LIÈGE université Library

https://lib.uliege.be

matheO

https://matheo.uliege.be

Master thesis : Observability and Visibility in the Cloud

Auteur : Sebati, Ilias

Promoteur(s) : Donnet, Benoît; 19444

Faculté : Faculté des Sciences appliquées

Diplôme : Master en sciences informatiques, à finalité spécialisée en "computer systems security"

Année académique : 2022-2023

URI/URL : http://hdl.handle.net/2268.2/17656

Avertissement à l'attention des usagers :

Tous les documents placés en accès ouvert sur le site le site MatheO sont protégés par le droit d'auteur. Conformément aux principes énoncés par la "Budapest Open Access Initiative"(BOAI, 2002), l'utilisateur du site peut lire, télécharger, copier, transmettre, imprimer, chercher ou faire un lien vers le texte intégral de ces documents, les disséquer pour les indexer, s'en servir de données pour un logiciel, ou s'en servir à toute autre fin légale (ou prévue par la réglementation relative au droit d'auteur). Toute utilisation du document à des fins commerciales est strictement interdite.

Par ailleurs, l'utilisateur s'engage à respecter les droits moraux de l'auteur, principalement le droit à l'intégrité de l'oeuvre et le droit de paternité et ce dans toute utilisation que l'utilisateur entreprend. Ainsi, à titre d'exemple, lorsqu'il reproduira un document par extrait ou dans son intégralité, l'utilisateur citera de manière complète les sources telles que mentionnées ci-dessus. Toute utilisation non explicitement autorisée ci-avant (telle que par exemple, la modification du document ou son résumé) nécessite l'autorisation préalable et expresse des auteurs ou de leurs ayants droit.

Observability and Visibility in the Cloud

Sebati Ilias

Academic Supervisor: Prof. Donnet Benoit

Industrial Supervisor: Christopher Paggen

University of Liege

A thesis presented for the degree of

M.Sc. in Computer Science

2022-2023

Copyright © 2023 by Sebati Ilias
All Rights Reserved

Acknowledgements

I am deeply grateful to **Allah**, the Most Merciful and All-Knowing, for granting me the ability to complete this thesis and successfully conclude my studies. I am forever thankful for His boundless mercy that has encompassed every aspect of my accomplishments.

I would also like to express my heartfelt gratitude to my mother and father for their unwavering support and for making my journey through studies much easier. Their constant encouragement and guidance have been invaluable to me. I am also grateful to my family for their continuous support and belief in my abilities.

I extend my thanks to my academic supervisor, Pr. Donnet Benoit, for his guidance throughout this thesis. Additionally, I would like to express my gratitude to all the professors and assistants at the University of Liege who have provided me with the opportunity to receive this amazing education. Their guidance and expertise have been invaluable and have greatly contributed to the successful completion of this thesis.

Furthermore, I would like to extend my appreciation to my industrial supervisor, Christopher Paggen, for his invaluable advice, unwavering support, and engaging discussions. I am grateful for the opportunity to work under his guidance. I would also like to express my gratitude to all the members of Cisco who have been part of my brainstorming journey.

Lastly, I would like to thank my friends who have accompanied me on this journey or have provided unwavering support. Their presence, encouragement, and camaraderie have made this experience all the more meaningful and memorable.

Abstract

This thesis focuses on the development of an application that aims to achieve visibility and observability in the cloud. It delves into the selection process of various technologies, such as programming languages, libraries, and security systems. Furthermore, the thesis emphasizes the utilization of a microservice architecture for the application. Additionally, it provides an in-depth explanation of the algorithm implemented for performing path reachability between services.

Keywords— Observability - Visibility - Cloud - Software Engineering

Table of Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Aim and Objectives . . . 3

2 Selecting Technologies **4**

2.1 From Monolithic to Microservices Architecture . . . 4

2.1.1 Traditional Monolithic Architecture . . . 5

2.1.2 Microservices Architecture: Advantages and Adoption . . . 5

2.1.3 Kubernetes and Microservices Deployment . . . 5

2.2 Language Choices . . . 6

2.2.1 Web Application . . . 7

2.2.1.1 JavaScript Frameworks . . . 7

2.2.2 Backend language . . . 9

2.2.2.1 Choosing a language . . . 9

2.2.2.2 Language comparison . . . 10

2.2.2.3 Speed comparison . . . 11

2.2.2.4 Number of Lines Comparison . . . 12

2.2.2.5 Conclusion . . . 13

2.3 Authentication . . . 13

2.3.1 Session Tokens . . . 14

2.3.2 JWT Tokens . . . 15

2.4 Infrastructure . . . 17

2.4.1 Our Test Infrastructure . . . 18

2.5 Swagger Documentation . . . 19

3 An Exploration of the Methodology: Components and Infrastructure **21**

3.1 An In-depth Look at the Frontend . . . 22

3.1.1 Cisco UI Kits . . . 22

3.1.2 Material UI . . . 23

3.1.3 Navigating with React Router Dom . . . 24

3.1.4 State management . . . 24

3.1.5 Using Object-Oriented Principles . . . 26

3.1.5.1 Showing Detailed Information . . . 26

3.1.5.2 Model . . . 27

3.1.5.3 Server Communications . . . 32

3.2 Understanding the Login Service . . . 33

3.2.1 Big Picture . . . 34

3.2.2 Database . . . 35

3.2.3 User Authentication 36
3.2.4 Detailed Documentation 38
3.2.5 Tests 41
3.3 Unravelling the Main Gateway 42
3.3.1 Authentication Via Cookie Verification 43
3.3.2 Extracting Relevant Information to Handle the Request . . 45
3.3.3 Fetching and Returning the Data 46
3.3.4 Testing Procedures 48
3.3.5 Throttling of APIs by CSPs 49
3.4 Exploring the Reachability Server 49
3.4.1 Service Technology Considerations 50
3.4.2 High level view of the Algorithm 51
3.4.3 Determining the Next Hop 52
3.4.4 Retrieving the Service Using Its Identifier 56
3.4.5 Returned Values and Their Applications 59
3.4.6 Testing 60

4 Conclusion and Future Prospects **61**
4.0.1 Looking Forward 61
4.0.2 Conclusion 61

Bibliography **64**

Appendix A Appendix **65**
A.0.1 UI of the web app 65

List of Figures

1.1 Screenshot of the AWS management console displaying the deployed subnets. The network topology is not easily discernible from the subnet list alone, as each subnet is linked to its respective VPC by its ID (vpc-...). Consequently, it is challenging to determine the exact structure of the topology, including the number of subnets associated with each VPC (in this case, 2 subnets per VPC) 2
1.2 Comparison of cloud service providers as evaluated by Gartner . . 2

2.1 Monolithic vs Microservices Architecture by Medium 4
2.2 Stack Overflow Survey 2022 8
2.3 Most popular JavaScript frameworks from here 8
2.4 Speed comparison of different languages 11
2.5 Three tier application 14
2.6 Authentication Sequence 14
2.7 JWT Token structure from Securitum 16
2.8 Illustration of Our Test Infrastructure 18
2.9 Example of Swagger API documentation from Swagger Hub . . . 20

3.1 Architecture of the Application 21
3.2 Service details 25
3.3 Provider structure 26
3.4 Adapting the Visitor design pattern with React hooks to display component details 27
3.5 UML of services used to store information that will then be displayed 28
3.6 UML model of the classes used to represent the graph 31
3.7 UML of the frontend API for backend communication 32
3.8 Architecture with a focus on the authentication component 34
3.9 Sequence diagram outlining the authentication process 35
3.10 Database diagram of the Authentication microservice 36
3.11 The Swagger hostname with the base path visible. It starts with `localhost:2000/api`, implying that any subsequent route (e.g., user) should follow the `/api` prefix, resulting in a complete path like `/api/user`. 39

3.12 Sign-in route depiction: We can see that it utilizes a POST method. The request's body is expected to contain two key values (i.e., email and password). The response will be a code that varies depending on the parameters received and processing. The format returned is JSON, with either two or three fields. Upon a successful sign-in, the 'data' field contains the result (a string in this case). Additionally, the response includes a 'message' and a 'status' to provide comprehensive information about the response. 40
3.13 Flow diagram depicting request handling in the main gateway . . 44
3.14 Main Gateway database structure 47
3.15 This basic topology presents a Vpc, which encompasses two subnets. Every subnet is linked with an ACL and a route table, and each one houses an EC2 instance secured by a security group. The red arrow illustrates the route from instance A (`10.10.10.50`) to instance B (`10.10.20.50`). 50
3.16 A portion of the UML utilized by the reachability service to execute the reachability algorithm. This primarily focuses on the routing aspect, with any redundant information omitted, which is why most classes don't display any fields/methods. 53
3.17 Architecture focusing on the reachability interaction 57
3.18 Traffic between the main gateway and the reachability service. The main gateway sends an id (actually it will be several id, e.g. id_user...) that the reachability microservices will then use to query subsequent services configurations. 57
3.19 Potential security breach when using IDs as context for the reachability service to retrieve configurations 58
3.20 Token-based reachability query. The token carries context information, and the hacker cannot fabricate a token. 59

