UNIVERSIDADE
LUSÓFONA

DEISI25
AWS Resource Management: A Private Collaborative Management Portal

End of Course Work

Final Report

Alexandre Marques | a22202130
António Antunes | a22202276
Supervisor: João Caldeira
Co-Supervisor: Luís Gomes

TFC | LEI | January, 2024

Copyright

Resumo

A *cloud* revolucionou a maneira como a sociedade interage com a tecnologia nos dias de hoje. Permitiu que as empresas construissem os seus negócios com base numa infraestrutura que suporta quase todos tipo de necessidade que se possa imaginar e possibilitou a distribuição em massa desses mesmos negócios e tecnologias para os consumidores. Hoje em dia, quase toda as tecnologia que usamos está de uma maneira ou de outra ligada aquilo que chamamos a *cloud*.

A cloud não é nada mais que um conjunto de serviços que são disponibilizados através da internet que possibilitam a criação das mais diversas tecnologias com uma distribuição em massa. Esses serviços incluem computação, armazenamento e bases de dados. O que faz da cloud uma área revolucionaria é a facilidade e a flexibilidade que oferece no que diz respeito à gestão desses mesmos recursos e também a eficiência de custos que proporciona.

Existem várias plataformas com bastante peso dentro deste ecossistema, no entanto a plataforma com maior quota de mercado é a Amazon Web Services (AWS). A AWS oferece uma extensa oferta de serviços que abrangem todo o tipo de soluções dentro da área, desde soluções de computação e armazenamento a serviços avançados de Machine Learning (ML). E a parte mais relevante é que oferece tudo isto a um custo bastante reduzido.

Embora a plataforma da AWS ofereça um vasto número de soluções a baixo custo, acaba por pecar noutros pontos, como por exemplo, na complexidade da plataforma. A plataforma da AWS acaba por não ser muito *user-friendly* e algumas das razões que levam a isso é o vasto número de serviços que oferece e a complexidade dos mesmos.

O objetivo deste TFC, passa então por apresentar uma solução, que de forma centralizada, simplifique, num contexto empresarial, a forma de reservar e gerir os serviços que a plataforma oferece. A isto, acrescenta-se ainda a ambição em reduzir a complexidade ao ponto de qualquer pessoa que tenha um conhecimento básico de cloud consiga gerir a plataforma de forma eficiente.

De forma a tornar este modelo possível, no processo de desenvolvimento foi criado uma base para um portal de gerenciamento que permite a criação de novos serviços, tendo sido incluído o serviço EC2 de Amazon Web Services (AWS), acompanhado com documentação, base de dados e recursos adicionais de utilizadores, sendo possível a sua expansão e continuidade no futuro.

Abstract

The *cloud* has revolutionized how society interacts with technology today. It has enabled businesses to build their operations on an infrastructure that supports almost every kind of need imaginable and has facilitated the mass distribution of these businesses and technologies to consumers. Nowadays, almost all the technology we use is in one way or another linked to what we call the *cloud*.

The cloud is nothing more than a set of services delivered through the internet that enable the creation of a wide variety of technologies with mass distribution. These services include computing, storage, and databases. What makes the cloud a revolutionary area is the ease and flexibility it offers in managing these resources, as well as the cost efficiency it provides.

There are several significant platforms within this ecosystem, but the one with the largest market share is Amazon Web Services (AWS). AWS offers an extensive range of services that cover all types of solutions in the area, from computing and storage solutions to advanced Machine Learning (ML) services. And the most relevant part is that it offers all this at a significantly reduced cost.

Although the AWS platform offers a vast number of low-cost solutions, it falls short in other areas, such as the complexity of the platform. The AWS platform is not very user-friendly, and some of the reasons for this include the vast number of services it offers and their complexity.

The objective of this Final Course Work (TFC) is, therefore, to present a solution that, in a centralized way, simplifies, in a business context, the way of booking and managing the services that the platform offers. To this, there is also the ambition to reduce complexity to the point that anyone who has a basic knowledge of cloud can manage the platform efficiently.

In order to make this model possible, in the development process a base was created for a management portal that allows new services to be created. The EC2 service of AWS was included, along with documentation, a database and additional user resources, so that it can be expanded and continued in the future.

Contents

Resumo 2

Abstract 3

Contents 4

List of Figures 5

List of Tables 6

1 - Problem Identification 7

2 - Viability and Relevance 10

3 - Benchmarking 13

4 - Engineering 17

5 - Proposed Solution 26

6 - Test and Validation Plan 33

7 - Method and Planning 36

8 - Results 37

9 - Conclusion and Future Work 39

Deployment Roadmap 40

Bibliography 41

Glossary 42

List of Figures

1 AWS logo . . . 7
2 Cloud Platforms Market Share . . . 11
3 Terraform Organization Creation . . . 13
4 Terraform Workspace . . . 14
5 Dashboard Mist . . . 14
6 Ylastic Login Page . . . 15
7 Ylastic Dashboard . . . 15
8 Ansible Compatibility . . . 16
9 Administrator Use Case Diagram . . . 20
10 User Use Case Diagram . . . 21
11 Create EC2 Instance Diagram . . . 22
12 Entity-Relationship Diagram . . . 23
13 Login Page (Light Theme) . . . 24
14 Login Page (Dark Theme) . . . 24
15 Dashboard (Light Theme) . . . 25
16 Dashboard (Dark Theme) . . . 25
17 Application Architecture Diagram (Container Diagram) . . . 27
18 Possible application authentication *flow* . . . 29
19 Gantt Planning . . . 36

List of Tables

1 Application Requirements 18
2 Application System Requirements 32
3 Tests Validation 34
4 Deployment Roadmap 40

1 - Problem Identification

Amazon Web Services (AWS) is a cloud platform that offers a wide variety of services. While this can be seen as a positive thing, it also demonstrates the level of complexity of the platform, which can prove challenging for new users and especially for users with little experience and knowledge in the cloud space.

Figure 1: AWS logo

Below we'll describe some of the main problems with AWS in terms of structure and usability.

1.1 Infrastructure complexity

AWS offers a wide range of services, which makes the platform complex by default.

In addition to the number of options available, each service the platform offers solves a different problem and requires specific, in-depth knowledge of each service, i.e. understanding its features and functionalities as well as best practices in order to be able to use them effectively and safely.

Another aspect that demonstrates the complexity of the platform is the interconnection of services. In other words, the fact that for a user to be able to use one of the services they want, they are often 'forced' to use other services on the platform. For example, a user who wants to *deploy* a web application via *AWS Amplify* and who wants to use a domain they already own needs to use the *AWS Route 53* service in order to be able to use it. This means that the user must not only understand how the *AWS Amplify* service works, but also how the *AWS Route 53* service works so that they can assign the domain to the application.

1.2 Demanding Learning Curve

Although some of the strengths of AWS are the variety of services and multifaceted nature that allows its users to build virtually anything imaginable when it comes to the cloud, the platform is not particularly friendly to new users.

One of the main factors that makes it difficult for new users to familiarize themselves with the platform is not only the number of services available, but also understanding the purpose of the different services and how they work, individually and collectively. One of the challenges a developer encounters when using AWS is figuring out which service is the most appropriate for the problem at hand, given that there are a vast number of services that are closely related or have similar functionalities. All this makes choosing the most efficient service (or combination of services) to solve the problem at hand a challenge in itself.

Finally, we have to talk about the technical (and sometimes complex) nature of the AWS documentation, which although it provides detailed guides for each service, can be intimidating for someone starting out and for users who have little experience and are not yet able to understand such technical language. This documentation also (often) assumes that the user already has a certain level of knowledge, which is not always confirmed.

1.3 Complexity in User and Permission Management

The large number of AWS services means that there are also a huge number of user permissions and an even greater number of possible combinations of these permissions.

Another factor that leads to the complexity of managing permissions is that it is often necessary to have specific knowledge of a particular domain, i.e. a particular service, in order to effectively manage permissions related to it. Sometimes it is crucial for the user to understand the details of each service and how that service can interconnect with others so that they can define strict and secure access controls.

Another aspect that is essential for managing permissions and mitigating security risks is understanding the *Minimum Privilege Principle*, which consists of granting a user only the permissions that are strictly necessary for them to be able to carry out their tasks. This principle may seem simple enough, but in practical terms it's not always easy to implement given the complex nature of the platform's permissions.

1.4 Time-consuming configuration and maintenance

Configuring and maintaining resources on Amazon's cloud is a time-consuming process for a number of reasons. One of the main reasons is the amount of customization that each of the services can perform, allowing virtually every aspect to be customizable.

Configuring and maintaining roles and Identity and Access Management policies is another factor that contributes to resource setup being an extensive process.

1.5 Shortage of Good Security Standards

Although, as a general rule, AWS is a platform with very strong and well-defined security standards, in some situations it lacks good security standards, which can lead to security breaches that affect users.

These problems are related to, and not limited to, complexity in managing permissions and policies (such as too many permissions), weak default settings, and human error by even the most experienced users (caused by a lack of user guidance).

Some of the most common mistakes made when configuring resources, which could have been prevented by good security standards, are creating public buckets in the AWS S3 service (instead of private), creating public databases in the AWS RDS service (when they should be private), volumes in the AWS EBS service are configured as unencrypted by default.

These types of configuration errors happen with some frequency and end up culminating in serious security breaches (especially when targeting large companies), which can even lead to data breaches, downtime and ransomware, which will subsequently result in reputational and monetary damage (e.g. loss of profit, ransomware payments, etc.). Below are some real-life scenarios arising from this type of configuration failure:

- A former Amazon employee was convicted of a data leak related to *Capital One* in 2019 that exposed the personal data of more than 100 million people. The data leak was caused by a fault in the configuration of a bucket of the AWS S3 service. All

this resulted in a fine of 80 million dollars (USD) and lawsuits with customers worth 190 million dollars (USD). [Gov22]

- The company *Capita* suffered a ransomware attack in 2023 that caused an estimated cost of between 15 and 20 million pounds. The attack was caused by a configuration fault in a bucket of the AWS S3 service because it didn't have a password configured. [Scr23]
- The company *Chatrbox* suffered a data leak in 2019 that resulted in the exposure of a database containing the personal data of around 350,000 users of the platform. The leak was due to a configuration failure on one of the AWS servers. [Whi19]

1.6 Unpredictability and Lack of Control over Costs

The unpredictability of costs is a factor to bear in mind as a user of AWS services because it is possible that they can get 'out of control' quite easily, especially if they are not managed efficiently and with some caution. The platform's pricing model is relatively complicated given the number of services and the way they are charged varies from service to service (by computing, by storage, etc.).

One of the essential aspects that AWS does not provide is a feature that allows you to easily limit the cost of your resources (in other words, a *stop loss*) if they exceed a previously defined limit. There are ways to implement this *stop loss* but they are not trivial, as they require a solid understanding of some of the platform's services, such as AWS Budgets (a service that allows you to create budgets), AWS CloudWatch (a service that allows you to monitor resources in real time) and AWS Lambda (a service that allows you to run code based on events). With the services described above, it is possible to create a pipeline that manages resources, namely limiting costs, based on an initial budget.

