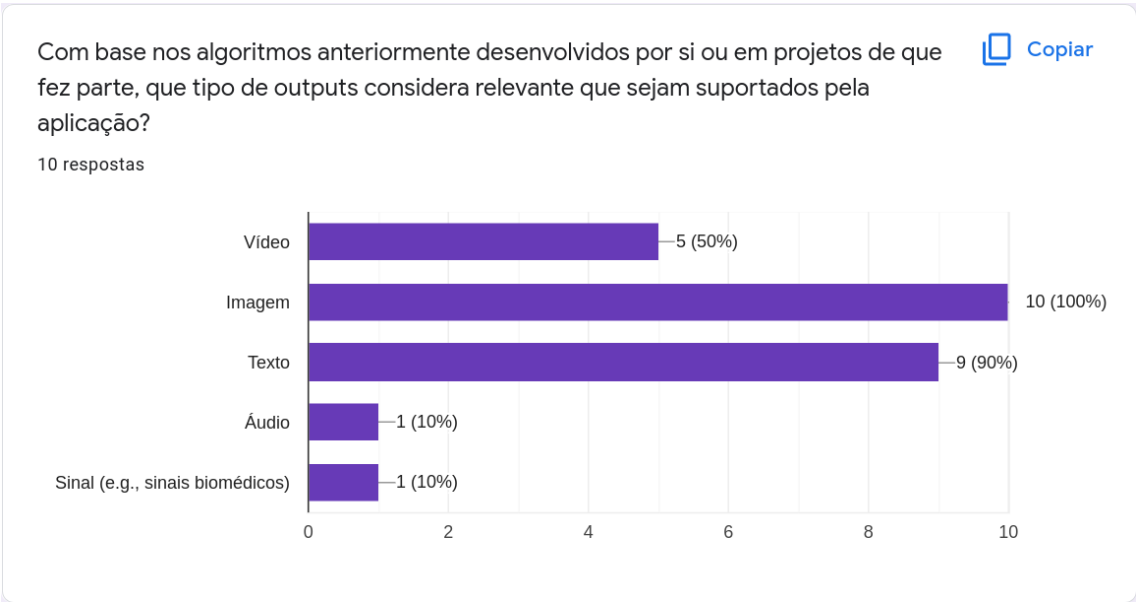


Figure A.22: Interactive demo questionnaire - Question 2 results.

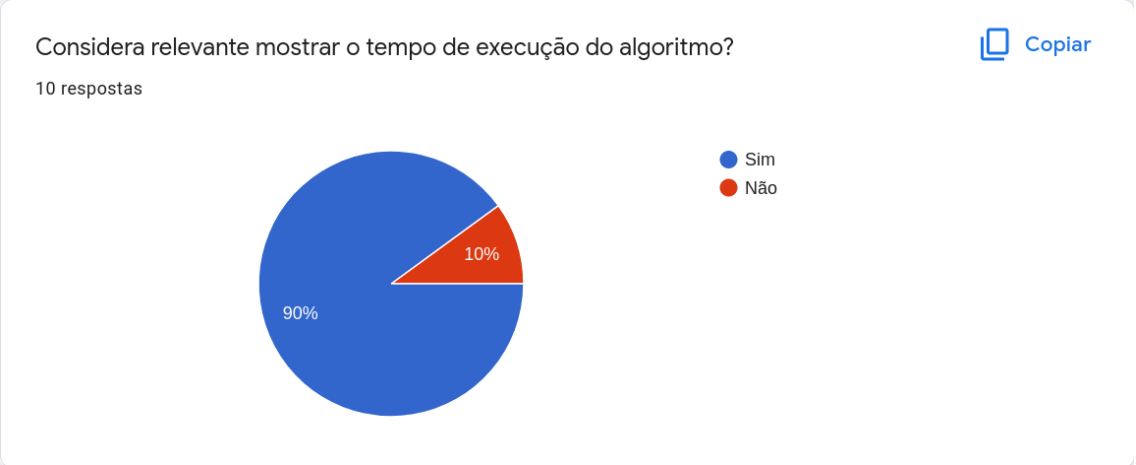


Figure A.23: Interactive demo questionnaire - Question 3 results.



Figure A.24: Interactive demo questionnaire - Question 4 results.

A.3 Initial database diagram

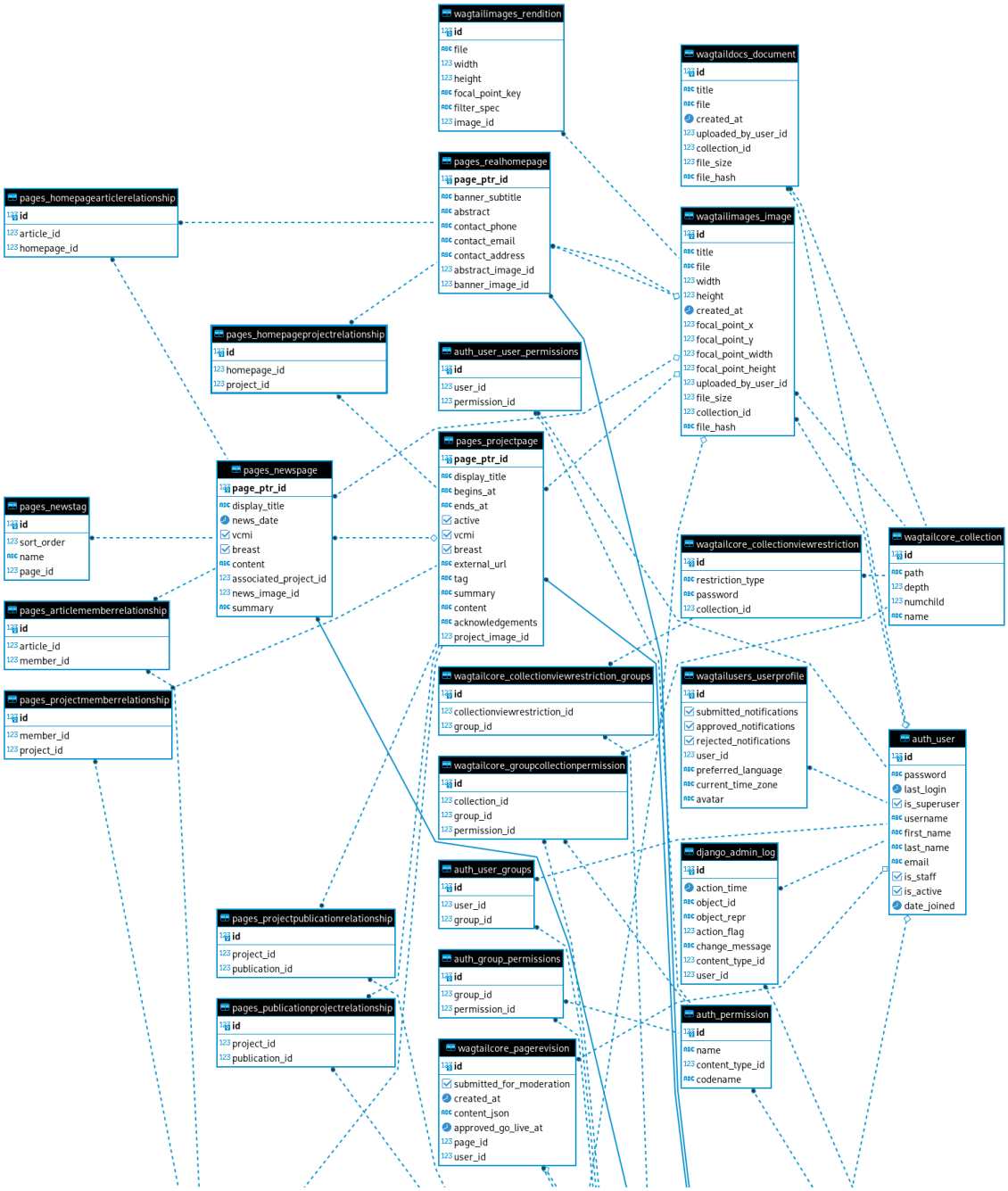


Figure A.25: Database diagram - part 1.