4.1 Stack of languages and for each the number of lines coded for this application 63

A.1 User login interface. This interface initiates user login by forwarding requests to the authentication microservice and retrieving a token for further interactions. 66
A.2 Infrastructure selection interface displaying two saved infrastructures. Each infrastructure is identified by a name. The yellow button enables viewing the deployed online topology. A user settings option is available on the top right corner of the toolbar. . . . 66

A.3 Main page showcasing your infrastructure on the right, with visible details of three VPCs, including their CIDR range and name (if tagged). Subnets are shown along with their CIDR range and last digits of their ID (with an option to display their name). The left pane contains two tabs, for viewing clicked service details and performing reachability testing. 67
A.4 Subnet detail view, displaying relevant service information upon clicking on a subnet. Top of the interface presents details such as the AZ, ID, etc., while the bottom segment lists tags and associated network ACL with inbound and outbound rules. 68
A.5 Detailed view of an EC2 instance, displaying network interfaces and associated security groups along with their respective rules. . 68
A.6 Reachability test interface, where "source" and "destination" are selected by clicking the hand icon next to them. User then chooses the protocol and port, upon which a request is sent and the path is displayed. In this instance, a successful path is shown, with individual steps detailing why the traffic was forwarded. 69
A.7 Display of an unreachable path, showcasing the reason for unreachability and the final step without a check mark, indicating that the packet was not forwarded. 70
A.8 Interface view in dark mode, activated by toggling the dark mode button in the settings. This mode adjusts all primary and secondary colors. The same procedure is used to apply Cisco colors. . 70
A.9 Page 2 (showcasing the routes) 73
A.10 Page 2 (showcasing the models) 73
A.11 PDF format of the interactive Swagger documentation typically viewed in the browser 73

List of Tables

2.1 SDK language support for major CSPs 9
2.2 Number of lines of code for each language 13
2.3 Comparison between Session Tokens and JWT Tokens 16

4.1 Summary of work performed across Git repositories 62

List of Abbreviations

CSP .... Cloud Service Provider

AWS .... Amazon Web Services

GCP .... Google Cloud Platform

JWT .... JSON Web Token

REST .... Representational state transfer

IaC .... Infrastructure as Code

VPC .... Virtual Private Cloud

API .... Application Programming Interface

NACL .... Network Access Control Lists

UI .... User interface

UX .... User Experience

SDK .... Software Development Kit

DSL .... Domain-Specific Language

CIDR .... Classless Inter-Domain Routing

1 | Introduction

1.1 Motivation

With the increasing adoption of cloud computing, more and more individuals and organizations are relying on cloud infrastructure to store, process, and analyse their data.

Cloud computing Cloud computing refers to the delivery of on-demand computing resources over the internet, including servers, storage, databases, and software applications. This technology offers numerous advantages over traditional on-premises infrastructure, including scalability, cost savings, and flexibility. By leveraging cloud infrastructure, organizations can quickly and easily provision the resources they need to support their business operations, without the need for expensive hardware and maintenance costs.

Visualizing the networking relations of services using the management console provided by cloud service providers can be challenging. As shown in Figure 1.1, the console typically presents information in a list format, which can make it difficult to see the relationships between different components of the infrastructure. It is like trying to visualize a city's streets by looking at a list of roads instead of using a map. As a result, it can be challenging for administrators to get a comprehensive view of their infrastructure and to identify potential issues quickly.

Furthermore, there are several cloud service providers (CSPs) in the market, including Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP), among others. Each CSP offers a unique set of features and pricing models, allowing organizations to choose the best fit for their needs. This diversity is illustrated in Figure 1.2.

However, most organizations do not rely on a single CSP for all their infrastructure needs (Tarraf, Cesarini, & Hughes, 2021). Instead, they often adopt a hybrid cloud approach, leveraging multiple CSPs to get the best of each platform. For example, an organization may use AWS for their compute needs, Azure for their data analytics, and GCP for their machine learning workloads. This hybrid approach allows organizations to optimize their cloud infrastructure based on cost, performance, and other factors, leading to a more efficient and effective cloud strategy.

Given the complexity of managing a hybrid cloud infrastructure, having a visualization platform that can provide a unified view of all cloud resources is essential. Such a platform can help organizations monitor their cloud usage, identify ineffi-

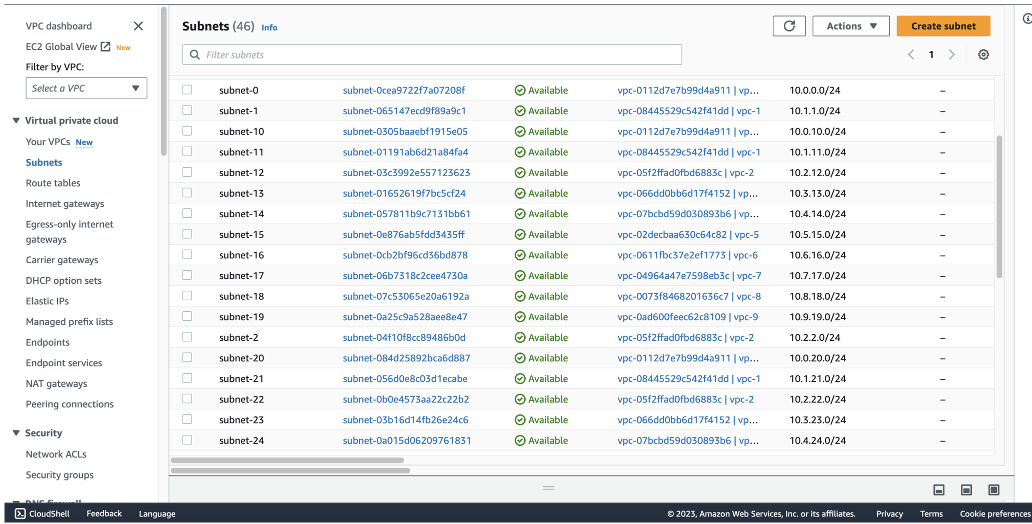

Figure 1.1: Screenshot of the AWS management console displaying the deployed subnets. The network topology is not easily discernible from the subnet list alone, as each subnet is linked to its respective VPC by its ID (vpc-...). Consequently, it is challenging to determine the exact structure of the topology, including the number of subnets associated with each VPC (in this case, 2 subnets per VPC)

Figure 1: Magic Quadrant for Cloud Infrastructure and Platform Services

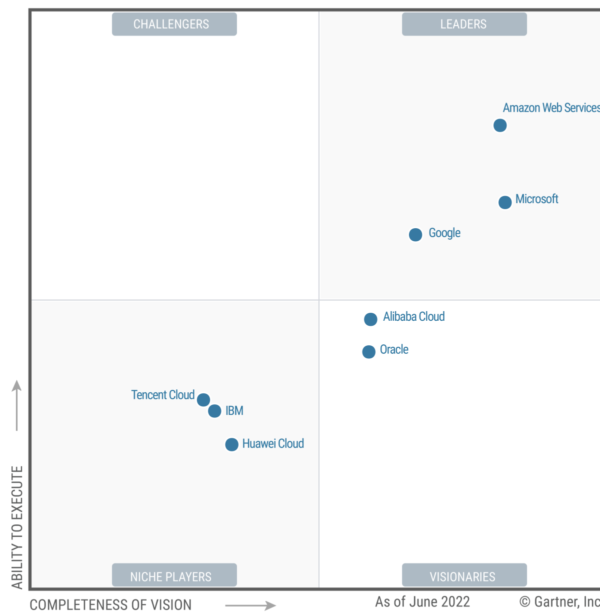

Figure 1.2: Comparison of cloud service providers as evaluated by Gartner

ciencies, and optimize their spending across all CSPs, leading to significant cost savings and improved performance.

Network Reachability One important aspect of managing a cloud infrastructure is ensuring that all servers and services can communicate with each other. From a configuration perspective, this means verifying that the necessary security groups, network access control lists (NACLs), and route tables are properly configured to allow traffic between servers.

For example, an administrator may want to test if an EC2 instance in one subnet can communicate with another EC2 instance in a different subnet, or if a server can reach a database hosted on a separate instance. By testing reachability between services, administrators can identify misconfigured security groups or NACLs that may be blocking traffic, or route tables that may be directing traffic to the wrong destination.

Having a visualization platform that can display the network topology of the cloud infrastructure and highlight potential connectivity issues can be incredibly valuable for administrators. By visualizing the connectivity between services, administrators can quickly identify potential issues and take corrective action before they impact users or cause downtime.

1.2 Aim and Objectives

The aim of our project is to address the challenges associated with visualizing cloud topology by developing a user-friendly web application. Our application will provide users with the ability to visualize their cloud infrastructure, regardless of whether it is a single cloud provider or a hybrid cloud environment.

We recognize that cloud infrastructures can be complex, and users may have numerous services that they need to manage. Therefore, we intend to develop a visualization platform that is not only aesthetically pleasing but also allows for easy navigation, filtering, and querying of the cloud services.

In addition to the above features, we plan to incorporate a simulation function in our application, which would enable users to simulate network traffic between their instances. With this feature, users will be able to verify the ability of their instances to communicate with each other using a particular protocol (e.g. TCP) and a designated port (e.g. port 22). This will help users identify potential connectivity issues and make necessary adjustments to their cloud infrastructure.