1.7 UI complexity

Possibly the most important element of Amazon's platform, which enables users to interact with it, is the *User Interface* (UI). The complexity of the UI comes not only from the vast number of services, but also from its technical nature, which can prove challenging for beginners.

In the end, this console is not very *user-friendly* in terms of navigation. It's not difficult, as a new user, to feel overwhelmed by the amount of functionality you have access to and the way the interface is organized may not be intuitive for all users. This can make finding the functionality you want a time-consuming and exhausting process.

1.8 Summary

Given the initial identification of the problem and the final solution, our project was consistent with the established objectives and a collaborative portal was developed with a user-friendly interface to solve the complexity problem encountered by AWS services in their platform.

2 - Viability and Relevance

In this chapter we intend to demonstrate the feasibility and relevance of the application. We will set out the reasons and arguments why we believe this project is viable and relevant to solving the problem identified above.

We believe that our solution is not only viable in terms of solving the problem, but could also have commercial value when applied to an appropriate business model.

2.1 Feasibility and Relevance of the Solution

In the following sub-sections, we will present substantiated reasons for the viability and relevance of our solution.

2.1.1 Increased Complexity in the Cloud

The complexity of the Cloud, more specifically the AWS, is the most obvious reason for using our solution, since one of its main objectives is to simplify resource management in this type of environment.

With the technological advances we are currently seeing, particularly in the area of Artificial Intelligence (AI), the complexity of the cloud environment will continue to increase, leading to the creation of new resources that are highly specialized in computing Machine Learning models (ML).

With all these developments culminating in an increase in the complexity of the space, there will also be greater demand for solutions that offer greater simplicity, efficiency and agility in managing these resources.

2.1.2 Increased Demand for Qualified AWS Professionals

According to the study by *Fortune Business Insights* [Ins23], the cloud market was valued at 569.31 billion dollars (USD) and is expected to reach 677.95 billion by 2023. It is also expected, taking into account the study, that by 2030 the value of this market will reach 2,432.87 billion dollars (USD).

With the rapid growth of the Cloud there will also be a huge increase in demand for professionals with knowledge in this area and it is predicted that there will be a shortage of them. According to a report by *SoftwareOne* [Tea23], when questioning 500 IT decision-makers (IT), 95% believe that their team is negatively impacted by knowledge in the cloud area.

This data shows that teams in IT have a gap in cloud knowledge and this could become even more significant in the future. Our solution can bridge this gap as it undertakes to simplify interaction with the most widely used cloud platform, in this case AWS, and enables users with little knowledge of the area to manage resources in the cloud.

2.1.3 Companies' difficulties in migrating to the Cloud

According to a blog published on the website of *Lucidchart* [Luc23], one of the main challenges companies face when migrating to the cloud is the lack of knowledge their professionals have in the cloud area and/or the lack of resources to find competent professionals with experience in cloud migrations.

As we mentioned earlier, our application would enable the employees of these companies to carry out this migration themselves, since the level of knowledge required is relatively basic.

2.1.4 Cloud and AWS Market Share

In a statistic made by *Statista* [Ric23] it is clear that AWS holds 32% (about 1/3) of the entire cloud market share, above the *Microsoft Azure* and *Google Cloud* platforms. Below is a graph of cloud market shares, represented by the figure 2.

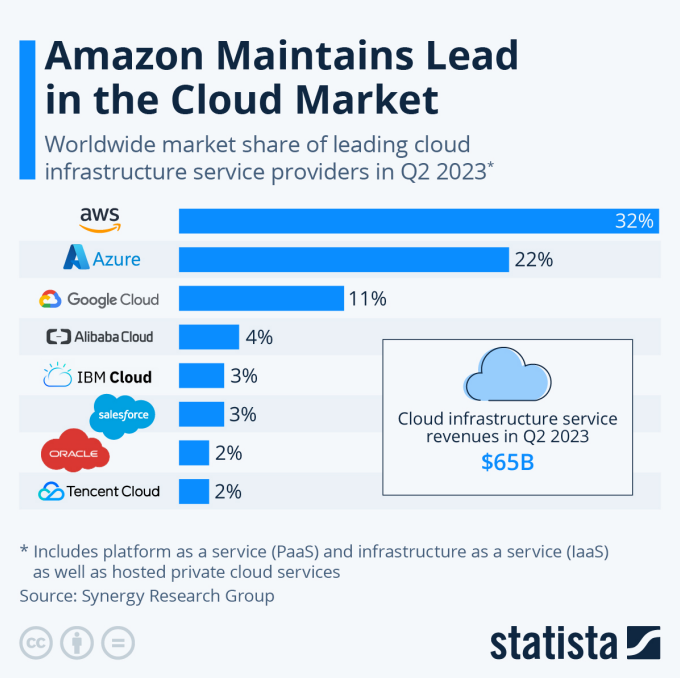


Figure 2: Cloud Platforms Market Share

This data makes our application feasible because, with AWS having such a significant market share, there is a large base of potential users who want a simpler solution.

2.1.5 Existing Third-Party Solutions

The *third-party* tools, such as those mentioned in the *Benchmarking* chapter, are themselves an indication that there is a market for this type of solution. They are all solutions that, in one way or another, expand the functionalities of the AWS platform and, in a way, simplify the resource management process.

2.2 Monetizing the Solution

Not only do we believe that our solution is viable in the sense that it solves the problem identified in chapter 1, but it can also become profitable in economic terms when an appropriate business model is applied.

This solution has the potential to become economically profitable if a Software as a Service (SaaS) model is applied, which would consist of creating a platform that hosts the solution and makes it available to customers on a subscription basis. In this way, all the infrastructure needed to run the application would be managed by the platform, making it possible for various types of clients (private clients and small, medium or large

companies) to access it by paying a monthly subscription, the amount of which would vary depending on the type of client. The cost would be higher for large companies, and lower for private customers or small businesses.

To clarify, the monthly subscription would be per associated user, i.e. the price would be charged according to the number of users using a particular account. A company with 100 users would have to pay for 100 monthly subscriptions (one for each user).

2.3 Summary

The viability and relevance of the topic remains valid and given the complexity of managing the most diverse AWS services, our solution of creating a portal to facilitate these procedures meets the relevance of this development. In this way, important foundations have been built for the portal that will allow the solution to continue post-TFC with the implementation of more services and configurations until a final product is achieved.

3 - Benchmarking

Our private portal AWS based on use in a business context is distinguished by the use of a simple and interactive interface, with the aim of facilitating the management of resources on the AWS platform.

In terms of resource management, our portal seeks to make it easy to manage resources, so that any company with a minimum of knowledge can have its information stored and share it with other users.

3.1 Market solutions

There are currently other platforms on the market which, in this case, take advantage of AWS and provide users and companies with an easier way to manage their resources.

3.1.1 Terraform

Terraform created at 2014, is a platform that allows through the use of DevOps methodologies, the creation and management of resources in the cloud. The main differentiating factor is the use of code, i.e. configuration files to manage the infrastructure.

This type of platform solves the complexity encountered when creating an app in the cloud, which usually requires configuring various options to get it running properly.

Terraform is also free and open source, which significantly reduces costs compared to other similar platforms.

When the user registers and then logs in, they are presented with a simple interface as visible in the figure 3 for creating a new organization.

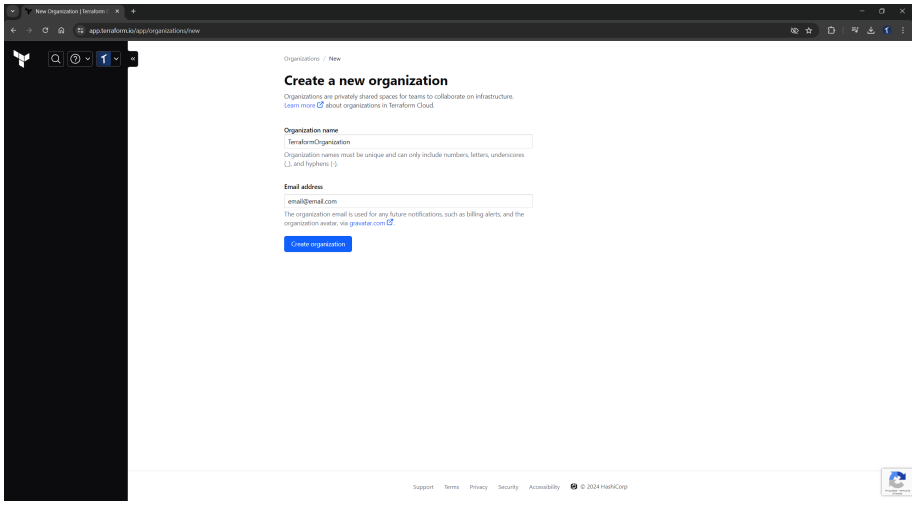


Figure 3: Terraform Organization Creation

Once an organization has been created, this tool allows to access the workspace area and create one according to the needs for the respective purpose. The figure 4 shows the example of such workspace area.

However, Terraform does not offer a dashboard or graphical interface to its users in order to help in the infrasctrute management, which, because of that, makes it more

complex to use. That way, the necessary configurations are made via CLI which requires more knowledge.

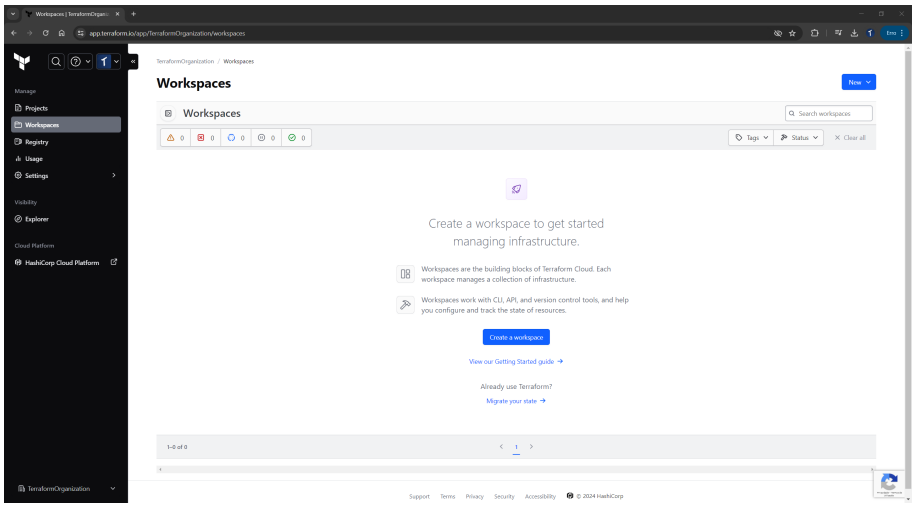


Figure 4: Terraform Workspace

3.1.2 Mist

Another example is the Mist platform, open source and created at 2013, which allows centralized management of cloud resources, offering various customization options, which allows it to meet the expectations of each company.

This platform already offers a dashboard for resource management, which is closer to our portal objective. It is on this basis that it is possible to access the resources and control the portal more easily, as visible in the figure 5.