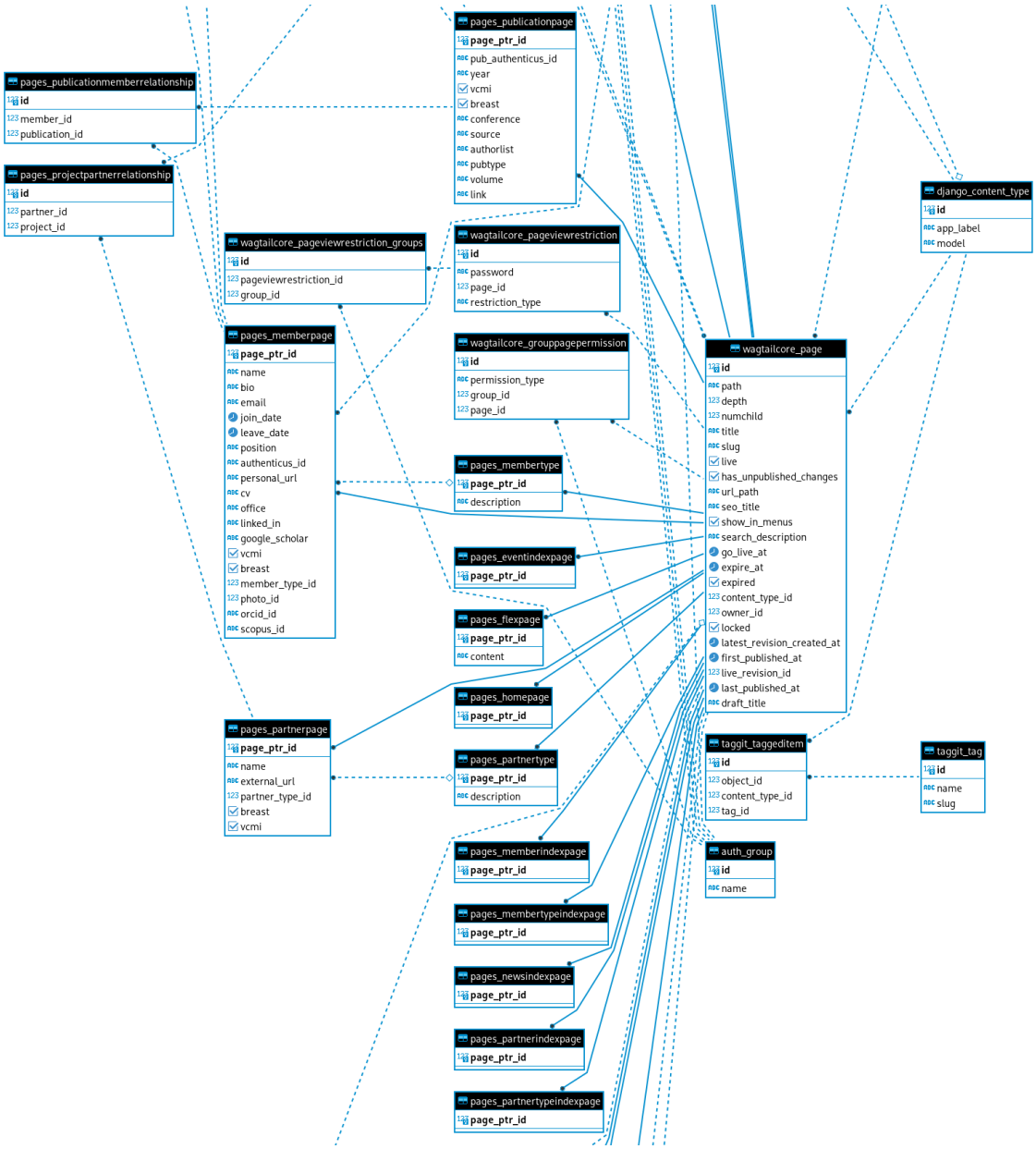


Figure A.26: Database diagram - part 2.

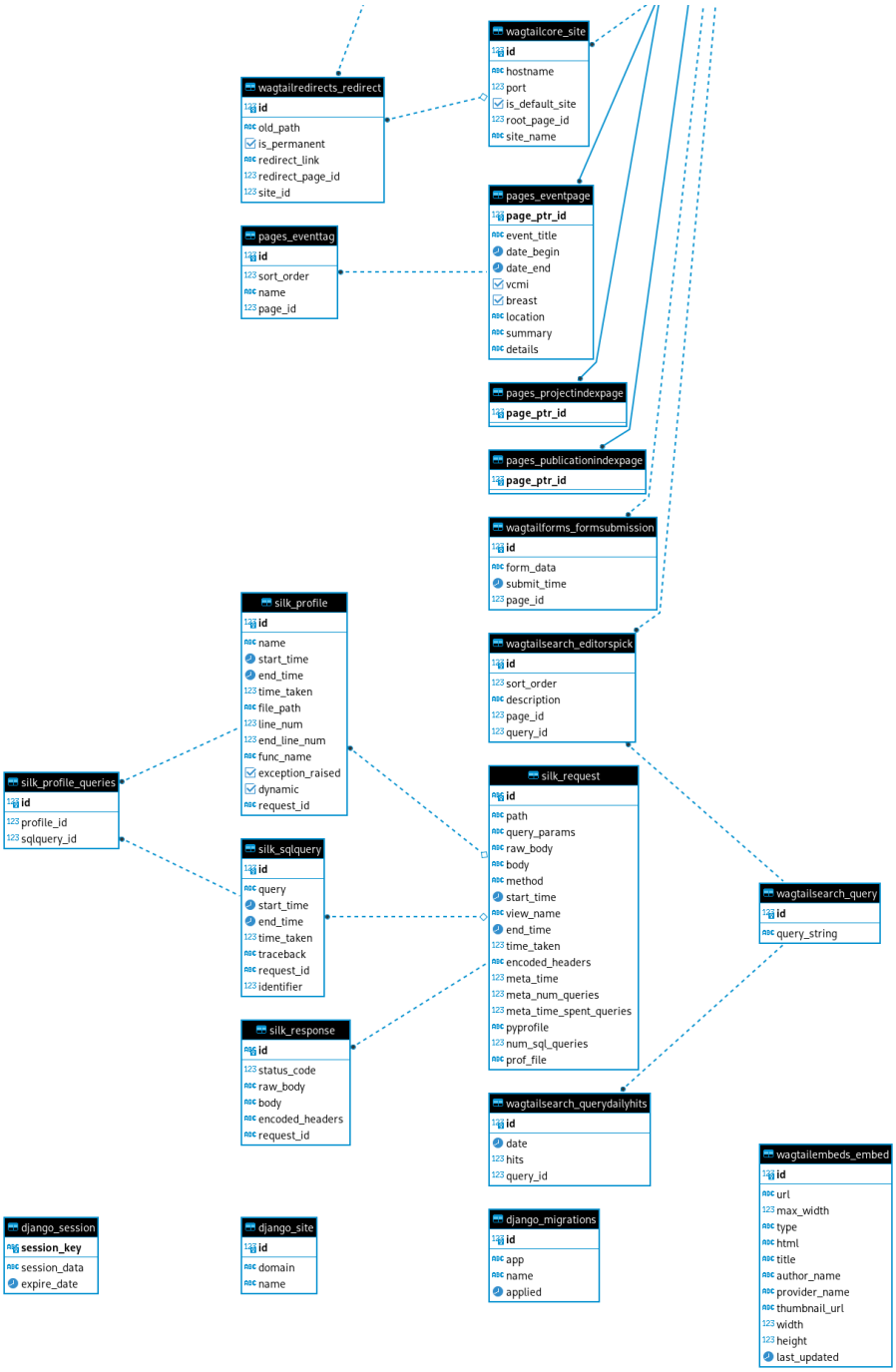


Figure A.27: Database diagram - part 3.