2 | Selecting Technologies

Identifying the appropriate infrastructure and technology stack is crucial for establishing a solid foundation that ensures the success of the application.

Our application development should adhere to the following principles (inspired by DevOps):

- Code-Centric Approach: We aim to have everything, from documentation to infrastructure and tests, as code. This allows us to run tests, automate processes, and maintain version control.
- Automation: By employing tools such as CI/CD or pre-commit hooks, we strive to detect potential issues and minimize time spent on debugging or redundant tasks.
- Modular and Comprehensible: We focus on designing separate modules so that understanding a specific part of the system does not require in-depth knowledge of the entire system.

2.1 From Monolithic to Microservices Architecture

The evolution of software architecture has seen a significant shift from monolithic to microservices-based designs. This transformation has impacted the way organizations develop, deploy, and manage applications (Davis, 2022). The Figure 2.1 shows this difference.

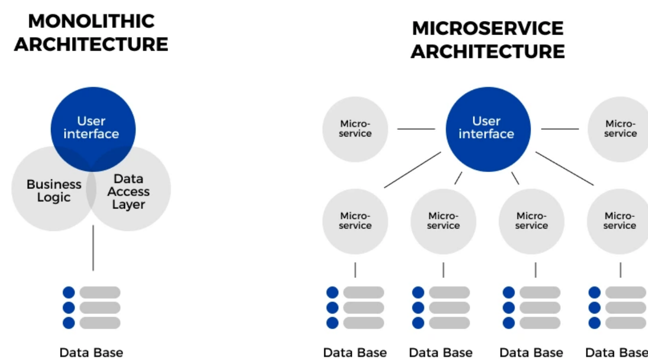

Figure 2.1: Monolithic vs Microservices Architecture by Medium

2.1.1 Traditional Monolithic Architecture

Traditionally, applications were built using a monolithic architecture, where all components and functionalities were tightly integrated into a single unit. This approach made it simpler to develop and deploy applications initially, as everything was contained within one codebase. However, as applications grew in size and complexity, this design had several drawbacks, such as:

- Difficulty in scaling specific components, as the entire application had to be scaled
- Limited flexibility in adopting new technologies or making updates.
- Increased risk of failure, as a bug in one component could affect the entire application
- Longer build and deployment times due to the size and complexity of the codebase

2.1.2 Microservices Architecture: Advantages and Adoption

Microservices architecture addresses these challenges by decomposing an application into a collection of loosely coupled, independently deployable services. Each microservice is responsible for a specific functionality and can be developed, tested, and deployed independently. This approach offers several benefits, such as:

- Improved scalability, as individual microservices can be scaled according to their specific needs
- Enhanced flexibility, as each microservice can be developed using the most suitable technology stack and programming language
- Increased resilience, as the failure of one microservice is less likely to impact the entire application
- Better collaboration between teams, as each team can focus on developing and maintaining a specific microservice

2.1.3 Kubernetes and Microservices Deployment

Kubernetes (Kubernetes, 2023) is a container orchestration platform that simplifies the deployment, scaling, and management of containerized applications, making it an ideal choice for deploying microservices. In a Kubernetes cluster, microservices can be run on "`pods`", which are groups of one or more containers

with shared storage and network resources. Kubernetes provides several features that support microservices architecture, such as:

- Automatic scaling and load balancing of pods to accommodate changes in traffic and resource demands
- Self-healing capabilities to restart failed containers or reschedule pods on healthy nodes
- Rolling updates and rollbacks to minimize downtime during updates and recover from failed deployments
- Service discovery and routing to enable communication between microservices

By leveraging Kubernetes, organizations can streamline the deployment and management of microservices, ensuring optimal performance, resilience, and scalability.

2.2 Language Choices

Choosing the "best" programming language can be a daunting task, as no single language is universally applicable to all situations. Several factors must be considered when selecting a programming language for a specific task or project. In this section, we will discuss these factors and then explore how to choose the best language for various use cases in subsequent subsections.

Firstly, the **maturity of a team** with a particular language is an important consideration. In a company setting, it is essential to assess the skill levels and experience of the team members with the language being considered. A language that the team is familiar with will likely lead to a more efficient development process and minimize the learning curve.

Another critical factor is the language's **ability to execute** the intended task effectively. This often depends on the availability of libraries, frameworks, and tools that can simplify and streamline the development process. For instance, a language with a rich ecosystem of libraries can greatly speed up the implementation of complex tasks and reduce the amount of custom code needed.

The **popularity** of a language should also be considered, as it affects the amount of support and resources available. A popular language is likely to have a larger community of developers, which means more documentation, forums, and tutorials to help troubleshoot and learn from. This can be particularly helpful in addressing any challenges that may arise during development.

Ease of use is another essential criterion when selecting a programming language. If a language requires significantly more lines of code to achieve the same outcome, it can impact the development timeline and overall productivity. A language that is easy to read, write, and understand can reduce the likelihood of bugs and make it easier for team members to collaborate.

Finally, the **performance** of the language should be taken into account, as it can directly impact the end-user experience. A faster language can reduce the delay users experience when interacting with the application, leading to better satisfaction and overall usability.

In the following subsections, we will apply these criteria to various use cases to help guide the selection of the best programming language for each component.

Note: here the team is both the one doing its thesis (Ilias Sebati) and the Cisco Systems team that will reuse the code to extend and maintain it.

2.2.1 Web Application

The user interface (UI) will serve as the primary point of interaction for users, making it crucial to develop an attractive and intuitive frontend (UX). Our objective is to create a visualization tool for network topologies, so the frontend must be capable of handling graph manipulation. Therefore, selecting a programming language with existing graph libraries is essential.

The 2022 Stack Overflow survey, as shown in Figure 2.2, reveals that JavaScript is the most popular programming language with 67% of the votes. Dart (used by Flutter) has a considerably smaller share at 6.67%.

JavaScript offers an extensive collection of libraries for creating diagrams, including the open-source library mxGraph, which serves as the foundation for the well-known draw.io web application. We will discuss this library in more detail later.

Initially, Flutter was considered as the primary choice for frontend development, given my experience with the language. However, JavaScript currently dominates web development, and Flutter's lack of graph libraries proved to be a significant disadvantage.

Based on these factors, JavaScript was chosen as the preferred language for this project. However, there are numerous JavaScript frameworks available:

2.2.1.1 JavaScript Frameworks

Figure 2.3 illustrates the most popular JavaScript frameworks.

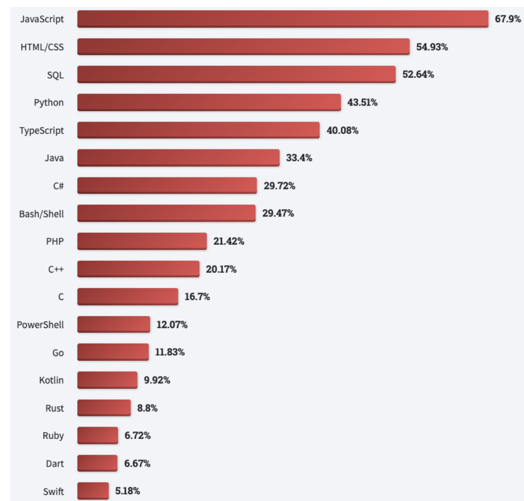

Figure 2.2: Stack Overflow Survey 2022

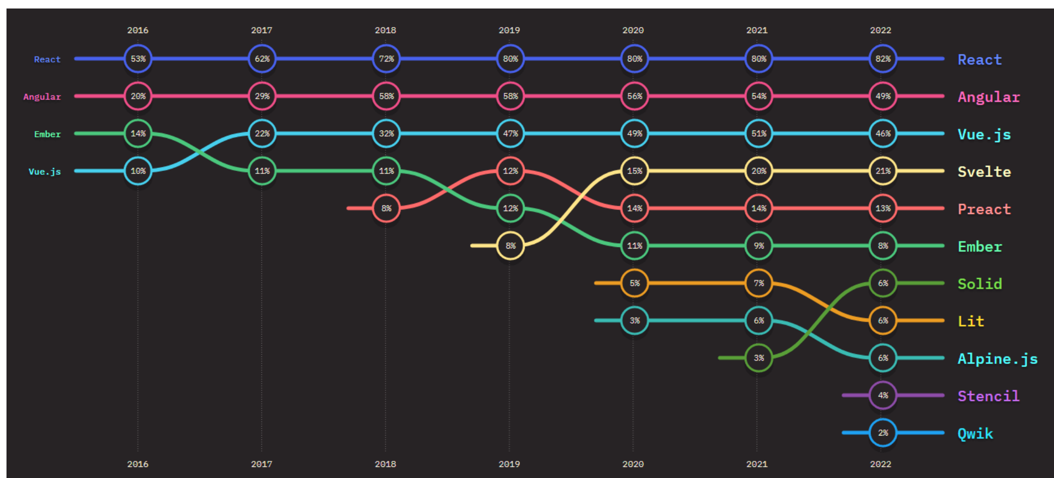

Figure 2.3: Most popular JavaScript frameworks from here

React has consistently been the most widely used framework since 2016 and remains so today. Consequently, we have decided to use React for this project.