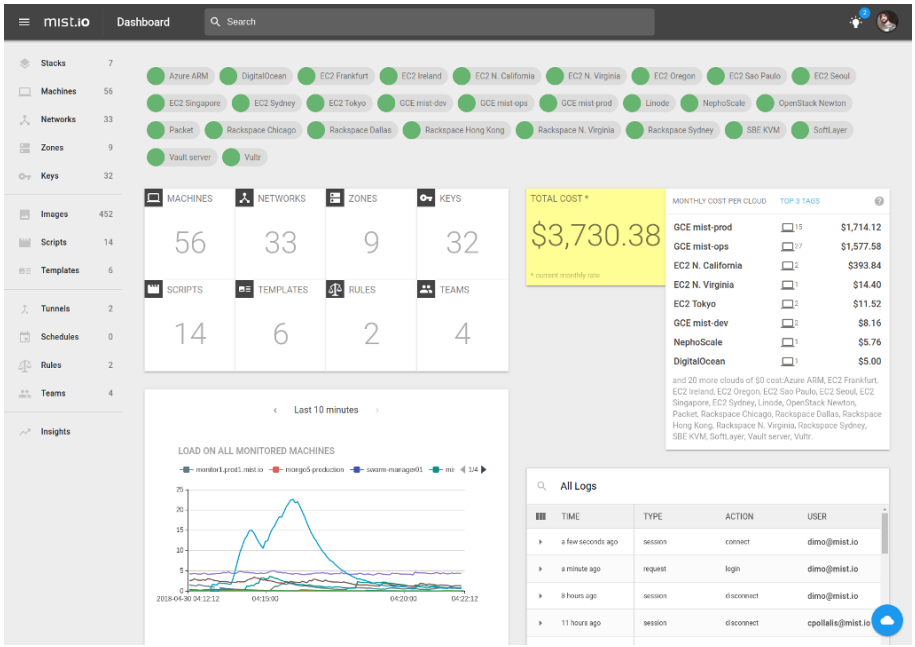


Figure 5: Dashboard Mist

Despite of this, it is a much less well-known platform worldwide than the others available on the market, but it still has a complex dashboard with lots of useful data.

3.1.3 Ylastic

Ylastic is another alternative that uses a CLI to allow the management of resources in a single panel, providing a simplified interface to control and manage the resources on the Amazon cloud.

On its initial login page, it displays few information, being quite simple and similar to our objective, as shown in the figure 6.

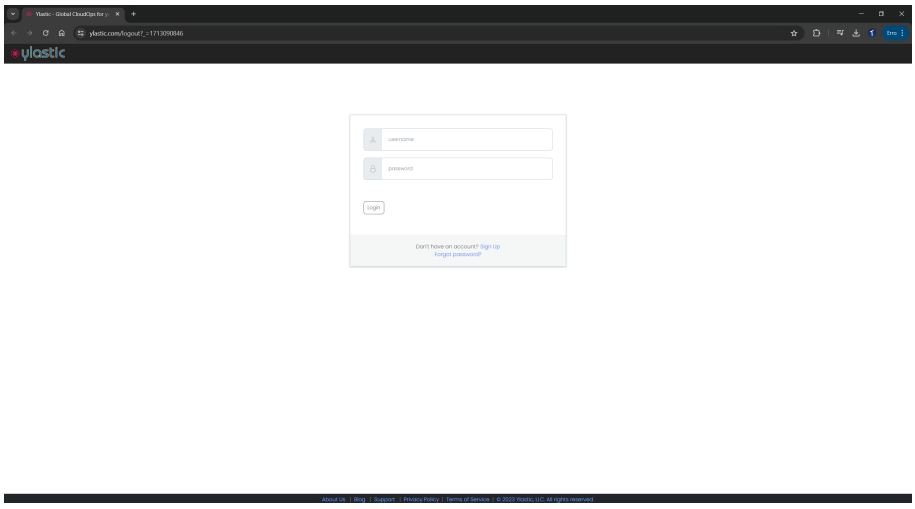


Figure 6: Ylastic Login Page

In its main page, users will have access to manage a range of resources and the information is divided into various sections. It is also possible to access where resources are running, their status and other information, but starts adding complexity given all these options. The figure 7 shows an example of it.

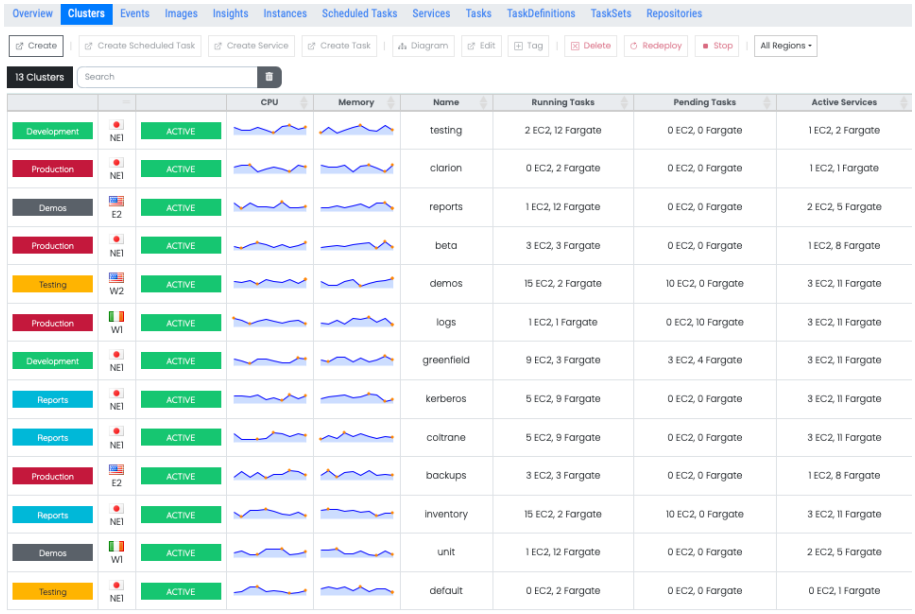


Figure 7: Ylastic Dashboard

3.1.4 Ansible

Ansible is as a powerful open source automation tool which helps to organize, simplify and optimise resource management. Created at 2012 and later acquired by a company called Red Hat, Ansible has a dedicated portal to manage different resources and has an important consideration in improving different infrastructures and automate different tasks.

These positive points include security, multi-platform support and the possibility of generalized resource management in a single tool, which is considered a strong candidate for other solutions on the market due to all the compatibility and variety of adaptations it allows.

For AWS is one of the most complete platforms, but our objective is something simpler and more accessible.

Below in the figure 8 there is in more detail the different possibilities with Ansible.

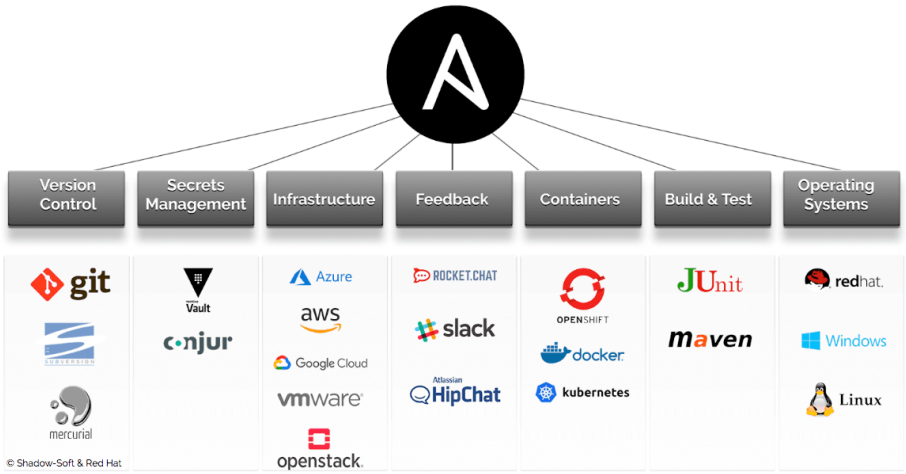


Figure 8: Ansible Compatibility

3.1.5 Other Solutions

There are also many other platforms, such as ”Morpheus”, ”OpenNebula”, ”Chef Automate”, ”ManageIQ”, among others, but many of these do not offer a dashboard for controlling resources, and generally if they include it, they have a complex interface.

Each of these platforms has its own specific characteristics and different ways of working, which leads users to choose one according to their needs.

4 - Engineering

This chapter describes the analysis and gathering of requirements for the development of the project. Section 4.1 describes the functional and non-functional requirements. In section 4.2, the use case diagrams. In section 4.3, the activity diagrams. Section 4.4 presents the entity-relationship model that allows data to be managed in a database. Section 4.5 presents the prototype pages for the AWS resource management portal.

4.1 - Requirements Gathering and Analysis

Identifying the entities for the development of the project was a crucial aspect in order to ensure the correct modelling. As such, the entities identified were the following:

- **Administrator** - Person who manages the application resources
- **User** - Person who uses the platform

Functional and non-functional requirements are differentiated as follows:

- **Functional** - Functionality that a system must have to deliver those results to the user.
- **Non-Functional** - Description of a system's characteristics, such as performance, simplicity of use and security.

The application requirements gathered are listed in the table 1 below.

Table 1: Application Requirements

REQ ID	User Story	Complexity	Importance	Estimated Effort
FR01	As an administrator, I want to be able to create, edit, and delete users in order to manage who has access to the application.	L	Must Have	High
FR02	As an administrator, I want to create, edit, and delete user roles in order to manage access to resources in the application.	L	Must Have	Alto
FR03	As an administrator, I want to manage user permissions to allow platform users to manage resources.	L	Must Have	High
FR04	As an administrator, I want to have access to logs of activities carried out within the platform so that I know what changes have been made, at what time and by whom.	M	Must Have	Low
FR05	As a user, I want to be able to add tags to resources so that I can categorize them and find them more easily.	S	Could Have	Low
FR06	As a user, I want to be able to create workspaces (i.e. groups of resources) to make it easier to manage the resources associated with a particular project.	L	Could Have	High
FR07	As a user, I want to have access to a command palette that allows me to search for features on the platform easily.	L	Should Have	High
FR08	As a user, I want a dashboard showing the resources I've managed recently so that I can manage them more efficiently.	L	Must Have	High
FT09	As a user, I want to be able to customize my dashboard so that I can access the relevant resources more quickly.	L	Could Have	Medium
FR10	As a user, I want to have the services available by category (e.g. computing, containers, databases, storage, etc.) so that it's easier to find the services I want.	S	Should Have	Low
FR11	As a user, I want to create AWS EC2 service resources on the platform so that I can deploy applications.	M	Must Have	Medium

Continued on next page

Table 1: continued from previous page

REQ ID	User Story	Complexity	Importance	Estimated Effort
FR12	As a user, I want to create AWS RDS service resources on the platform so that I can add relational databases to my applications.	M	Must Have	Medium
FR13	As a user, I want to create AWS Lambda service resources on the platform so that I can deploy serverless services.	L	Must Have	High
FR14	As a user, I want to create AWS S3 service resources on the platform so that I can store content from my applications.	S	Must Have	Low
FR15	As a user, I want to create AWS DynamoDB service resources on the platform so that I can create NoSQL databases for my applications.	M	Must Have	Medium
FR16	As a user, I want to be able to perform quick actions on all resources (such as starting, stopping, restarting and viewing logs) with a maximum of 2 clicks for greater efficiency in resource management.	S	Could Have	Low
FR16	As a user, I want to be able to create budgets with an alert system so that I don't go over the limit of what I want to spend.	L	Could Have	High
FR17	As a user, I want to be able to stop resources when I exceed my predefined budget so that I'm not surprised by unexpected expenses.	L	Could Have	High
NFR1	As a user, I want to create resources with a maximum of 3 clicks so that I can create resources quickly.	S	Could Have	Low
NFR2	As a user, I want detailed descriptions of each type of service, to better understand its use.	S	Could Have	Low
NFR3	As a user, I want a good user experience within the application for simplification and efficiency in the resource management process.	M	Must Have	Medium
NFR4	As a user, I want the platform to always opt for secure default settings so that there are no vulnerabilities in my applications.	M	Must Have	Medium
NFR5	As a user, I want the platform to have predefined fields (with default values) so that I can create resources more quickly and efficiently.	S	Must Have	Low

4.2 - Use-Case Diagrams

Use-Case Diagrams allows to show a visual representation of how the system works with the declared entities. This makes it possible to see the relationship between the actors and the system and understand which actions are expected.