A.4 First usability evaluation: SUS

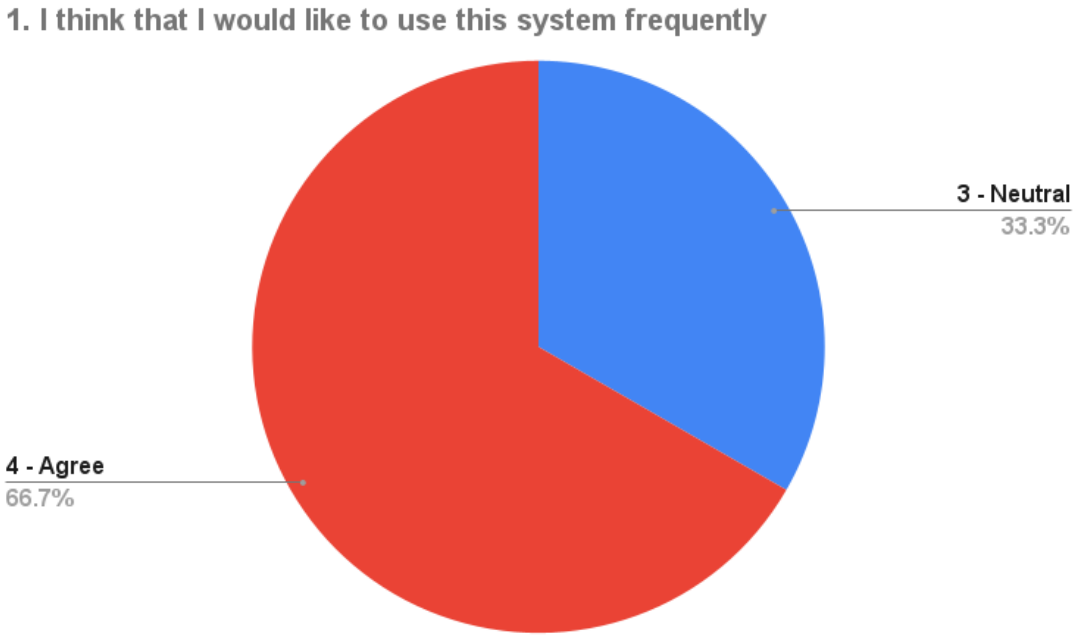


Figure A.28: Usability evaluation 1 - SUS - item 1.

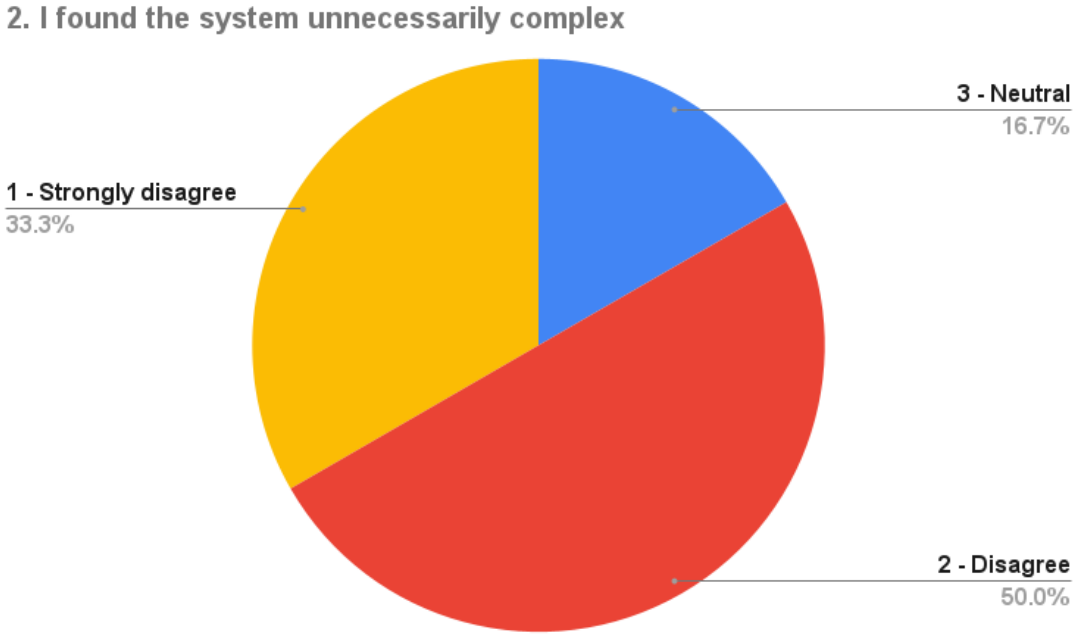


Figure A.29: Usability evaluation 1 - SUS - item 2.

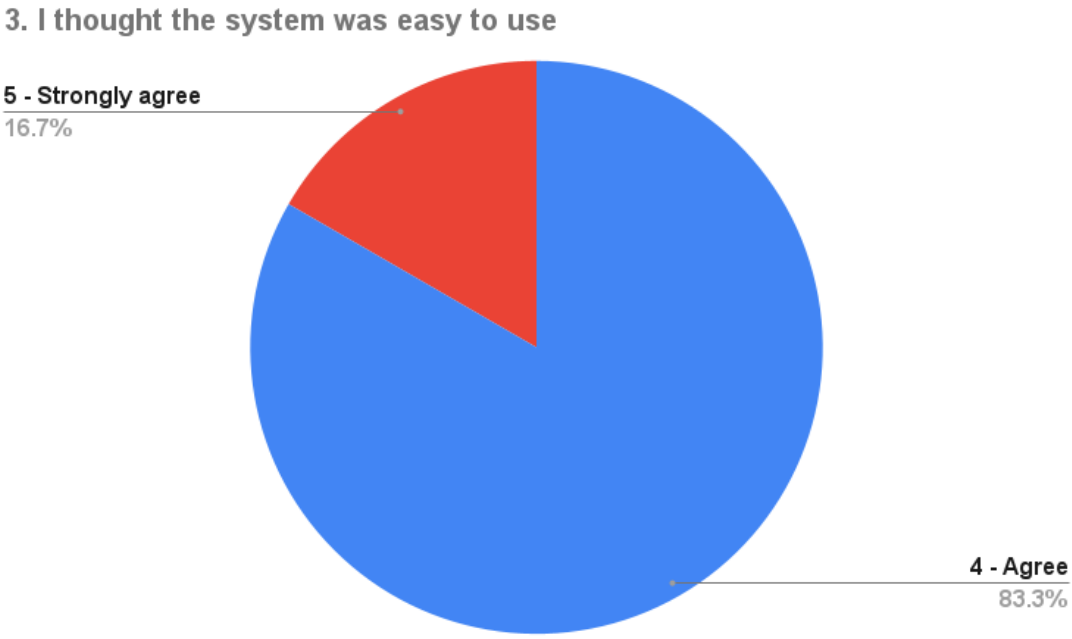


Figure A.30: Usability evaluation 1 - SUS - item 3.