2.2.2 Backend language

As said in Section 2.1, we will be using a microservice architecture. Which means that each microservice will be running independently. Hence, each microservice will be developed with the most appropriate language for the task.

One microservice deployed will be the responsible to fetch the configurations of the service using the CSPs' APIs. Each CSP provide its own set of libraries to talk with their APIs also called a **Software Development Kit** (SDK). It is mandatory to choose a programming language that has an SDK for at least the three biggest CSPs (AWS, Azure, GCP).

As the SDK can be used with a lot of programming languages, finding the most efficient one is a key point to develop good applications.

2.2.2.1 Choosing a language

To decide on which language to use, we first have to select a language that is proposed by the three CSPs' SDK.

	C++	Go	Java	JS/Node.js	Kotlin	.NET	PHP	Python	Ruby	Rust
AWS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Azure	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗
GCP	✗	✓	✓	✓	✗	✓	✓	✓	✓	✗

Table 2.1: SDK language support for major CSPs

As we can see, only Go, Java, Node.js, .NET, and Python can be used.

As a requirement from my manager, Java cannot be used. Knowing that .NET uses C# which has been created by Microsoft to be "their" version of Java (it is quite similar), we will avoid this one too.

This leaves the three following languages:

- Go
- Node.js
- Python

In this application, we would like the backend to be able to fetch the services running on the cloud providers, clean those data (following some filtering required

by the caller of the route), and return those data to the frontend application that will then have all the fun of making this data useful.

2.2.2.2 Language comparison

Python Python is really easy to use and has a lot of support online. However, it is known to be a slow programming language. Using the boto3 package, fetching the service can just be done in one line of code:

```
boto3.client('ec2', region_name="us-east-1").
    describe_vpcs()
```

It is really convenient. The Documentation is also really clear.

A nice point of Python is also its Object-oriented part. Creating classes and inheritance is easy and powerful.

Go Go is a programming language released by Google in 2009. It is famous for its go routines. It is a programming language used to handle a lot of simultaneous requests. Those go routines can be used to "clean" the data received from the CSPs really fast.

The code is a bit longer (essentially due to the error handling done in Go, i.e. not using exceptions). To fetch the VPCs as above:

```
result, err := svc.DescribeVpcs(context.TODO(), &ec2.
    DescribeVpcsInput{})
if err != nil {
    fmt.Println("Got an error retrieving vpcs:")
    fmt.Println(err)
    return
}
```

The documentation is also quite clear (but a point is given to Python).

As opposed to Python, Go does not have classes. However, you can create methods to structures which can lead to a similar result.

Node.js Node.js is famous for its programming language. Indeed, you can create a backend in JavaScript. Thus, one may do the frontend and backend using the same programming language.

It seems like Node.js is not a popular option if you do not know JavaScript. It has, however, the most support as it is one of the most used programming languages.

To fetch the VPCs, it is also quite simple:

```
var ec2 = new AWS.EC2({apiVersion: '2016-11-15',
   region: 'us-east-1'});

ec2.describeVpcs({}, function(err, data) {
        if (err) {
            reject(err);
        } else {
            resolve(data);
        }
    });
```

The JavaScript documentation is also quite nice. A lot of support is provided.

2.2.2.3 Speed comparison

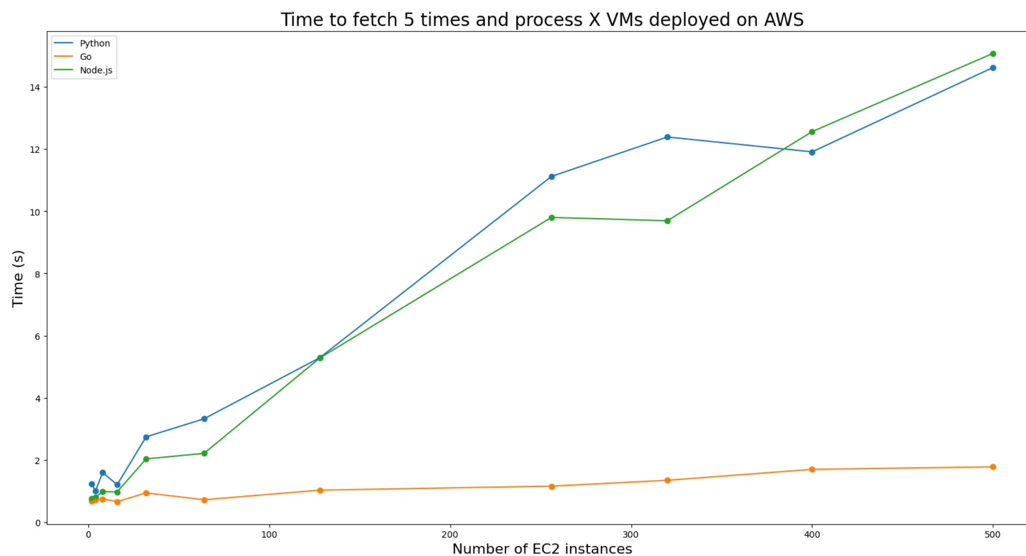

Figure 2.4: Speed comparison of different languages

To compare the speed, we have created a Terraform file that can create an infrastructure at will. The test file `measurements.py` is going to make 5 requests simultaneously to the server Python (then Node, and Go). Upon receiving those requests, the servers will fetch the services of AWS. Then, it will search in those services for tags matching a pattern (the process part).

The time displayed is the time needed for each server to complete those 5 requests.

The services fetched:

- VPC
- Subnet
- Network interface
- NACL
- Security Group
- EC2 instance
- Internet Gateway
- Route table

We can see in the Figure 2.4 that Python and Node.js are taking a lot of time to process the requests when they have a lot of EC2 instances. Their computation time is growing linearly with the number of resources. While the computation time for Go is pretty much constant.

Why is Go so much faster? Go is much faster than Python and Node.js for several reasons (Cheney, 2020):

- **Concurrency**: Go has goroutines, which are lightweight threads managed by the Go runtime. They enable efficient concurrent execution, allowing Go to handle multiple tasks simultaneously with a small memory overhead compared to traditional threads.
- **Compiled language**: Go is a compiled language, which means it is converted directly into machine code before execution. This allows for faster execution and better optimization by the compiler. In contrast, Python and Node.js are interpreted languages, which can lead to slower execution times.
- **Static typing**: Go uses static typing, which enables the compiler to catch errors and optimize the code at compile-time. This results in better performance at runtime compared to dynamically-typed languages like Python and JavaScript.

These factors contribute to Go's superior performance, particularly in handling concurrent tasks and processing large amounts of data.

2.2.2.4 Number of Lines Comparison

The number of lines of code has been computed using the cloc program. One may find a way of writting the code with much less lines of code. It is good to keep in mind that the documents have been written with this level of expertise:

- Python: Senior
- Node.js: Entry
- Go: Middle

Language	Number of Lines
Python	109
Node.js	183
Go	244

Table 2.2: Number of lines of code for each language

As we can see, Go requires more lines of code to accomplish the same tasks as Python and Node.js. Python is an expressive language, allowing more functionality to be written with fewer words. However, its threading system and speed are drawbacks.

Node.js's asynchronous system is not ideal, with the system of Promises being a significant constraint compared to Go's goroutines and waiting groups.

2.2.2.5 Conclusion

Considering its speed and ease of use, Go has been chosen as the preferred programming language for this project. Its superior performance in handling concurrent tasks and processing large amounts of data outweigh the additional lines of code required to be compared to Python and Node.js.

2.3 Authentication

When dealing with a three-tier application (frontend, backend, and database), this is important to consider the way that the services will communicate between themselves. Or saying it the other way, the security used by the different services APIs to answer requests.

The basic scheme of a three-tier application is the following:

As we see, a lot of communication between the frontend and the server will occur. We need to make sure that those communications are secure (by using https) and also to make sure that person who receive the ressources are indeed the person who have access to those ressources.

To do that, we will need to use tokens.

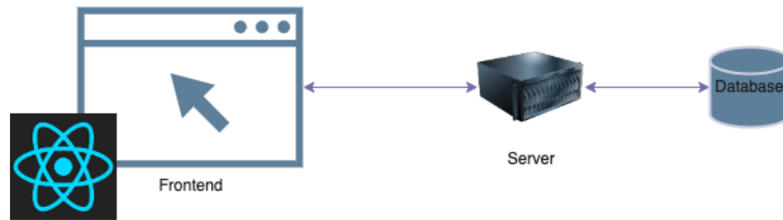

Figure 2.5: Three tier application

The server will be using tokens to authenticate the requests and thus know who is asking what. The token will be shared when the frontend will try to authenticate the user with their password and username. From there on, the next requests from the frontend to the server will be using the tokens. As illustrated in Figure 3.9.