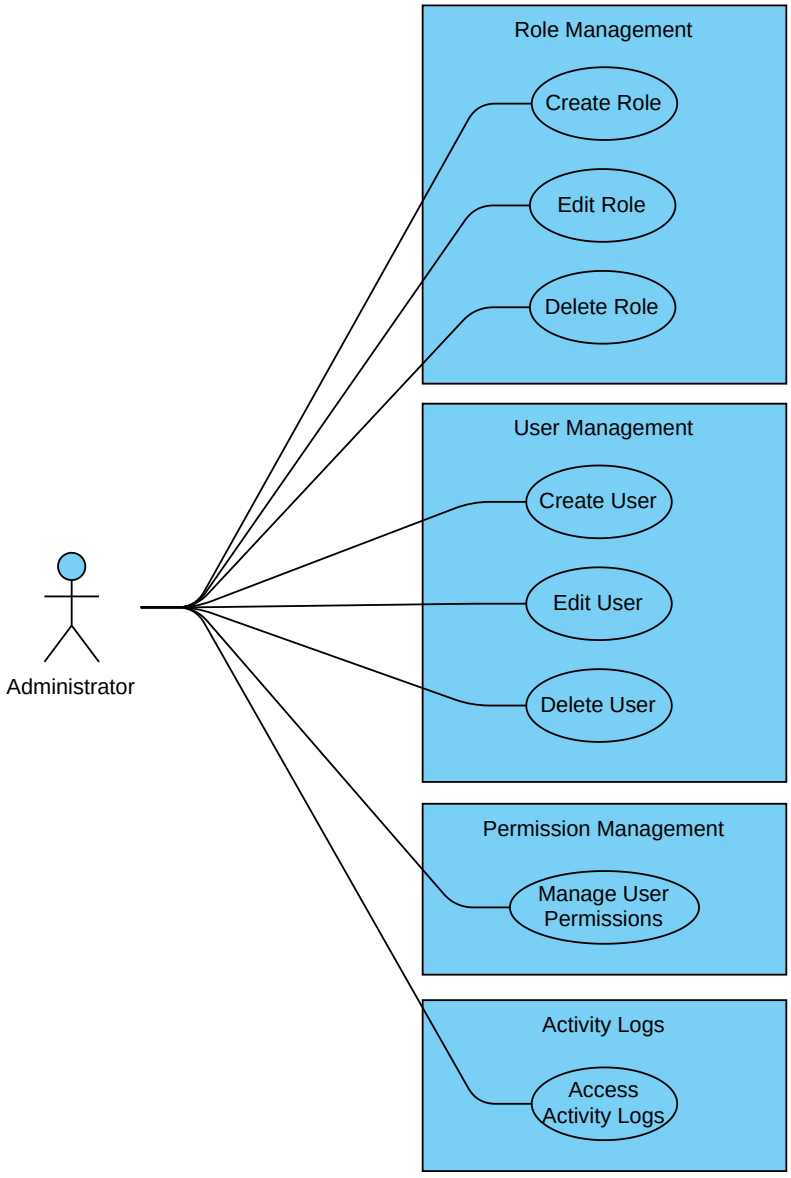


Figure 9: Administrator Use Case Diagram

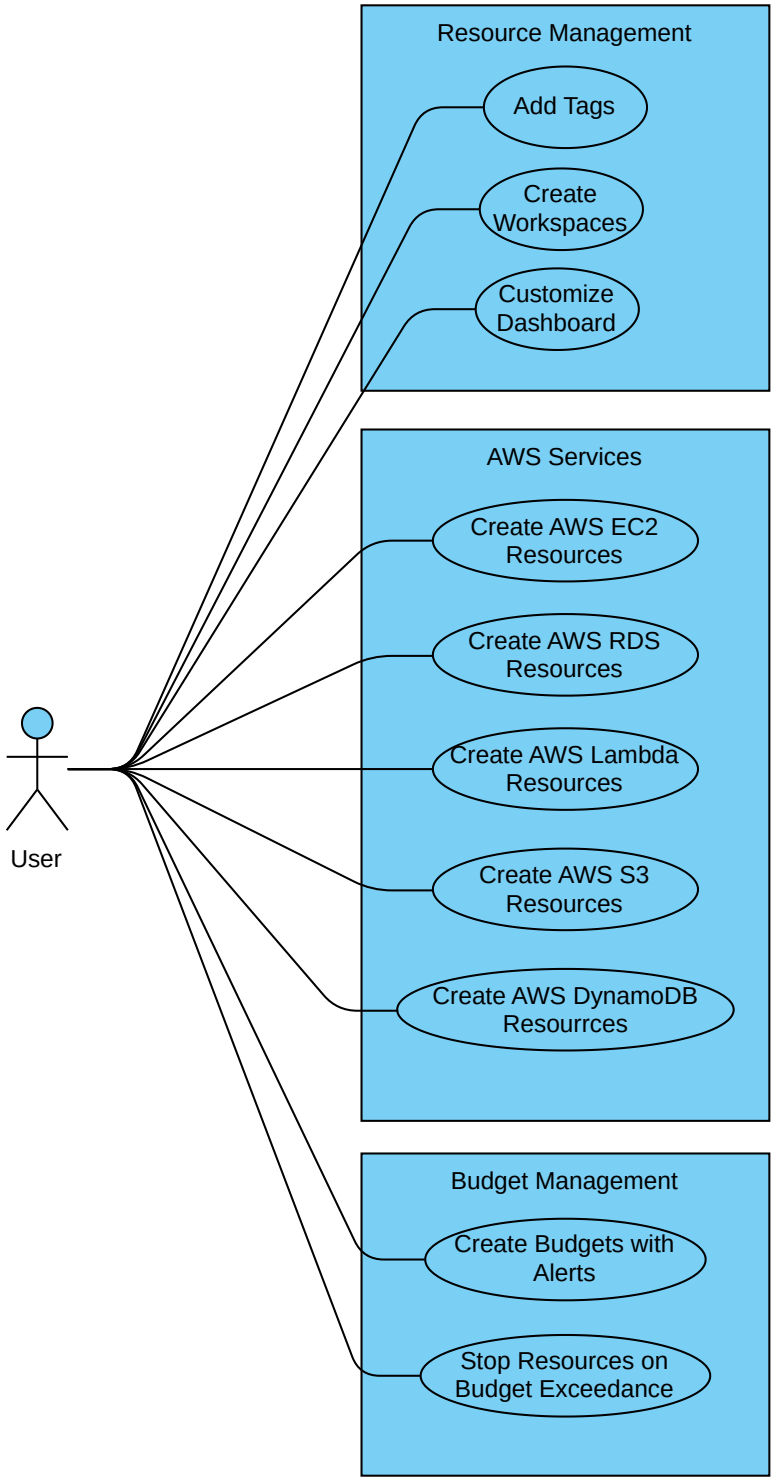


Figure 10: User Use Case Diagram

4.3 - Activity Diagrams

Activity diagrams are important to organize and identify the actions between the users and the application system. Below there are examples for some of the use cases mentioned earlier.

In the figure 11 is shown the diagram for the creation of EC2 service resources by an user.

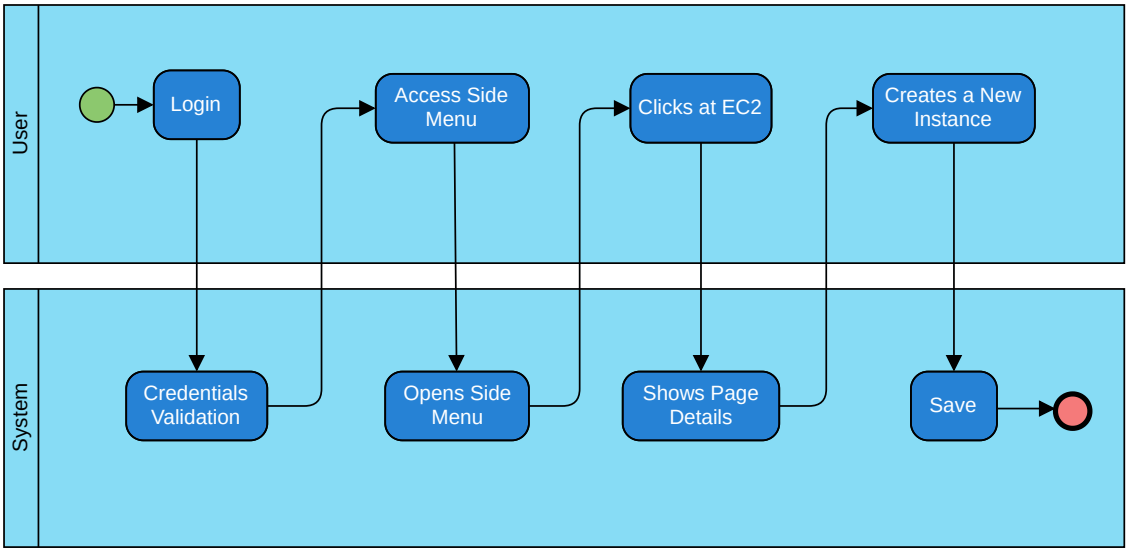


Figure 11: Create EC2 Instance Diagram

4.4 - Relevant Models

Considering the scope and management of the application's resources, the respective data makes it important to create a database for storing and consulting it.

This data is divided into two main groups: user information and the resources created in the application. This allows to access the resources created by a user and keep records of these actions. The Entity-Relationship model in figure 12 shows the relationships between the entities created.