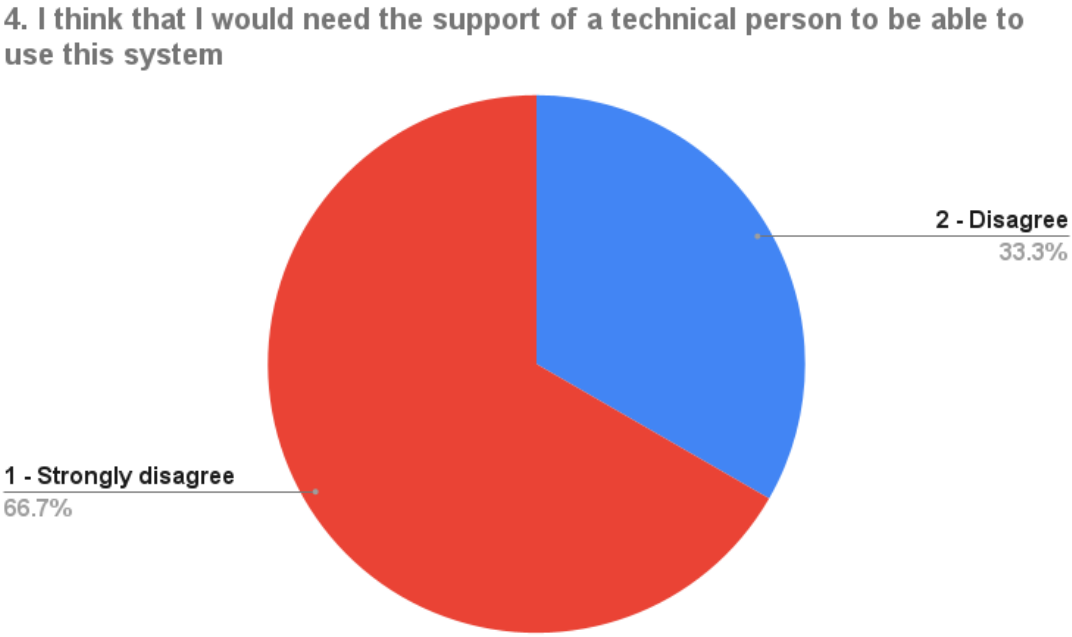


Figure A.31: Usability evaluation 1 - SUS - item 4.

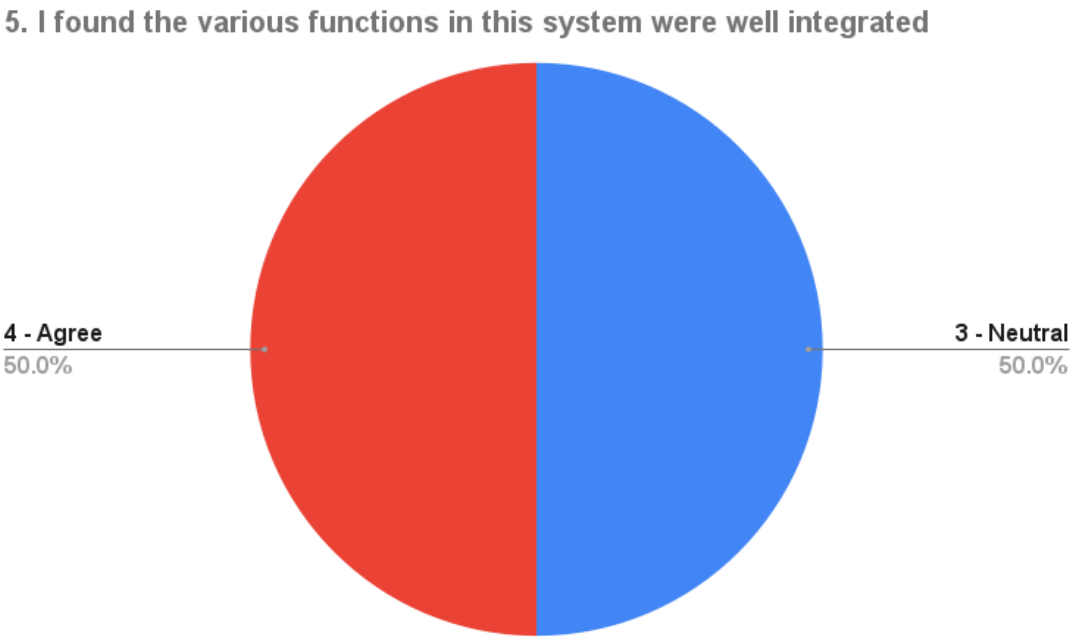


Figure A.32: Usability evaluation 1 - SUS - item 5.

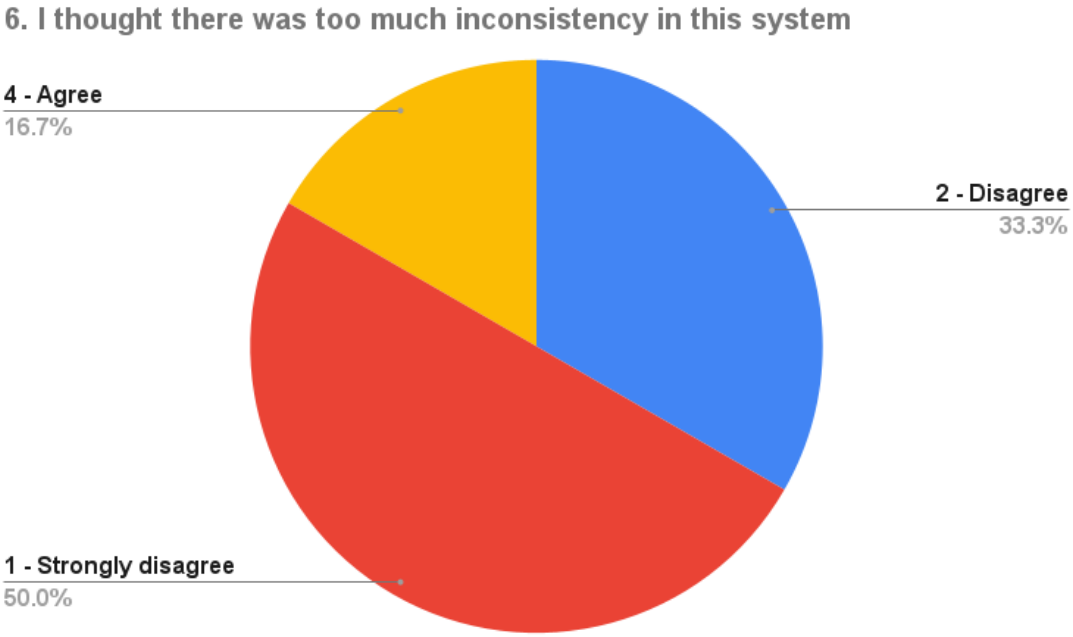


Figure A.33: Usability evaluation 1 - SUS - item 6.

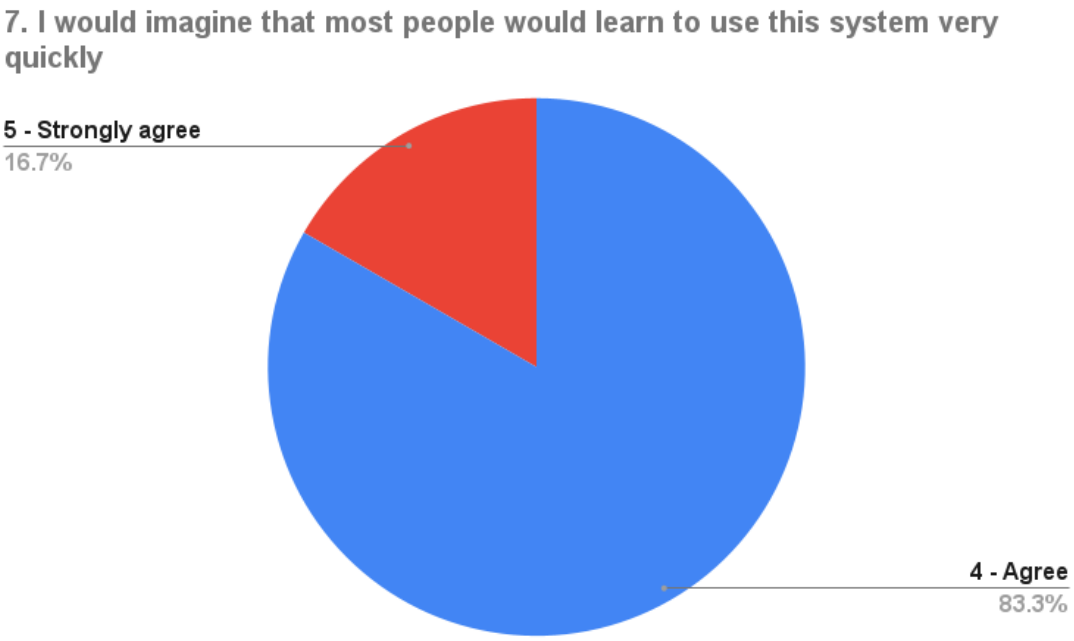


Figure A.34: Usability evaluation 1 - SUS - item 7.