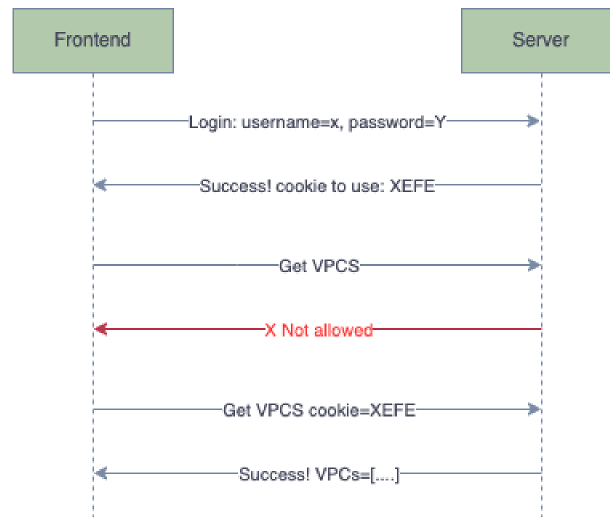

Figure 2.6: Authentication Sequence

Several types of tokens exists, let's review two of the most common ones:

2.3.1 Session Tokens

Session tokens are considered **stateful**. This means they are stored on the server side and change whenever the user makes new requests. When the user authenticates, a session token is created server-side, and an ID is returned. All upcoming requests contain the ID so that the server can recognize the user. Upon receiving a request from a user, the server increases the lifetime period of the token.

Pros:

- Ability to know who is connected. As we store all the IDs, we know who is connected and who is not.
- Ability to kick out everyone from the session and ask for new authentication.
- Less vulnerable to token theft, as tokens are not self-contained and require server-side validation.
- Easier revocation of access, as the server can immediately invalidate a session token.

Cons:

- Has to store the tokens, which can result in higher server overhead and storage requirements.
- May require more complex infrastructure for distributed systems or load balancing scenarios, due to the need for shared session state.
- Vulnerable to Cross-Site Request Forgery (CSRF) attacks if proper security measures are not implemented.

2.3.2 JWT Tokens

JWT tokens or JSON Web Tokens are **stateless** tokens, which means the server does not have to store any kind of information (Ahmed & Mahmood, 2019).

How do JWT tokens work

As we can see in Figure 2.7, JWT tokens are composed of three parts:

- **The header**: This part contains metadata and specifies important information, such as the algorithm used for signing the token (e.g., HS256 or RS256) and the token type (typically "JWT").
- **The payload**: Base64Url encoded, this section stores the actual data or claims you want to include in the token. It can contain user-related information such as the user ID, associated roles, and custom data. It's essential to include the token's expiration date, usually using the "exp" claim, to prevent indefinite token validity.
- **The signature**: This part is generated by signing the concatenation of the encoded header and payload, separated by a period (.), using the algorithm specified in the header. The signature helps ensure the token's integrity and verifies that the data has not been tampered with.

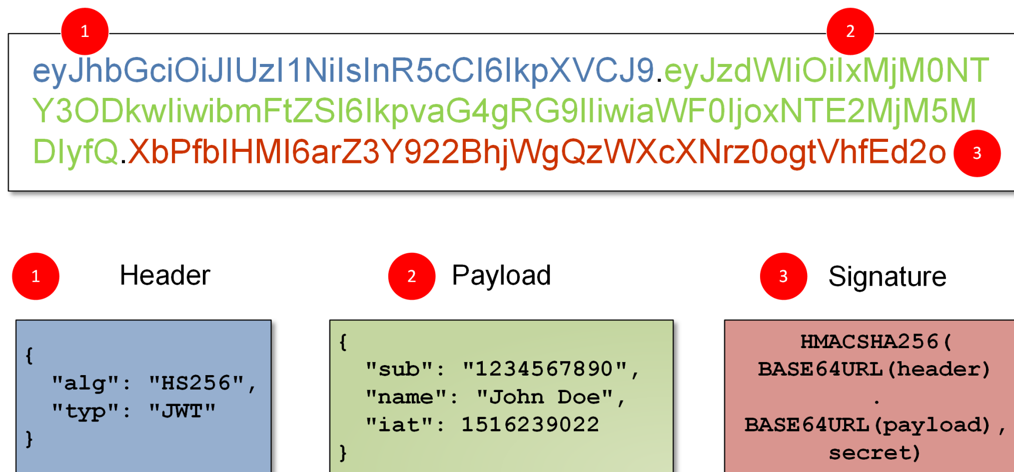

Figure 2.7: JWT Token structure from Securitum

Known attack of the no signing algorithm One known attack on JWT tokens is when an attacker forges a token and claims that the signing algorithm is "None". This can potentially bypass the signature verification process, allowing the attacker to impersonate a user or gain unauthorized access.

To protect against this attack, it's important to always specify a list of allowed signing algorithms when validating JWT tokens and to never include the "None" algorithm in that list.

	Session Tokens	**JWT Tokens**
Can know who is connected	✓	✗
Need storage	✓	✗
Need key rotation	✗	✓
Self-contained	✗	✓
Vulnerable to CSRF	✓	✗

Table 2.3: Comparison between Session Tokens and JWT Tokens

Session Tokens vs JWT Tokens We chose to use JWT tokens for our application because they provide lower overhead compared to session tokens. Additionally, our application did not require the ability to track connected users or forcibly disconnect them, making JWT tokens a more suitable option.

2.4 Infrastructure

As our application aims to visualize an infrastructure, we need to deploy one. CSPs offer various methods for deploying services, with the two most prominent approaches being the use of the management console and Infrastructure as Code (IaC).

The management console The management console is a web-based interface that allows users to manually create, configure, and manage cloud services. However, the main drawback of using the management console is the lack of automation in the process. Each time a service needs to be created, deleted, or modified, manual intervention is required, making it less efficient and prone to human error.

Infrastructure as Code (IaC) In line with DevOps principles, we aim to adopt a code-centric approach for every aspect of our project, including infrastructure management. IaC (Artac, Borovssak, Di Nitto, Guerriero, & Tamburri, 2017) allows us to define, provision, and manage cloud resources using code, which brings several advantages, such as:

- Automation: Processes can be streamlined by creating scripts or pipelines that execute the IaC code, significantly reducing manual effort and minimizing errors.
- Versioning: IaC code can be stored in a version control system like Git, allowing for tracking changes and reverting to previous versions if necessary.
- Collaboration: Version control systems like Git facilitate code sharing and collaboration among team members, improving the overall development process.

Given these benefits, the next step is to choose an appropriate IaC language. Most CSPs provide their own domain-specific language (DSL) to deploy and manage their services. However, since our infrastructure is designed to be hybrid cloud (utilizing multiple CSPs), it makes sense to select a language that is independent of any specific CSP. HashiCorp's Terraform is a leading IaC tool that supports multiple CSPs and offers a consistent way to define and manage cloud resources across different platforms. By using Terraform, we can ensure flexibility and adaptability in our hybrid cloud infrastructure.

2.4.1 Our Test Infrastructure

To thoroughly test our application, we implemented a comprehensive test infrastructure, the details of which can be found in our GitLab repository at this link.

The test infrastructure we deployed is visualized in Figure 2.8.

Figure 2.8: Illustration of Our Test Infrastructure

As shown in the figure, the infrastructure design takes a parameter, `n`, which denotes the number of Virtual Private Clouds (VPCs). It will create `n` VPCs, with each VPC comprising the following components:

- A VPC named `vpc-i`, where `i` represents the VPC number, with a CIDR range of 10.i.0.0/16.
- Two subnets within each VPC, named `vpc-i-subnet-1` and `vpc-i-subnet-2`.
- An EC2 instance within each subnet, named `vpc-i-instance-subnetNumber`.
- A network interface associated with each EC2 instance, which equips the instances with networking capabilities.
- A NACL and a route table for each subnet to manage access and routing policies.

- An internet gateway connected to each VPC, facilitating communication with the internet.

This test infrastructure design enables us to create multiple VPCs with the required subnets, instances, and networking resources, allowing for thorough testing and evaluation of our application under various scenarios and conditions.

2.5 Swagger Documentation

In our microservices-based application, it is essential to provide comprehensive and easily accessible documentation for each microservice's routes. This documentation serves as a reference guide, allowing developers and users to understand the available endpoints and their functionalities without needing to examine the underlying code, which may not always be readily accessible.

One widely adopted framework for documenting REST APIs is Swagger API. Swagger provides a powerful set of tools for designing, building, and documenting APIs. One of its key features is the generation of interactive and user-friendly API documentation.

An example of Swagger API documentation (Musib, 2019) is depicted in Figure 2.9. The Swagger UI provides a visually appealing and intuitive interface where users can explore the available endpoints, view request and response schemas, and even test the API directly from the documentation page.

To make the documentation easily accessible, it is typically hosted on a dedicated route within the microservice. For instance, if the microservice is deployed at `http://localhost:8080/`, the Swagger documentation can be accessed at `http://localhost:8080/api/index.html`. This centralized location ensures that developers and users can quickly find the relevant documentation without having to search extensively.

One of the significant advantages of Swagger API is its ability to generate documentation automatically from code annotations. By annotating structures, functions, and API endpoints with Swagger-specific tags and descriptions, the documentation can be automatically generated. This process may involve running a command (e.g., `make doc`) to trigger the generation of the documentation file. With this approach, the documentation remains up to date as it reflects the current state of the codebase. Additionally, Swagger provides powerful features beyond documentation, including request/response validation, parameter exploration, and even client SDK generation.