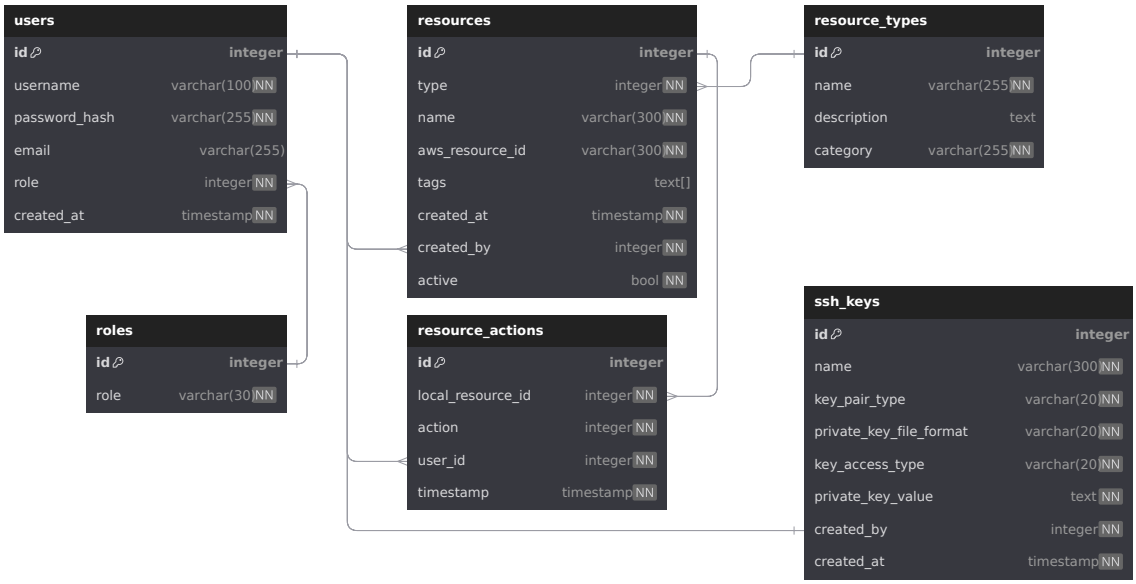


Figure 12: Entity-Relationship Diagram

4.5 - Mockups

This section presents the prototype pages of the AWS management portal. It will be possible to choose the type of background the user want to use, either light or dark. Initially, the portal includes a login page, where users can log in with their details. If they don't have an account, they will have the opportunity to create one. As visible in figure 13 there is the version of light theme for login and in figure 14 the respective login for the dark theme. Regarding the dashboard page, the user can still choose the theme, as presented in figure 15 the light theme and in figure 16 the dark theme.

After accessing the portal, the main page of the portal is displayed, which includes a side menu to facilitate access to the various resources available.

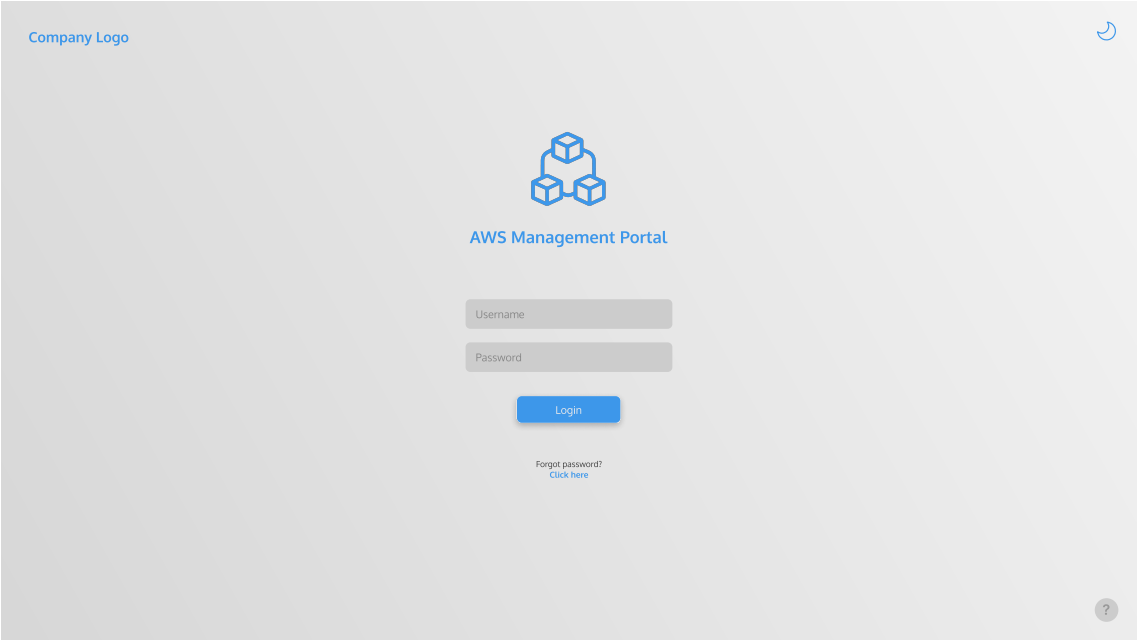


Figure 13: Login Page (Light Theme)

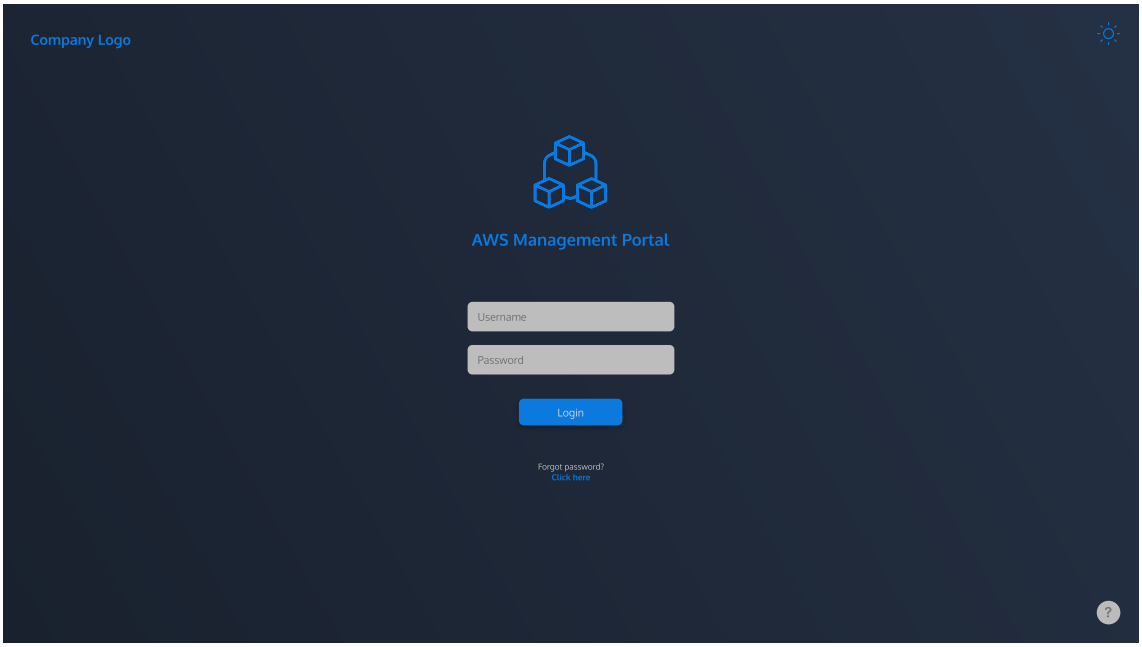


Figure 14: Login Page (Dark Theme)

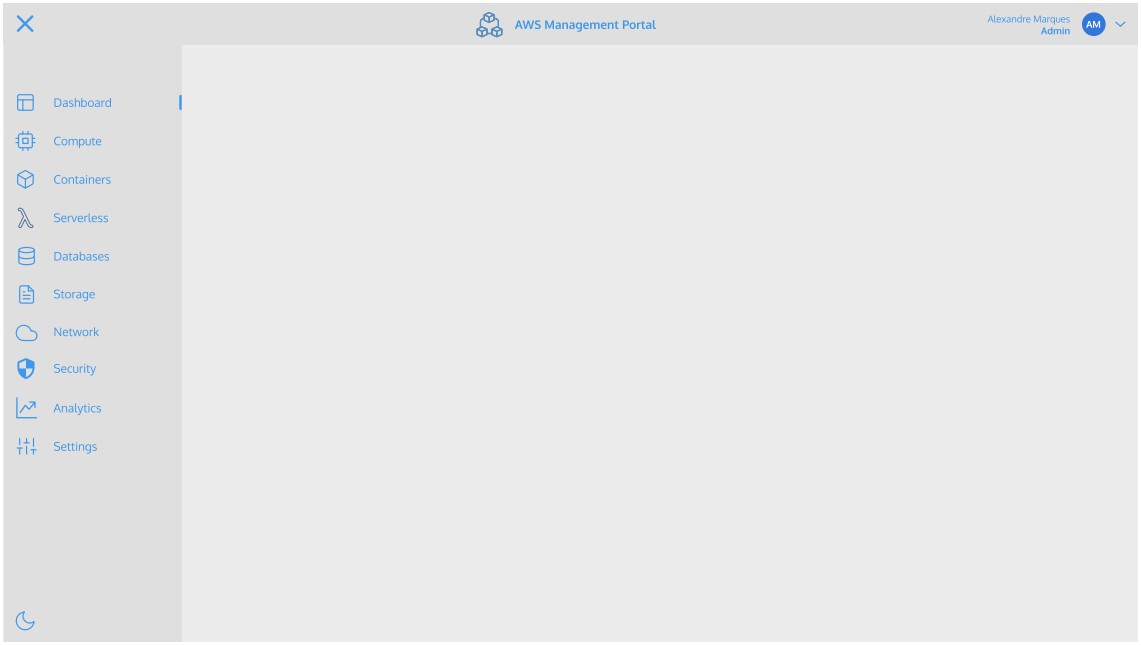


Figure 15: Dashboard (Light Theme)

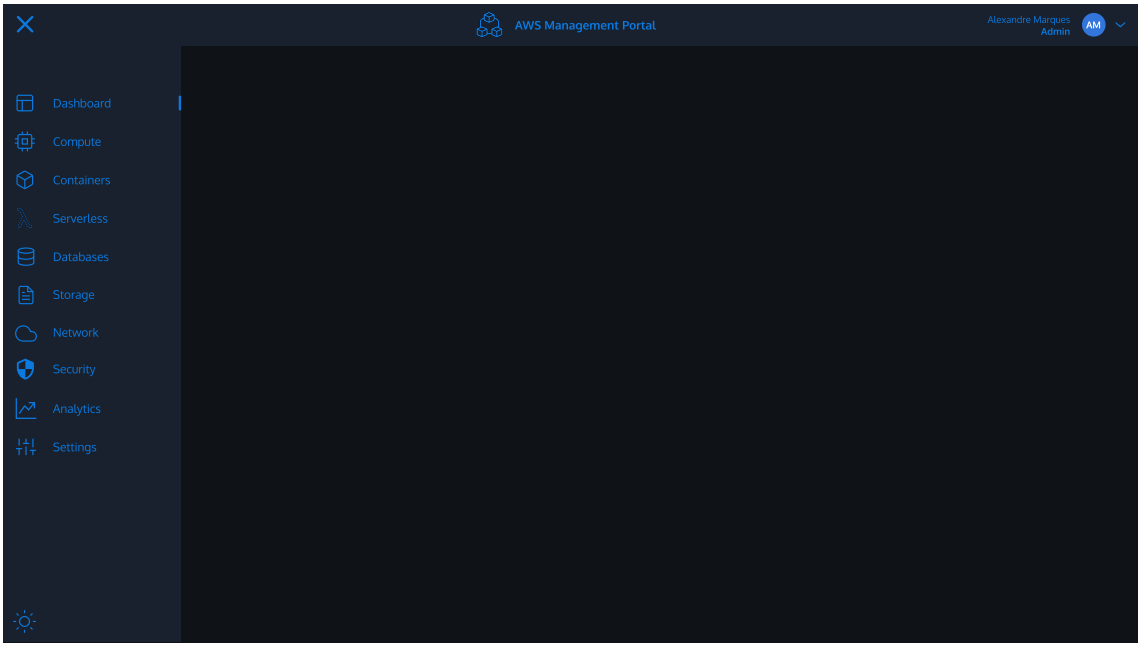


Figure 16: Dashboard (Dark Theme)

5 - Proposed Solution

5.1 - Introduction

Our proposed solution involves developing a private domain collaborative portal that can be used in a business environment. The aim of this portal is to facilitate the management of resources hosted on the AWS platform with a focus on simplicity, efficiency and ease of use.