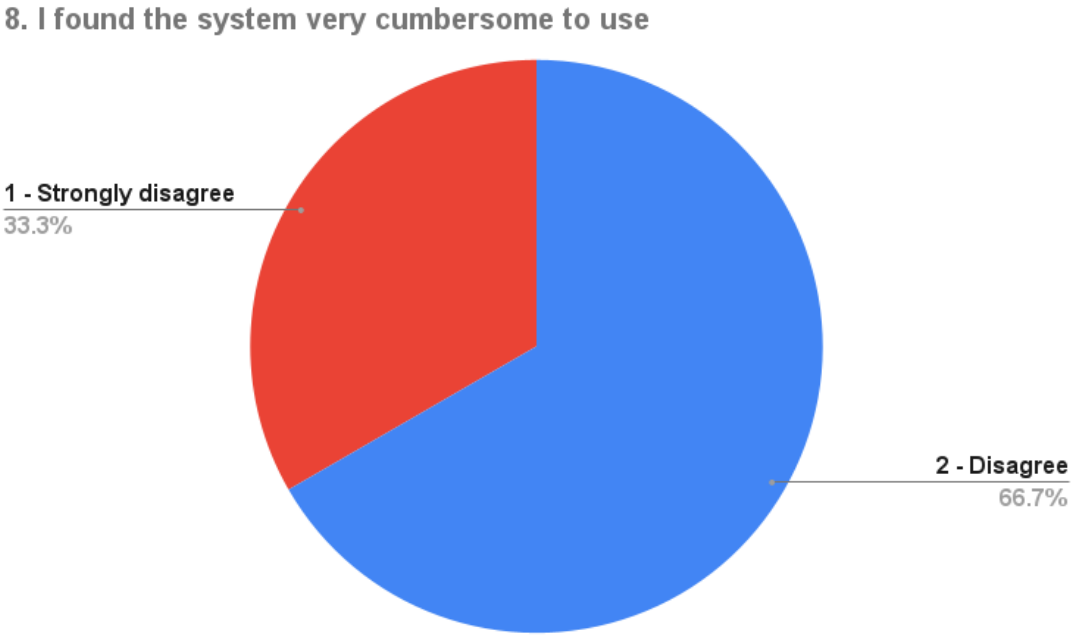


Figure A.35: Usability evaluation 1 - SUS - item 8.

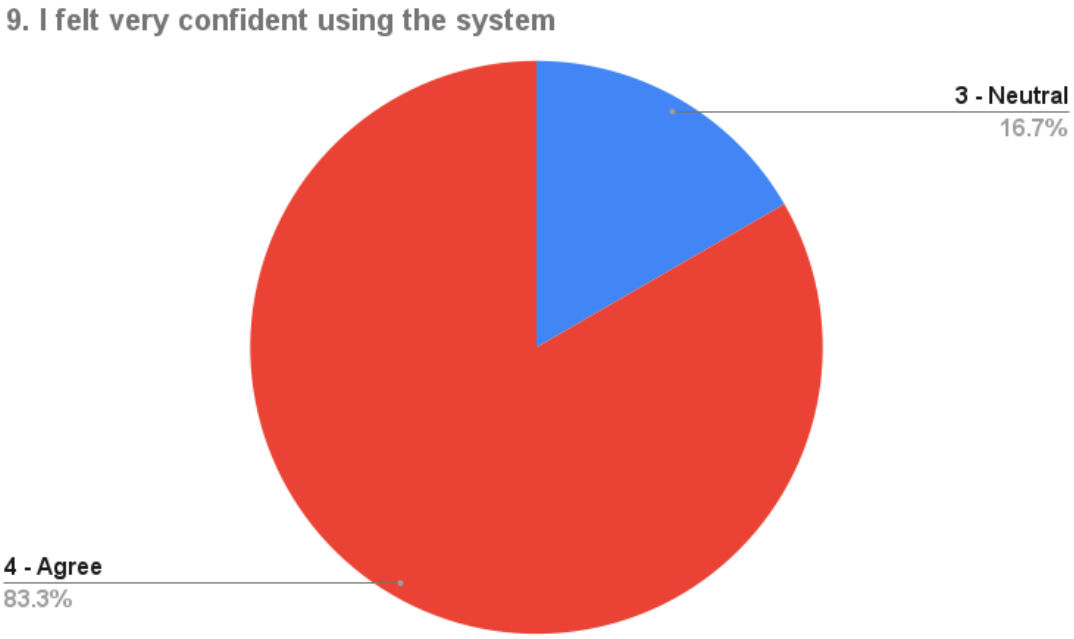


Figure A.36: Usability evaluation 1 - SUS - item 9.

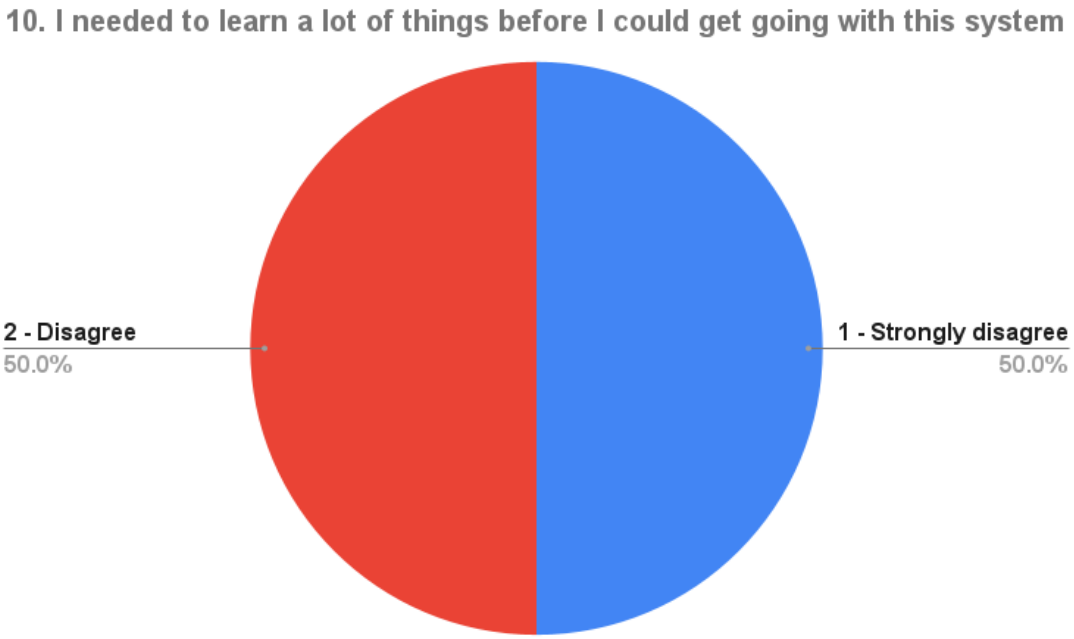


Figure A.37: Usability evaluation 1 - SUS - item 10.