In our specific case, we will rely on Postman for route testing, as it offers a robust

environment for sending requests and inspecting responses. Postman complements Swagger's documentation capabilities by allowing developers to execute HTTP requests against the microservice's endpoints and verify their behavior.

By leveraging Swagger API documentation and tools like Postman, we aim to streamline the development and usage of our microservices, providing clear and accessible information to developers and users alike.

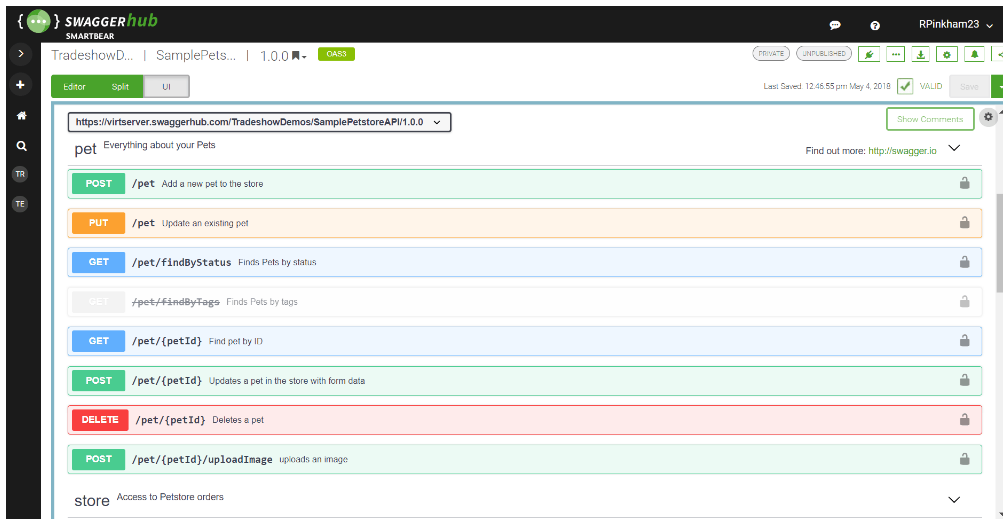

Figure 2.9: Example of Swagger API documentation from Swagger Hub

3 | An Exploration of the Methodology: Components and Infrastructure

As we embark on the journey to develop this application, the initial step involves unraveling and understanding the different components that come together to form the structure of the application. As outlined in Section 2.1, the infrastructure we are utilizing is based on a microservice model. This model deviates from traditional backend designs, as it isn't just a single server but comprises multiple servers, each responsible for a specific service.

In this chapter, we will delve into the details of each component, providing a dedicated section for each to gain a comprehensive understanding of their unique roles and functionalities.

To begin with, let's illustrate the architecture of our application for a better understanding of how these components interacts:

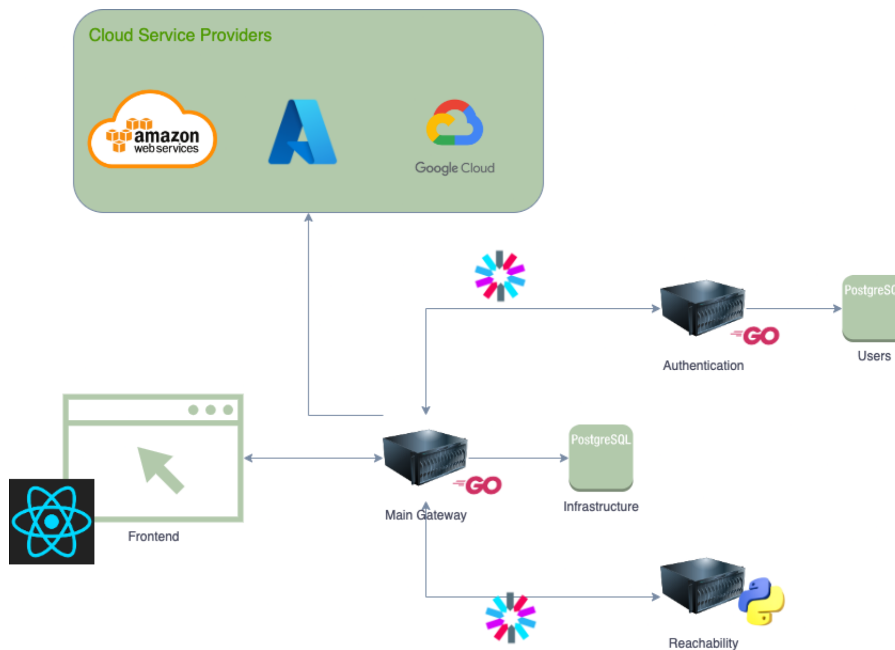

Figure 3.1: Architecture of the Application

From the above representation, it can be observed that the infrastructure of this application is composed of four main components:

- **The Frontend:** Developed using React, this component is the user-facing part of the application. It is designed to facilitate direct interaction with the customer, providing an intuitive and responsive user interface.
- **The Main Gateway:** This component serves dual purposes. It interacts with the CSPs' APIs to fetch and store the infrastructure data, and it also functions as a gateway for the frontend to query the multiple servers that make up our backend.
- **The Authentication Service:** This component is tasked with user authentication. It manages the process of logging users in and out of the application, ensuring secure access to the application features.
- **The Reachability Server:** This component is responsible for performing reachability tests between services. Its job is to simulate the traffic between services and return the full path as well as some feedback regarding the reachability.

The sections that follow will provide an in-depth analysis of each of these components, shedding light on their roles, functionalities, and the algorithms chosen to develop those components.

3.1 An In-depth Look at the Frontend

For all the reasons mentioned in Section 2.2.1.1, we will be using React to develop our client-side interface. React presents a robust foundation for our application, thanks to its support of highly interactive web interfaces.

React is based on the notion of components. Each component acts as an independent entity that can be visually represented with minor variations, based on the parameters used during its instantiation. There is flexibility in the creation of these components; we can custom-build them as per our requirements or utilize pre-existing component libraries.

To align our frontend design with the aesthetic of other Cisco products, it would make sense to explore the possibility of employing a Cisco component library.

3.1.1 Cisco UI Kits

Cisco provides an extensive array of UI kits that are instrumental in constructing web applications in line with their existing design language.

Among these, one particular UI kit named `React Cisco UI` initially caught our attention. However, our journey with this library was fraught with challenges. Despite persistent efforts, we faced numerous issues, some of which led to significant problems. Upon reaching an impasse, we reached out to the community for support via an internal channel. To our surprise, we discovered that the library was no longer actively maintained, a fact that was unfortunately not mentioned in the documentation (since it's no longer maintained!!).

We asked about alternative Cisco libraries suitable for React. The most common response was:
Cisco suffers from too many competing UI libraries.

They advice us to use Material UI for our project.

3.1.2 Material UI

Material UI is a widely embraced library within the React ecosystem, designed to facilitate the development of interfaces following Google's Material Design principles. Its popularity among the developer community is a testament to its robustness, versatility, and ease of use, especially when it comes to creating a library of React components.

One of the standout features of Material UI is its built-in support for various themes, including a dark mode. This feature makes it simple to tailor the aesthetics of an application while preserving the underlying functionality. For our purposes, this means we can create a unique theme that aligns with Cisco's design language without having to modify the existing codebase extensively. It's a perfect blend of customization and preservation of the original design elements, allowing us to maintain a consistent user experience.

The library's popularity comes with another significant advantage: a substantial and active support community. A widely-used library like Material UI is continually being tested, updated, and expanded by developers worldwide. This means any issues or challenges we might face are likely to have been encountered and addressed by someone else in the community. Therefore, troubleshooting becomes more streamlined, and the availability of community-created resources like tutorials, guides, and forums further eases the development process.

In summary, Material UI provides a comprehensive, user-friendly solution for creating dynamic, aesthetically pleasing web applications in line with Google's Material Design. Its support for custom themes, coupled with a strong and active community, make it an excellent choice for our frontend development.

3.1.3 Navigating with React Router Dom

In React, we use a router for smooth page transitions without needing a new request each time. There are several router libraries in React, and we've chosen React Router Dom for its lightness compared to others solution.

To start, we need to define our application's routes. We have four pages:

- **The Login page:** Handles user authentication and fetches the token.
- **The Profile selection page:** Here, you choose the infrastructure to view.
- **The Settings page:** Add your API keys here, which will be used for fetching services in the CSPs.
- **The main page:** This page displays your infrastructure diagram.

Setting up these pages involves using the `Routes` and `Route` hooks. Here's how it looks:

```
<Routes>
    <Route path="/login" element={<SignIn />} />
    <Route path="/" element={<ProfileSelection />} />
    <Route path="/settings" element={<
       AccountSettingsScreen />} />
    <Route path="/main" element={<MainScreen />} />
</Routes>
```

As you can see, we simply map the path to the component.