- GitHub repository: https://github.com/al3x-13/DEISI25-Aws-Management-Portal
- Backend Documentation: https://al3x-13.github.io/DEISI25-Aws-Management-Portal/
- Deployment Guide: https://github.com/al3x-13/DEISI25-Aws-Management-Portal/blob/main/README.md
- Application Demo (video): https://youtu.be/Ve8SiVDAh5s

In the following subsections, we will cover aspects of the application's architecture that will provide a clear view of the application's design and structure. We will present a comprehensive view of its architecture and explain all the components (individually) that make it up and how they will interconnect to form a cohesive system in the end. We will also present all the technologies we will be using and why we chose them.

5.2 Architecture

The structure of the application consists of a microservices architecture based on containers Docker in which each microservice will run in isolation from the others. In this way we are able to encapsulate the application and its execution environment, ensuring that all the services run uniformly and that no matter where we place the application, it will always behave in the same way.

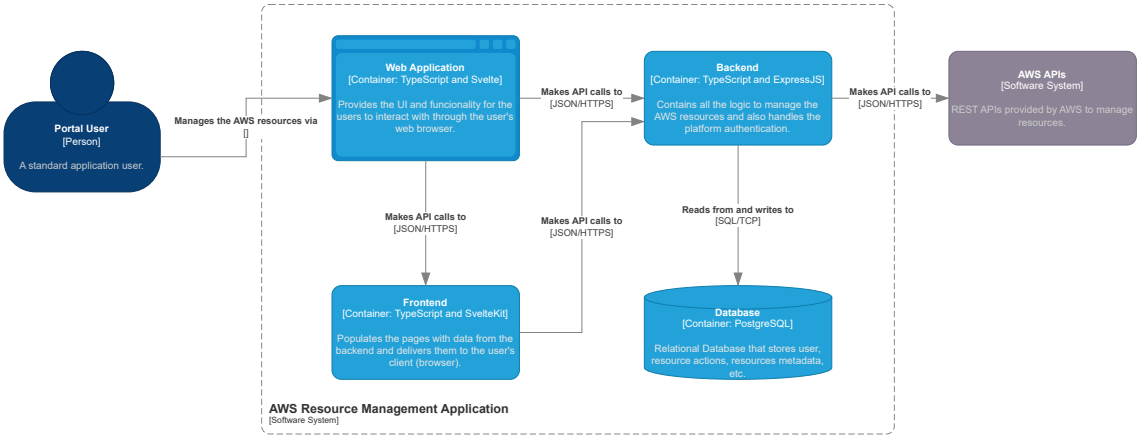
This type of architecture also offers advantages in terms of scalability: if we need to scale any of the components, we can do so easily through horizontal scalability, we just have to start another container.

In addition, due to the nature of the architecture, the application is broken down into several parts with their own responsibility and hosted independently, which means that if part of the application fails, it is less likely to impact the application as a whole, which makes the system more robust.

To manage the code base we will use a monorepo. By using just one repository we can take advantage of several benefits, such as managing dependencies more efficiently, because if we have several services that share dependencies we don't need to install the same dependency several times and manage it in different places.

We will also implement an CI/CD pipeline through Github Actions that will be responsible for the setup, testing and deployment of each containers that will host the microservices. This approach allows us to manage development in a more efficient and automated way.

Below, in figure 17, is a diagram of the application's current architecture.



[Containers] AWS Resource Management Application - System Architecture Diagram

Figure 17: Application Architecture Diagram (Container Diagram)

5.2.1 Backend

The backend of our project will be a REST API that will run on a server supported by ExpressJS, which is a backend framework for web applications based on NodeJS. The backend will be the main ”engine” of the application, since it is where all the logic used to manage the resources in the AWS will be located. To communicate with the AWS platform and all the services associated with it, we will use the SDKs and REST APIs provided by AWS.

From a high-level perspective, this component will be responsible for exposing the APIs that will be used in the frontend.

5.2.2 Frontend

When building the frontend, we decided to use Svelte, which is an innovative framework that makes it possible to create user interfaces with little complexity and a lot of efficiency. Unlike other front-end frameworks such as React, Vue or Angular, Svelte is a much less ”bloated” framework (compared to those mentioned above) and does not use a Virtual DOM (as is the case with React), managing to keep everything that is essential in a front-end framework, which results in lighter, more flexible and better performing applications.

We also used the framework SvelteKit, a tool that allows us to develop more complex applications using Svelte. This library contains various features that make it easier to develop web applications, such as *routing* through the file system, rendering content on the server, generating static content on the server, SPAs, MPAs, generating *boilerplate* code, etc.

5.2.3 Database

As for the SGDB (Database Management System), we had initially planned to use *SQLite* because at the moment we only needed to store data about users. However, we wanted to run the database in a container as an independent microservice, but SQLite is a *serverless* solution, which means it doesn’t need a server to operate because it uses the operating system’s own file system and isn’t suitable for distributed systems, which is the case with the architecture we’re using. SQLite has another disadvantage: it doesn’t support writing to the database of several clients simultaneously.

With this in mind, we decided to use *PostgreSQL* which is an SGDB that operates as a server, unlike SQLite and supports all the situations described above. We decided to use PostgreSQL over other SGDBs because it has some features that the others don't, such as arrays and better management of simultaneous data access.

5.2.4 Communication between Services

When developing the application architecture, we decided to use the REST APIs in JSON format as the main method for communication between the various microservices. The reason for this choice is related to the simplicity and flexibility intrinsic to the *REST* model, which suits the system's architecture.

Given that this communication method is based on the HTTP protocol and that it is a *stateless* model, i.e. the server does not store any state or context of the client's sessions, its implementation is very simple and efficient.

Another advantage of this type of communication is that it can be consumed by any client that implements the HTTP protocol, unlike protocols such as RPC and SOAP, which require specific libraries in order to work.

5.2.5 Authentication

To authenticate users in the application, we decided to use JWTs (JSON Web Tokens) because this method offers advantages that make sense in the context of our application.

The service in our architecture that will manage the authentication part will be backend. At this early stage, we considered that there would be no need to implement authentication directly in frontend since, if we had chosen this path, we would also have had to implement a machine-to-machine authentication system (frontend - backend) that would allow communication between these two components of the system. We therefore decided to manage the users' tokens (JWT) in the frontend and implement authentication only in the backend. The JWTs will be sent in each HTTP request made to the backend in order to validate the user.

This authentication method also allows us to encode information in the token itself, such as user data, permissions, etc. In our application, this feature is advantageous as it allows us to encode each user's permissions in the token. When we send the token to backend, we don't need to query the database to obtain the user's permissions, since the permissions are already encoded in the token.

In the diagram represented by the figure 18, we can see a possible example of this.

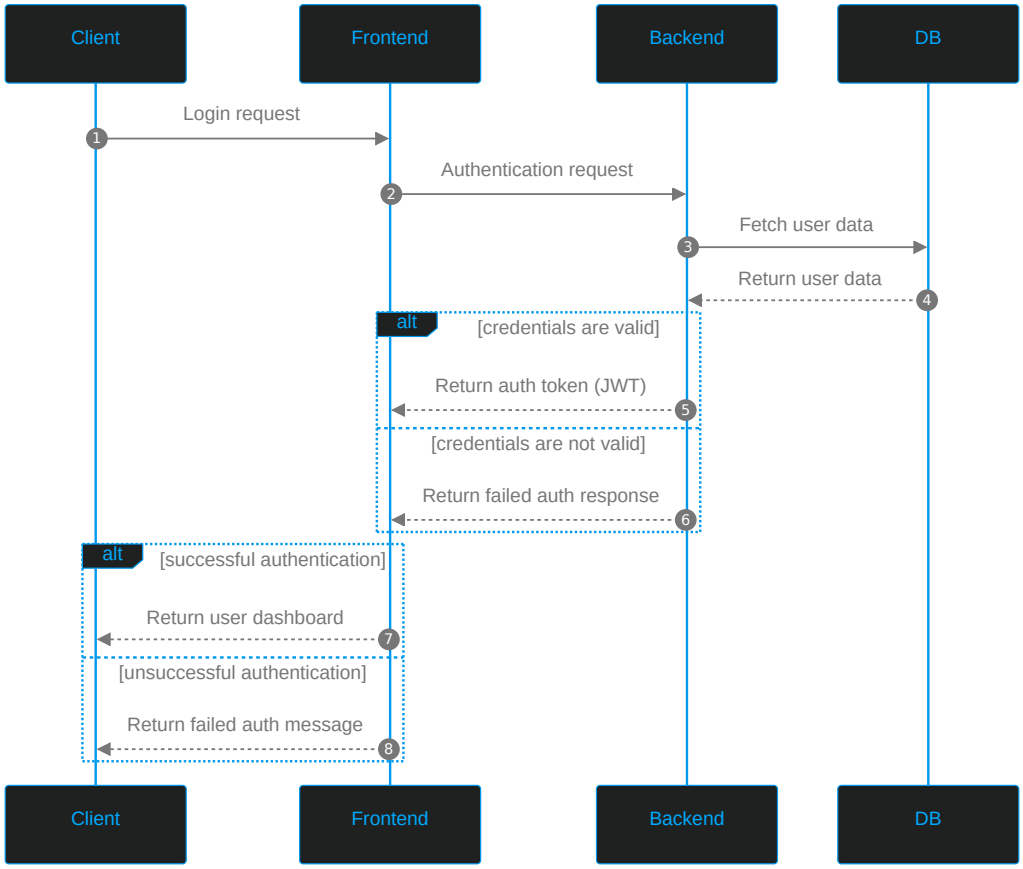


Figure 18: Possible application authentication *flow*

5.2.6 Storing Resources Metadata

We store metadata related to AWS resources to enhance functionality and keep records of resource actions. This includes tracking user actions, resource actions, associating tags, and attributing resource names and other pertinent details. Storing this data allows us to provide a history of modifications, facilitate resource management, and enable customized user interactions.