A.5 Second usability evaluation: SUS

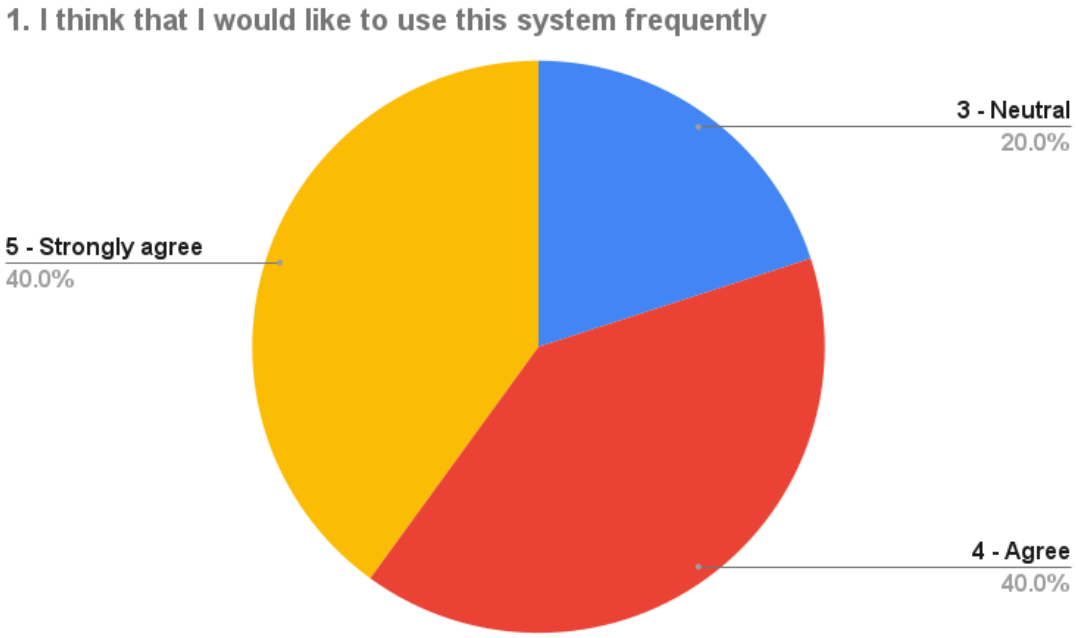


Figure A.38: Usability evaluation 2 - SUS - item 1.

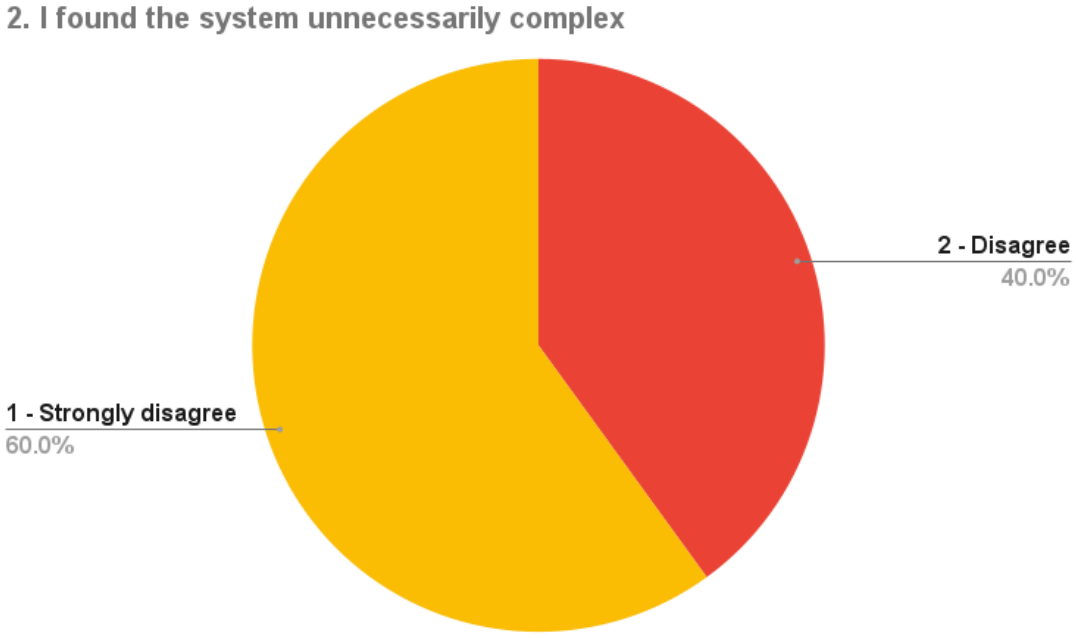


Figure A.39: Usability evaluation 2 - SUS - item 2.

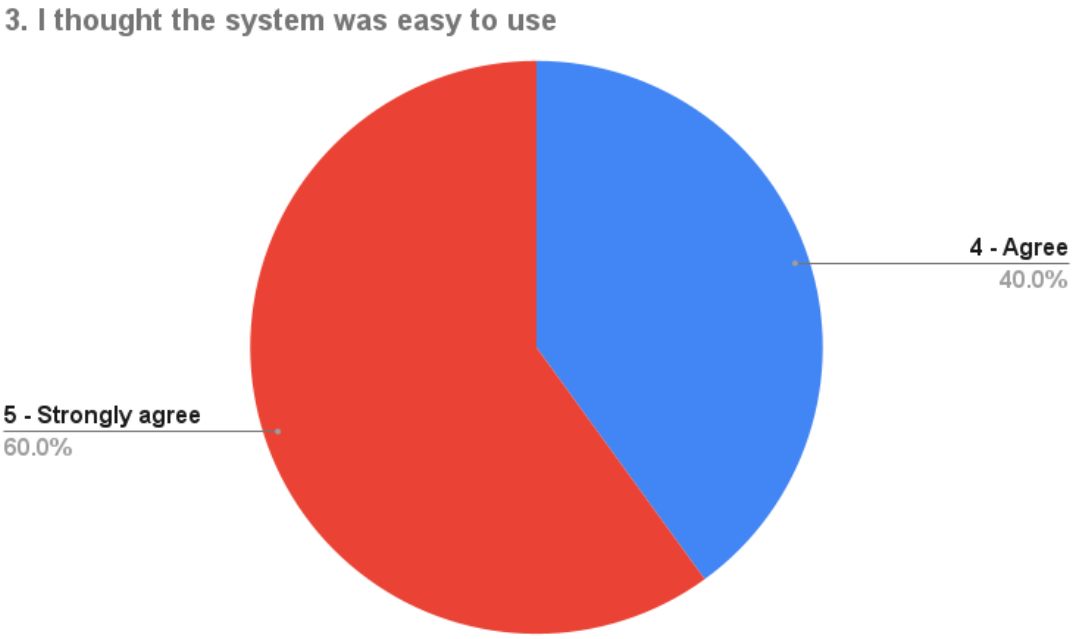


Figure A.40: Usability evaluation 2 - SUS - item 3.