To navigate to a different page within the application, we use the `navigate` hook and specify the destination path:

```
  const navigate = useNavigate();
  navigate("/")
```

3.1.4 State management

State management refers to the sharing of state across multiple components. While there are numerous techniques for this, including the well-known Redux, it can sometimes be overkill. For many applications, React's built-in useState hook is sufficient and easier to use (Patil & Javagal, 2022).

For its utilization, we constructed components that acted as wrappers around other components. Through the "children" props of React components, state was propagated down the tree.

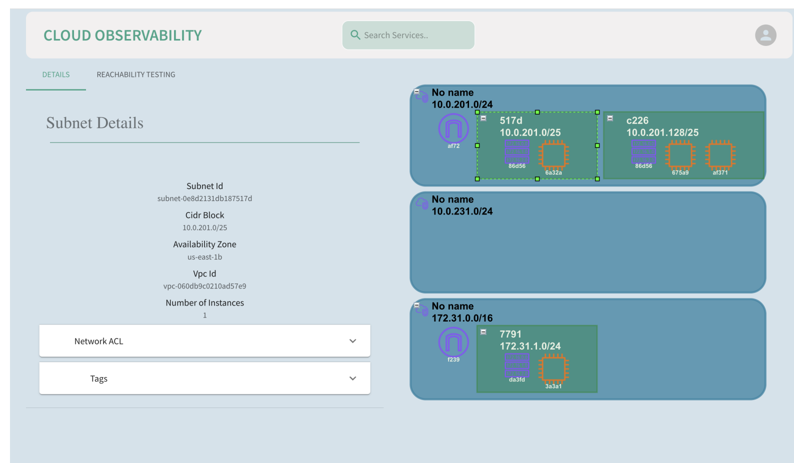

Figure 3.2: Service details

In our application, a feature exists that allows us to get comprehensive details about a service upon its selection, as shown in Figure 3.2.

There are two components, the graph and the detailed view, which require interaction. To utilize this provider, we simply place the main page component (which contains both the graph and the detail view) within the `ClickedServiceProvider` component as follows:

```
<ClickedServiceProvider >
    <Content />
</ClickedServiceProvider>
```

Consequently, all the elements contained within my content will have access to the data within this provider.

The structure of the component is illustrated below:

The main page functions as a parent to two components: `DetailedView` and `GraphView`. Both of these act as "notifiers" for the `ClickedProvider`. When the user interacts with the graph, the `GraphView` notifies the provider and passes along the service information. This triggers the `DetailedView` to rerender and show the details of the clicked service.

This is just one example of how we use state management in our application. We employ this paradigm in many other places throughout the app.

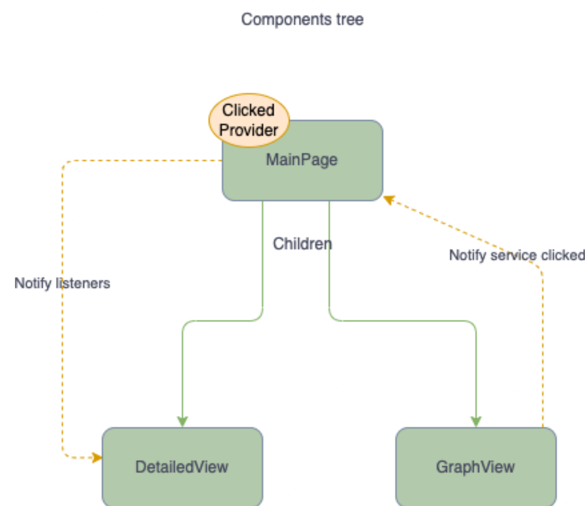

Figure 3.3: Provider structure

3.1.5 Using Object-Oriented Principles

Thanks to TypeScript, which allows the use of classes in React, we've employed object-oriented principles to make our code cleaner and more efficient.

We'll now explore some fascinating uses of object-oriented programming in our application. We'll showcase these with the help of UML diagrams for each class.

3.1.5.1 Showing Detailed Information

As mentioned earlier, our app presents detailed information of the clicked service. Each service comes with its own set of data. For example, an EC2 instance might need to display associated network interfaces to reveal the corresponding IP addresses. However, if we're showing details of a subnet, we'd need to show different information, like the route table or associated Nacl.

We took inspiration from the well-known visitor design pattern to achieve this. However, we only drew inspiration from it as we're using React hooks for our components instead of React class components.

As shown in Figure 3.4, the `DetailsView` component uses its function (React hook) `drawDetailService` to display a component with the service details. This `drawDetailService` function is our take on the "visit" class from the conventional visitor design pattern.

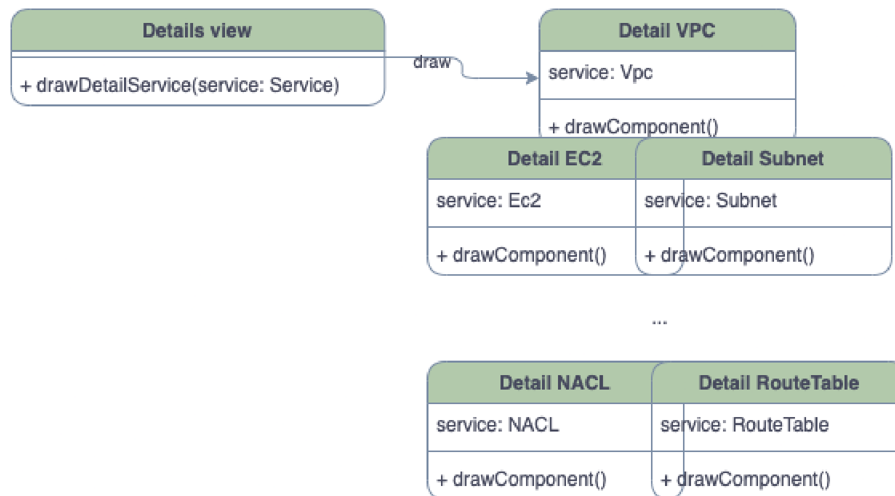

Figure 3.4: Adapting the Visitor design pattern with React hooks to display component details

3.1.5.2 Model

In order to display various services, we first need to store them. We do this using classes, which the components then utilize to display relevant information.

Refer to the service model in Figure 3.5.

The model caters to all possible services from different CSPs. In our project, we focused only on AWS due to time constraints. However, our `Service` abstract class makes it easy to add services from other CSPs.

Let's delve into the more intriguing classes.

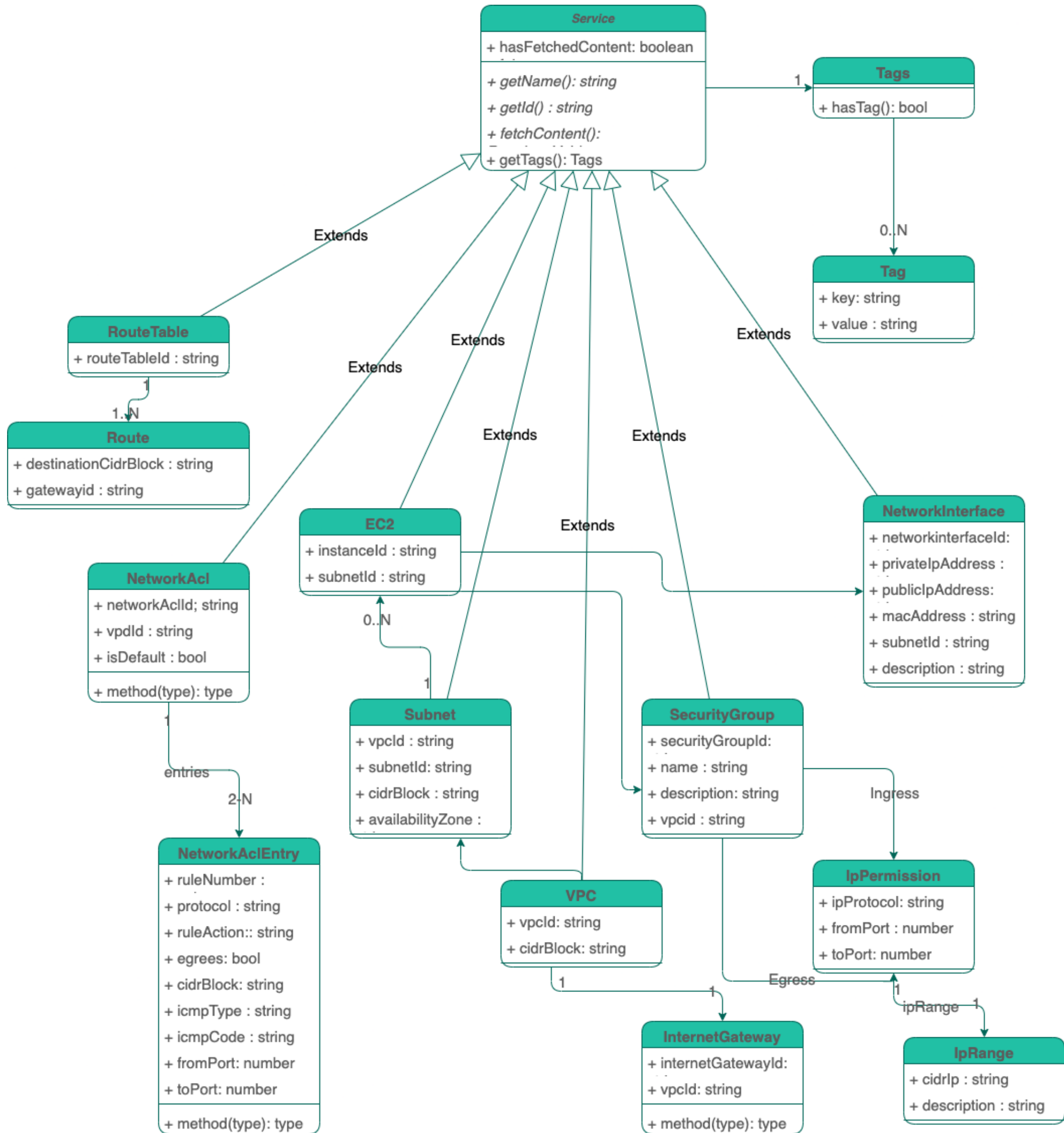

Figure 3.5: UML of services used to store information that will then be displayed