We do this by mapping AWS resources to entries in the database. Whenever a resource is updated, such as when it is created, modified, or deleted, our application captures these changes and updates our local database accordingly. Simultaneously, we make corresponding requests to AWS to ensure both our local records and the cloud state remain synchronized. This dual-update mechanism ensures data consistency and also provides enhanced resource management.

5.2.7 Keeping Track of User Actions

The platform logs every action performed on resources within the application. Each entry in the log has information about the action taken, the user responsible for the action, and the time of the interaction. By tracking who updated a resource and when we allow administrators of the platform to audit interactions comprehensively, identify unauthorized changes, and verify system integrity.

Each user action on a resource is recorded as an entry in our centralized database. Unlike systems that may store logs in separate files, our integrated approach ensures that logs are easily accessible and can be correlated with other data for more detailed analysis. This method also simplifies backup, recovery, and auditing processes, making them more efficient and less prone to errors.

5.3 Technologies and Tools Used

5.3.1 Programming Languages

The programming language we chose to use for the frontend and backend was TypeScript.

The reason we chose this language is practical: we wanted to use a framework for frontend based on JavaScript since we already have development experience with it and there are several frameworks that would suit our purpose, which is based on a simple, intuitive and efficient graphical interface that users can interact with. That said, we decided to combine the useful with the pleasant and use TypeScript, because not only is it compatible with the JavaScript ecosystem (compiled directly to JS), but it also adds benefits.

There are multiple reasons for choosing TypeScript over JavaScript. The most obvious reason is that it is a language with static types, rather than , which makes it possible to detect errors early in the development process through *type checking* and as it is also a compiled language, it makes it possible to detect a large number of errors during the compilation process and prevent those same errors from occurring during program execution. All this makes the code much more robust and easier to maintain, increasing the efficiency of the development process.

For the backend of the application we decided to opt for TypeScript as well, since it's the language we're using in the frontend and we thought it wouldn't make sense to opt for a different language since it would only increase the complexity of the project and we didn't find any advantage that was significant or relevant enough to use another language.

5.3.2 TS-REST (REST API)

TS-REST is a TypeScript library designed to ensure type safety in REST APIs. It facilitates communication between the backend and frontend and helps reduce runtime errors by ensuring that both ends of our application, backend and frontend, adhere to the same data structures. Type safety helps in catching potential bugs during development before they reach the production build.

5.3.6 Code Management and Dependencies

To manage the various components of the application within monorepo we used PNPM, which is a Package Manager for NodeJS that replaces NPM and not only has better performance but also has support for monorepos out of the box. This tool can also manage dependencies in a single place, so that if the various components of the application share dependencies, they can be managed centrally.

We will also be using Git to perform version control on the project and GitHub to store the application code and facilitate collaboration in development.

5.4 Implementation

The production environment that this project will require is not too complex and it depends on the company/individual that chooses to use the application. A small company probably needs less resources given its size, while a larger company will probably need more resources to scale the application to all its workers.

5.4.1 AWS Account

Given the nature of this application, a requirement that is indispensable is an Amazon AWS account, more precisely the *ROOT* account, which is the account that can generate the access keys that the application requires to function.
Requirements: Amazon AWS account with a valid credit card associated.

5.4.2 Backend

The backend is the component that can vary the most depending on the type of consumer using the application. Although it is the component that varies the most, it still only needs a relatively small amount of resources due to the application tasks being very light when it comes to resource consumption.

A small company might only need a container, so a simple and inexpensive type of compute will do the job. On the contrary, a larger organization might need more compute to host more container due to having more users which leads to more requests being handled at the same time and the compute it will need will depend on the company size.

5.4.3 Frontend

The frontend is the component that serves the User Interface that the user interact with, but other than that it only acts as a middleware between the user and the backend. This means that the frontend should also be able to run with low resources since it does not perform any heavy tasks.

5.4.4 Database

The requirements for the database are very low due to the nature of the application. The only information that will be stored on the database is user data (information about the users and login information) and metadata about the resources hosted on the platform. With just this information on the storage size should be minimal.

5.4.5 Network

In terms of networking there are no must-have requirements. It will depend on the amount of users and the amount of request per unit of time.

Table 2: Application System Requirements

Component	**Minimum Requirements**
Backend Server	1 vCPU, 1 GiB RAM, 10 GiB SSD
Frontend Server	1 vCPU, 1 GiB RAM, 10 GiB SSD
Database Server	1 vCPU, 1 GiB RAM, 5 GiB SSD
Network	10 Mbps

5.5 Scope of Curricular Units

Programming Fundamentals (1st semester) - Basic programming knowledge that is applied throughout the whole project.
Algorithms and Data Structures (2nd semester) - Essential for selecting the right data structures for the APIs as well as the right algorithms to process that data.
Programming Languages 2 (3rd semester) - Provided good principles and fundamentals for organizing and structuring the code both in the frontend and the backend.
Databases (3rd semester) - Essential for the database, providing fundamental knowledge on how to work with relational databases.
Web Programming (4th semester) - Fundamental knowledge used in the frontend, particularly in the structure of the web pages (HTML and CSS), and also on the backend, for creating JSON based APIs.
Computer Networks (4th semester) - Crucial for connecting the microservices (in the containers) and managing the connections between services (web protocols for exchanging information, etc).
Distribuited Systems (5th semester) - Essential when it comes to communication between services, for example, choosing the best technologies and protocols for it, as well as, valuables skills at configuring these types of services.

6 - Test and validation Plan

In order to validate the proposed solution, this chapter presents the test plan for the requirements developed in Engineering chapter, containing a description of the problem and its expected outcome.

The following tests will be performed as visible in table 3:

Table 3: Tests Validation

Title	Description	REQ ID	Expected Output
Manage Users	The administrator create, edit or delete users.	FR01	The administrator is able to create, edit or delete users by selecting one of those actions.
Manage Roles	The administrator create, edit or delete roles.	FR02	The administrator is able to create, edit or delete roles by selecting one of those actions.
Manage User Permissions	The administrator manage user permissions.	FR03	The administrator edit an user permission and permissions are applied.
Get Logs Access	The administrator get access to the activity logs.	FR04	The administrator by clicking on the logs button, receives all the activity logs of the platform.
Add Tags to Resources	The user add tags to resources.	FR05	The user is able to add tags to resources and see them categorized.
Create Workspaces	The user create a workspace.	FR06	The user clicks on the button to create a workspace and it gets created.
Access to Command Palette	The user read command palette.	FR07	The user by clicking on the button to read command palette, gets access to a group of helpers.
Dashboard Resources	The user visualize the dashboard resources.	FR08	The user by accessing the dashboard should have information of their managed resources.
Customize Dashboard	The user customize the dashboard.	FR09	The user by clicking on the settings icon should be able to customize the dashboard.
Services Categories	The user see diferent categories.	FR10	The user when accessing the drawer has the content separated in different categories.
AWS EC2	The user create EC2 service.	FR11	The user in EC2 page should be able to insert all the data and create succefully the resource.
AWS RDS	The user create RDS service.	FR12	The user in RDS page should be able to insert all the data and create succefully the resource.

Continued on next page

Table 3: continued from previous page

Title	Description	REQ ID	Expected Output
AWS Lambda	The user create Lambda service.	FR13	The user in Lambda page should be able to insert all the data and create succefully the resource.
AWS S3	The user create S3 service.	FR14	The user in S3 page should be able to insert all the data and create succefully the resource.
AWS DynamoDB	The user create DynamoDB service.	FR15	The user in DynamoDB page should be able to insert all the data and create succefully the resource.
Create Budgets	The user create a budget.	FR16	The user defines a budget which should alert him when getting closer.
Stop Resources	Resource stop according to budget.	FR17	The user after defining a budget, the resource should automatically stop if it exceeds their budget.

7 - Method and Planning

This chapter is dedicated to the planning of the solution's development process. In the following sections will cover requirements and also list the application's deliverables.

During the development of our TFC, some of the services initially planned were altered due to the complexity of implementing each one on the platform. In this respect, we prefer to move progress with an approach focused on developing a stable base that will later make it easier to continue working on this TFC and implement more services on the platform, which is why only the EC2 service and auxiliary management tools were developed.

Nevertheless, planning remained on schedule and the deadlines for analyzing, identifying and implementing requirements were met, in relation to the timetable proposed in previous reports.

7.1 Calendar

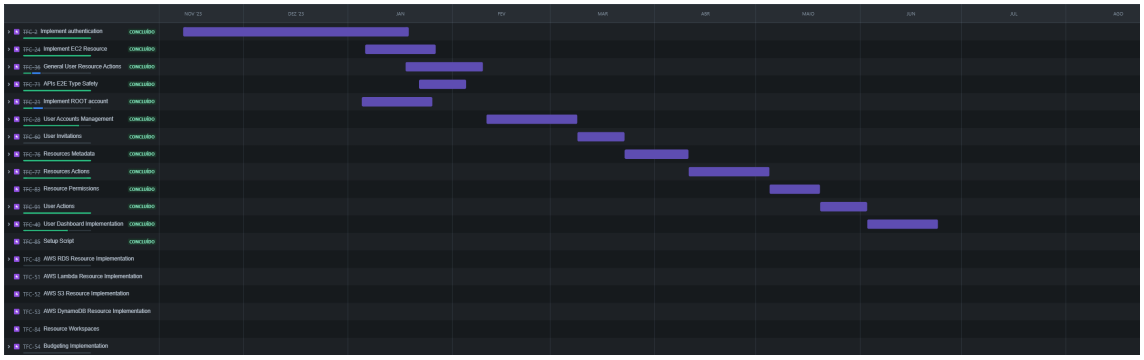


Figure 19: Gantt Planning

7.2 Main Deliverables

The main deliverables of the application are:

- Application with all the associated components (frontend, backend, database, containers, etc.);
- Backend APIs' documentation;
- Instruction manual with steps for configuring and deploying the application;
- User manual with description and details of the UI.

8 - Results

This section presents a comprehensive overview of the final results achieved by our TFC solution.

8.1 Portal Deployment

For the end user to use the collaborative AWS portal developed, the project is stored in Docker containers that allow a simplified and consistent configuration of the production environment. In this way, all the necessary dependencies are guaranteed and the portal works in the same way on different machines and environments, without the need to download the application code and manually run the application on different consoles.

For the installation to be successfully deployed a documented installation guide has been prepared and it includes all the steps and an installer for configuring and downloading the application's environment settings, with all the associated steps are documented in the project's GitHub repository. Given that, the end user by creating a docker-compose file with the settings provided can get the project up and running in their local machine to access the portal.