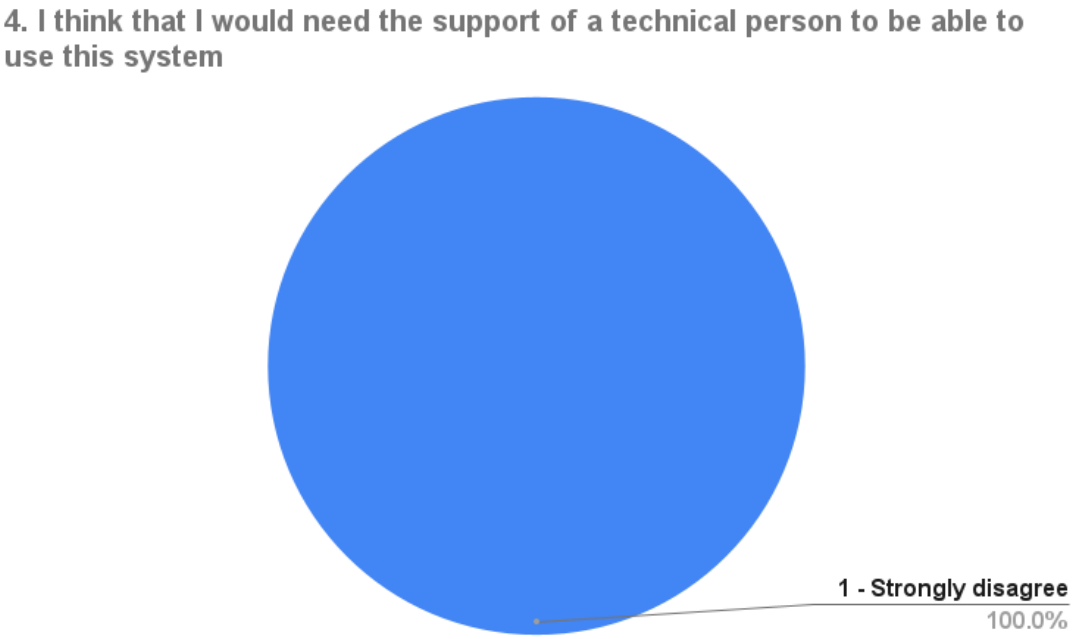


Figure A.41: Usability evaluation 2 - SUS - item 4.

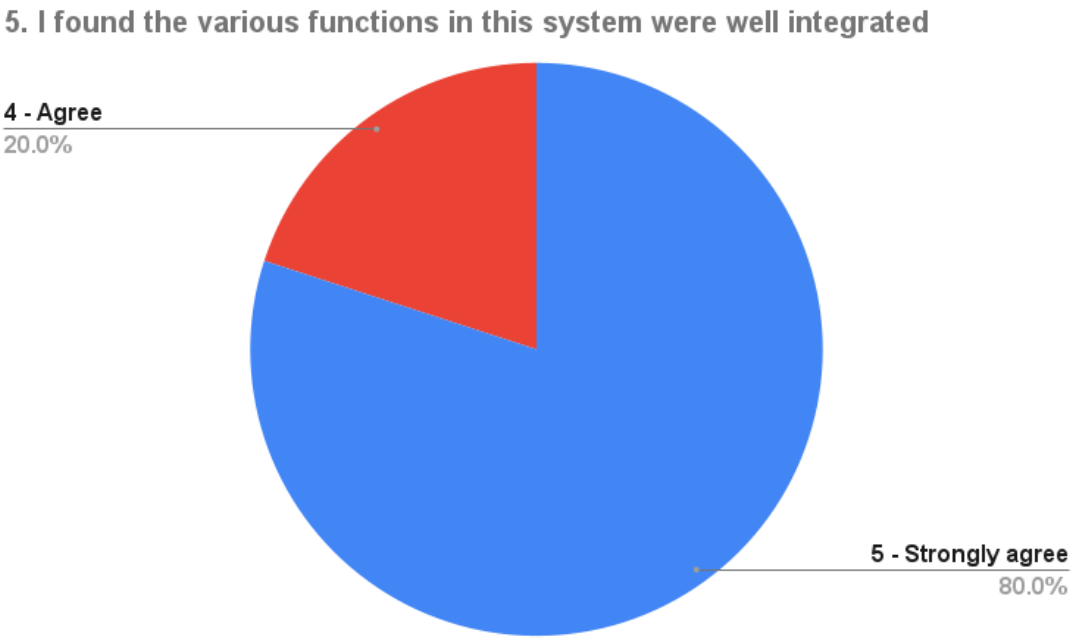


Figure A.42: Usability evaluation 2 - SUS - item 5.

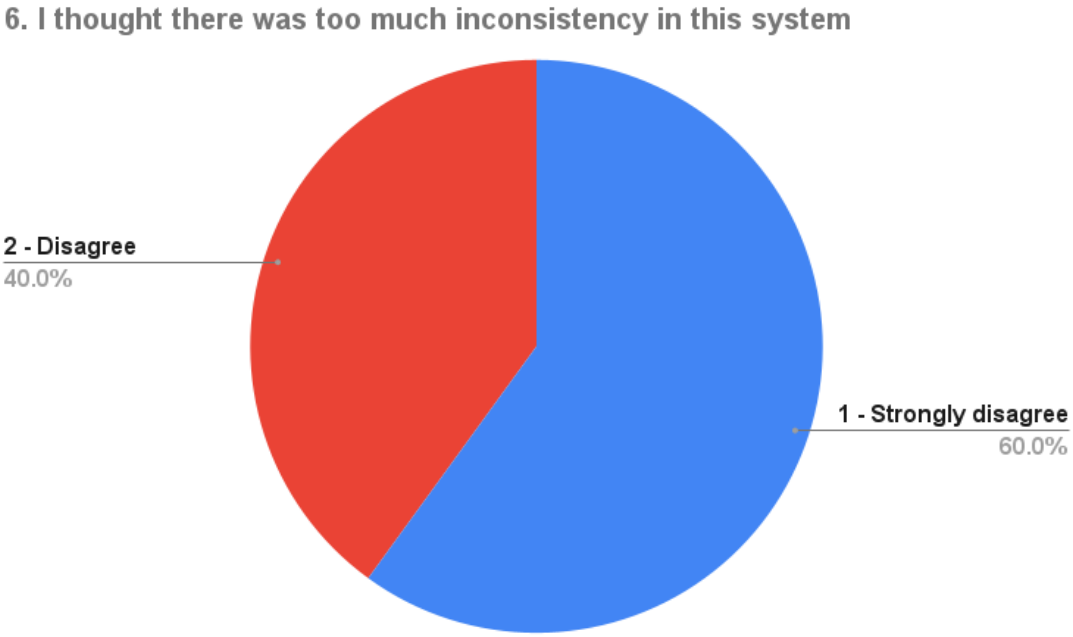


Figure A.43: Usability evaluation 2 - SUS - item 6.

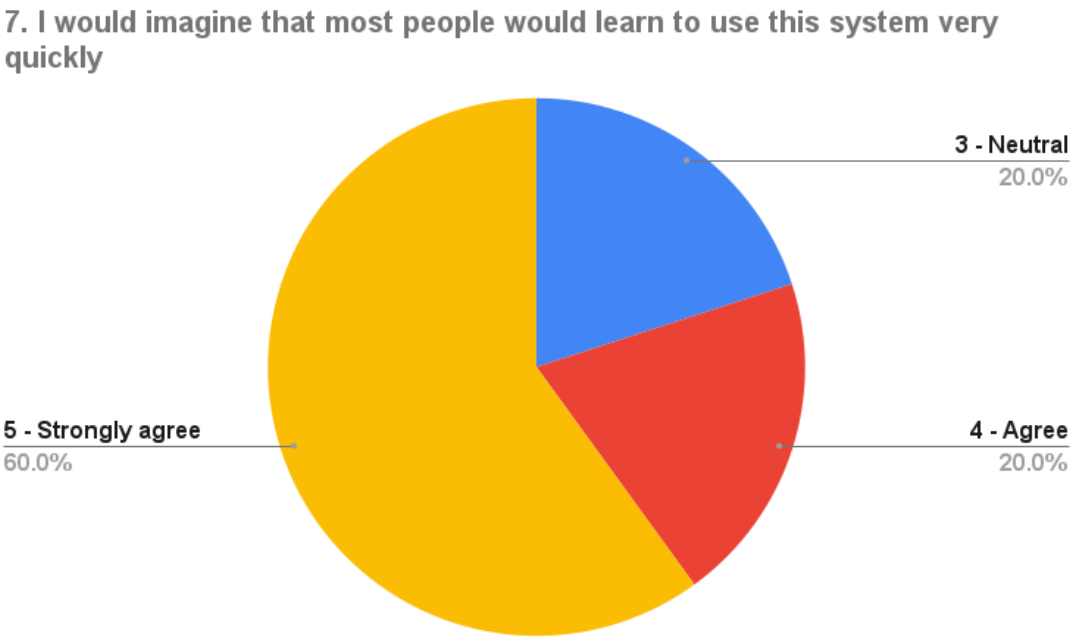


Figure A.44: Usability evaluation 2 - SUS - item 7.