8.2 Login Page

Once this is done, when the user opens the application in their localhost, they will see the login page designed to start the session. By default there is a root account which has been implemented so that managing the entire application and give permissions to various users or associate someone as an administrator be something possible to happen.

In this context of the login page and other pages of the portal, a feature has also been added to switch between the dark or light mode of the implementation at the user's choice.

8.3 Dashboard

Regarding the post-login part, has been developed a dashboard which, for accounts with permissions, will display a list of existing users and pending user requests. We believe that this information on registered users is important in order to have information on who has access to the platform in the business context and the pending user requests, which makes it easier to analyze them quickly.

Moving forward, the top right of the portal shows the user who is logged in, their roles and is also the place to log out that user.

On the left-hand side of the application, a side menu has been developed where some of the functionalities developed can be found and prepared for the future implementation of different AWS services in the future continuation of this TFC.

At this section is where we currently have the AWS EC2 access panel. Amazon Elastic Compute Cloud is a central part of Amazon.com's cloud computing platform, Amazon Web Services. EC2 allows users to rent virtual computers on which to run their own applications.

8.4 EC2 Service

By accessing EC2, we can see the various configuration options that have been created and which, through the user's AWS account, allow them to use this service and take advantage of it. Our portal works as a tool which, by connecting to the AWS APIs, can communicate and execute external requests to the AWS services, but by taking advantage of this, provide a simple and intuitive portal which allows these services to be used easily

8.5 Backend API Documentation

Given the importance of AWS services in this work, we have also developed Appliaction Programming Interface (API) documentation for the backend used, which includes the services and their respective endpoints. This documentation includes examples and what each endpoint returns when called, whether it's a GET to get information from the server or a POST to send information, and what type of message is returned. This documentation is extremely important for developers who want to use the API and know what it contains. In the context of continuing post-TFC work, this backend documentation facilitates and speeds up the development process for implementing new services.

8.6 Code Layout

For the continuity of work post-TFC, we have left the code easily organized so that the new services can be implemented, since it was essential to build a stable base first for the creation and organization of the portal. This way, depending on the needs, the options can be chosen by those who will be using this solution.

9 - Conclusion and Future Work

9.1 Conclusion

With the development of the AWS Resource Management: A Private Collaborative Management Portal project, we were able to clearly identify the objective of this work, its importance and carry out research into the various AWS services and options available to us. The AWS platform, which has the largest market share, also has many specific services for various needs and we aimed to build a portal that easily allows new services to be added, which was the case of EC2 and its respective documentation, while allowing the continuation of this work for implementation of other AWS services.

Given that, the objective of which is to facilitate user interaction with the most diverse configurations and options of AWS services was met and we were able to develop a solution that can continue to be developed until a final product by focusing on structuring a base for the portal.

9.2 Post-TFC Solution

Given the type of solution we intend to develop, it is not difficult to come up with reasons and characteristics that demonstrate the continuity of this solution in a post-TFC phase.

The most obvious reason is that it won't be possible to immediately implement all the services that the AWS platform offers, given the huge range on offer. In view of this, the implementation of other services would be something that could be done after the conclusion of the TFC.

Another reason would be the complexity that the solution requires. In our case, it would be a very customizable interface that preserves the desired simplicity, which could take a long time to develop.

It would also be possible to implement more features later, such as optimized cost control that would be able to predict the costs that a given project would have based on the resources needed, and the use of Artificial Intelligence (AI) tools could be taken into account.

Attachments

Below in the table 4 it is available a deployment roadmap, containing all the important steps to install the application succefully.

Table 4: Deployment Roadmap

Step	Stage	Task
1	Setup	AWS Root Access Key creation
2	Setup	docker-compose.yml file creation
3	Setup	Setup script download
4	Setup	Setup script execution
5	Access	Login to GHCR from Docker
6	Run	Docker Compose Run
7	Run	Accessing application locally

Bibliography

[Gov22] Justice Gov. *Former Seattle tech worker convicted of wire fraud and computer intrusions*. June 2022. URL: https://www.justice.gov/usao-wdwa/pr/former-seattle-tech-worker-convicted-wire-fraud-and-computer-intrusions (visited on 11/18/2023).

[Scr23] Alex Scroxton. *Black Basta ransomware attack to cost Capita over £15m*. May 2023. URL: https://www.computerweekly.com/news/366537238/Black-Basta-ransomware-attack-to-cost-Capita-over-15m?source=post_page-----3518245b6fab-------------------------------- (visited on 11/18/2023).

[Whi19] Zack Whittaker. *Millions of Instagram influencers had their contact data scraped and exposed*. May 2019. URL: https://techcrunch.com/2019/05/20/instagram-influencer-celebrity-accounts-scraped/ (visited on 11/18/2023).

[Ins23] Fortune Business Insights. *Cloud Computing Market Size, Share & COVID-19 Impact Analysis, By Type (Public Cloud, Private Cloud, and Hybrid Cloud), By Service (Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS)), By Industry (BFSI, IT and Telecommunications, Government, Consumer Goods and Retail, Healthcare, Manufacturing, and Others), and Regional Forecast, 2023-2030*. May 2023. URL: https://www.fortunebusinessinsights.com/cloud-computing-market-102697 (visited on 11/18/2023).

[Tea23] Blog Editorial Team. *The cloud skills gap in 2023 – our new research and report*. Nov. 2023. URL: https://www.softwareone.com/en-us/blog/articles/2023/11/13/cloud-skills-gap-in-2023 (visited on 11/18/2023).

[Luc23] Lucidchart. *Overcoming the most common cloud migration challenges*. 2023. URL: https://www.lucidchart.com/blog/challenges-with-moving-to-the-cloud (visited on 11/18/2023).

[Ric23] Felix Richter. *Amazon Maintains Lead in the Cloud Market*. Aug. 2023. URL: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/ (visited on 11/18/2023).

Glossary

AI Artificial Intelligence. 10, 39

Amazon Web Services Amazon Web Services is a cloud-based computing platform owned by Amazon that offers a wide range of services and is based on a pay-as-you-go model.. 7

API Appliaction Programming Interface. 27, 38

AWS Amazon Web Services. 2, 3, 7–13, 16, 17, 23, 26, 27, 30, 31, 37–39

AWS Amplify AWS service that gives developers the possibility to build easily scalable full-stack applications based on the cloud, including frontend and backend. 7

AWS Budgets AWS service that allows the creation of personalised budgets to monitor the costs of the various services within the platform and allows alerts to be created for them. 9

AWS CloudWatch AWS service for monitoring and managing platform resources in real time. 9

AWS EBS AWS service that offers high-performance block storage to be used in combination with the EC2 service. 8

AWS Lambda AWS service based on server less computing that allows you to create functions that execute code in response to events without the user having to manage servers or infrastructure. 9

AWS RDS AWS service that simplifies the use, management and scalability of relational databases in the cloud. 8

AWS Route 53 AWS service offering Domain Name System (DNS) management including domain registration, DNS routing and status monitoring services. 7

AWS S3 AWS service that offers scalable object storage via a web interface. 8, 9

backend All the logic, processing and data storage that takes place on the server side. 27, 28, 30, 31, 36

bucket Container for storing objects in the AWS S3 service. 8, 9

CI/CD Continuous Integration and Continuous Delivery. 26

CLI Command line interface. 14, 15

cloud Computing model that allows computing, data storage, network management and access to applications on remote servers via the Internet. 2, 3, 7, 8, 10, 13–15

container Lightweight, self-contained and executable package (i.e. container) that includes everything needed to run a particular piece of software, including code, dependencies, libraries, execution environment and configurations.. 26, 27, 36

dashboard Dashboard is an interactive graphic interface that allows you to analyse data. 13, 14, 16

DevOps Combination of development and operations to increase the efficiency, speed and security of software development. 13

Docker Open-source platform that allows virtualisation at operating system level and enables the creation, execution and delivery of applications in isolated, lightweight and portable environments in container format.. 26, 37

EC2 Amazon Elastic Compute Cloud (Amazon EC2) provides on-demand, scalable computing capacity in the Amazon Web Services (AWS) Cloud. 2, 3, 36–39

ExpressJS Backend framework for web applications based on NodeJS with support for API development (e.g. REST APIs). 27

framework A set of libraries that provide generic functionalities and impose a development model and a set of rules on which applications can be developed.. 27, 30

frontend All client-side components (in this case the browser) that the user interacts with (e.g. graphical interface). 27, 28, 30, 31, 36

GET HyperText Transfer Protocol (HTTP) method that is applied while requesting information from a particular source. 38

Github Actions CI/CD (continuous integration and continuous delivery) platform that allows automation of pipelines for build, testing and deployment using YAML. 26

HTTP Hypertext Transfer Protocol. 28

IT Information Technology. 10

JavaScript High-level programming language with dynamic types interpreted or compiled using JIT (Just-In-Time) compilation. Widely used for web development. 30

JS JavaScript. 30

JSON JavaScript Object Notation. 28

JWT JSON Web Token. 28

microservices Software architecture in which an application is made up of several independent and self-sufficient components that communicate using well-defined APIs. 26, 28

ML Machine Learning. 2, 3, 10

monorepo Development strategy in which the entire code base is stored in a single repository. 26, 31

MPA Multiple Page Application. 27

NodeJS Open-source, cross-platform, single-threaded JavaScript execution environment for building performant and scalable server-side applications. 27, 31

NPM Node package manager. 31

open source Code that is accessible to everyone. Not private. 13, 14, 16

Package Manager Tool that automates the installation and management of system dependencies. 31

PNPM Performant npm. 31

POST HyperText Transfer Protocol (HTTP) method that is designed to send loads of data to a server from a specified resource. 38

PostgreSQL An object-relational database management system. 28

ransomware Type of malware that encrypts all the data on a system, blocking user access to it until a ransom is paid. 8, 9

REST Representational State Transfer. 28

REST API Interface based on the REST (Representational State Transfer) architecture style for APIs that defines a set of rules that dictate how interaction with a web service should be carried out.. 27, 28

RPC Remote Procedure Call. 28

SaaS Software as a Service. 11

SDK Software Development Kit. 27

serverless System that doesn't need a server to operate. 27

SGDB Sistema de Gestão de Base de Dados. 27, 28

SOAP Simple Object Access Protocol. 28

SPA Single Page Application. 27

stateless A feature of REST APIs that allows each request to be made independently, i.e. the server doesn't store any information about the requests. Each time a request is made to the server, the request must contain all the information needed to process it. 28

Svelte Modern, simple and efficient component-based JavaScript framework for developing web applications. 27

SvelteKit JavaScript framework used to develop efficient web applications using Svelte. This library offers functionalities such as rendering on the server, generating static pages on the server and routing between pages. 27

TypeScript Compiled programming language that adds static types and object-orientated programming concepts to the JavaScript language.. 30, 31

UI User Interface. 9, 36

Virtual DOM A memory representation of the DOM structure of a web page. Frontend frameworks such as React use this method to realise what has changed on the page, calculate the differences in the virtual DOM and then apply the changes to the real DOM. This method consumes a lot of memory and is not very efficient. 27