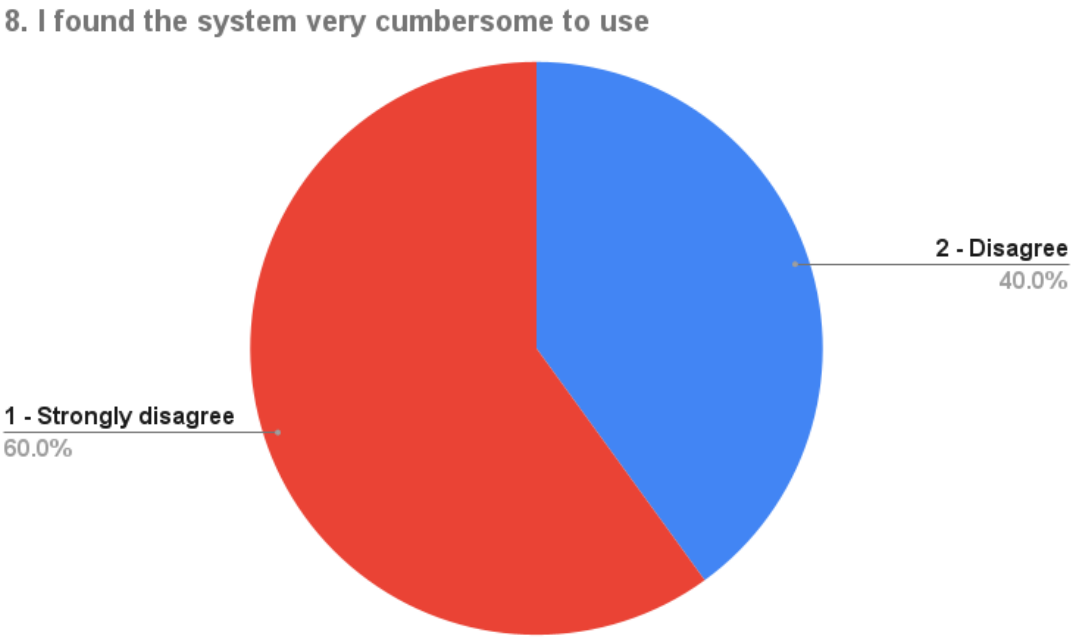


Figure A.45: Usability evaluation 2 - SUS - item 8.

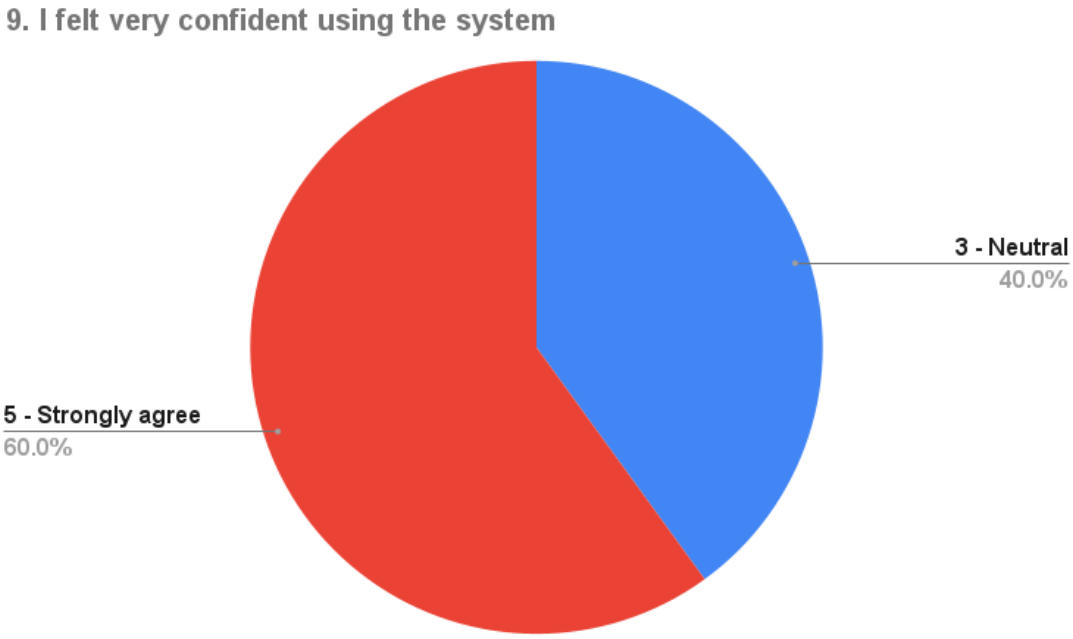


Figure A.46: Usability evaluation 2 - SUS - item 9.

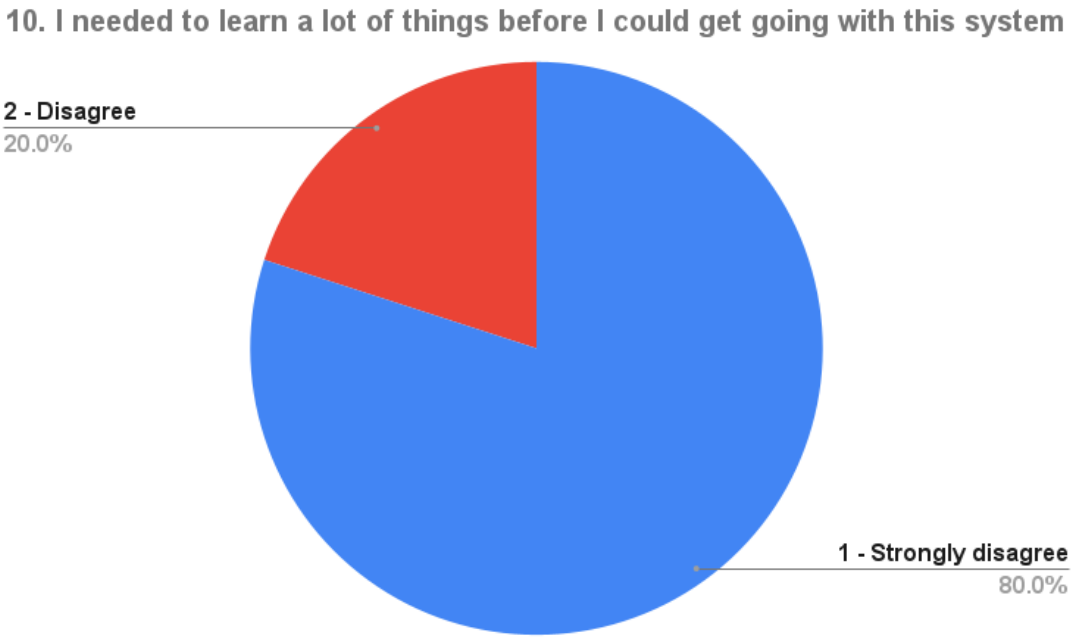


Figure A.47: Usability evaluation 2 - SUS - item 10.