Service The `Service` abstract class is at the top of the hierarchy and outlines the minimal features a service should possess. We use the methods system because the location of these pieces of information may vary depending on the CSP. In our case, a service must include:

- A Name: Typically represented by the *Name* tag.
- An ID: Crucial for fetching subservices of a service (e.g., firewalls of a virtual machine).
- A FetchContent method: This method is vital as it helps limit the memory consumption of our application. Each service is initially fetched with basic, high-level information. When we want to see more detailed information about a service, we call this method to fetch advanced information, like the firewall associated with a virtual machine.
- Some Tags: Regardless of the cloud service provider, each service can have tags associated with it. These tags provide additional information about the service, such as the name of the application it is part of or the name of the environment (e.g., dev).

Every service extends this class to maintain a consistent interface, enabling us to display common information (e.g. id) across all services.

VPC The `VPC` class stores the CIDR block and the region name of the current VPC. Initially, only these pieces of information are stored. When we need to fetch the content of the VPC, we retrieve data about the `Internet Gateway` and the subnets within this VPC.

Subnet The `Subnet` class represents a subnet with a specific CIDR block and availability zone. When fetching its advanced content, we retrieve the associated route table and NACL, as well as all the EC2 instances within it.

Route Table The `Route Table` class holds information about the possible routes a packet can take. Therefore, it contains a list of `Route` objects. Each `Route` is defined by a destination CIDR block and a gateway ID. When running the Longest Prefix Match algorithm on all the route table entries, the gateway ID of the matching route signifies the next hop of the packet. In our application, the gateway ID could represent several things:

- Local: If the traffic is intended to go to another subnet within the same VPC.
- Internet Gateway: If the traffic is to be sent to the internet.

- A Service: If the traffic is meant to go to another service like a Transit Gateway or a NAT Gateway.

Note: On the frontend, we only display information. The routing algorithm, including Longest Prefix Match, etc., is handled by the microservice responsible for that.

NetworkAcl The `Network Access Control List (Network ACL)` class represents the firewall of the subnet. To fulfill its function, it needs a set of `NetworkAclEntry` rules to manage the traffic.

Each NetworkAclEntry holds:

- A rule number: This is essential as the order of the rules matters â the system will match the first rule and then apply the action.
- The protocol: The match can be based on the protocol.
- The rule action: Should the system allow or block the traffic? (Visual representation: allow in green and block in red)
- A flag indicating if the rule is for egress or ingress traffic.
- The CIDR block to match.
- The ICMP type, in case of ICMP traffic.
- The ICMP code, in case of ICMP traffic.
- The port range to match.

EC2 The `EC2` class stores fundamental EC2 information, as well as more detailed data like the list of `network interfaces` associated with the instance. It also keeps track of the list of security groups (e.g., firewalls) linked to the network interface.

SecurityGroup The `SecurityGroup` class represents an instance-level firewall. Hence, it needs to store rules. It maintains two sets of rules:

- Ingress rules for traffic going into the instance.
- Egress rules for traffic leaving the instance.

Both these rules are instances of the `IpPermission` class, which stores the IP protocol, the range, and the CIDR IP for matching packets.

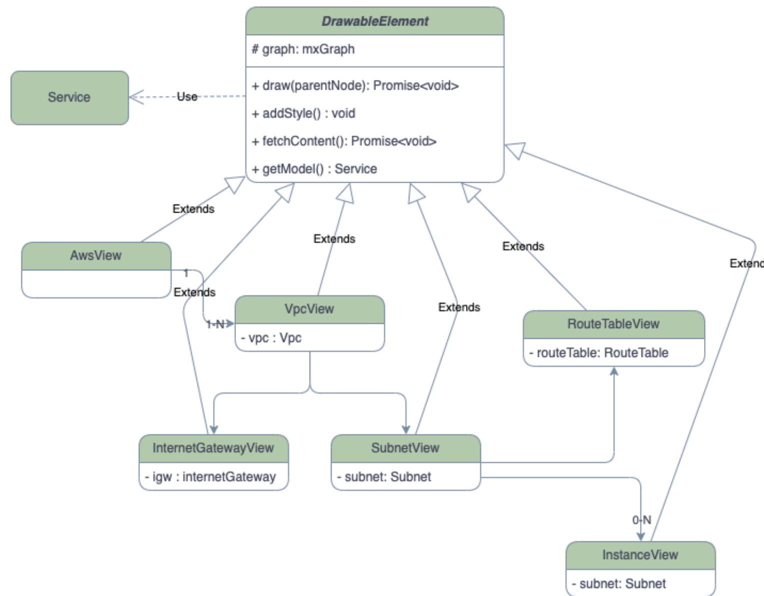

Figure 3.6: UML model of the classes used to represent the graph

Graph As previously mentioned, we're using the mxGraph library to render our graphs. Since mxGraph is a JavaScript library, we need to create our own graph components. We chose to use class-based components for this, as they provide more clarity when managing states.

The UML of our graph classes is shown at Figure 3.6.

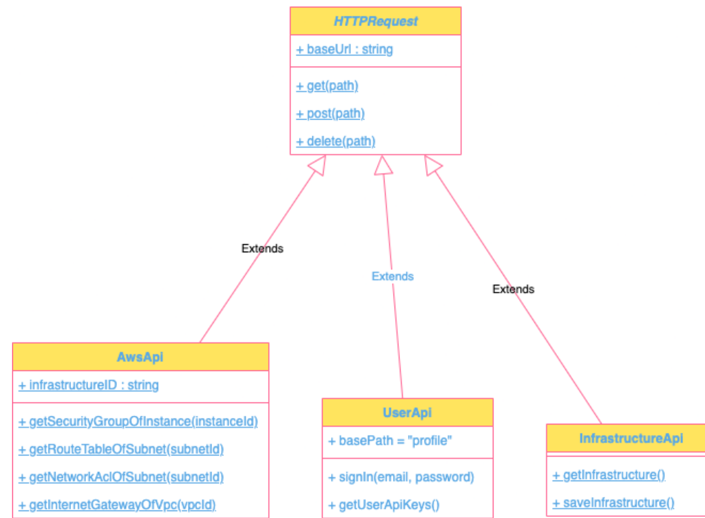

Figure 3.7: UML of the frontend API for backend communication

We have two types of components:

1. The `DrawableElement` class is the main, top class and the parent class of all others. Each DrawableElement is an element that can be drawn using mxGraph, which is why the graph is one of the fields of the class. Each DrawableElement has a `draw` method used for rendering the current service on the graph, with the parent node as an argument. It also has an `addStyle` method for adding the service's style to the mxGraph style set. The `fetchContent` method, similar to the one in the previous section, fetches detailed information about a service.

2. Service-specific classes extend the `DrawableElement` class. These services implement the methods defined in the DrawableElement class. Additionally, each service has its own model as defined in the previous section (e.g., a SubnetView has a Subnet instance associated with it).

Together, they form the main components of the graph. The `DrawableElement` abstract class allows us to easily expand the set of supported services without worrying about their specific implementations.

3.1.5.3 Server Communications

The frontend interacts with the backend services for various purposes such as user authentication, fetching services, saving infrastructures, and more. In this section, we'll explore how these requests are managed in our web application. As illustrated in Figure 3.7

The cornerstone of our architecture is the abstract class `HTTPRequest`. This class, alongside its descendants, follows the Utility design pattern, with all fields and methods being static. The class holds a `baseUrl`, which represents the backend gateway's URL. It also incorporates basic HTTP methods such as GET, POST, and DELETE.

The `AwsApi` class caters to AWS-specific requests to the backend. It encapsulates all the specific routes used in the frontend, like a route to fetch the security groups linked to an instance. This architecture provides an abstraction layer for interacting with the API, wherein each function just accepts the parameters and knows the structure of the route. In scenarios where multiple cloud service providers are in use, adding a new provider like Azure would only necessitate an additional `AzureApi` class with the corresponding routes.

The `InfrastructureApi` class is employed to fetch and save infrastructure data. As the infrastructures span multiple CSPs, these routes can't be contained within the `AWSApi` or `AzureApi` classes, hence the need for a separate class.

Lastly, we have the routes for user management, encapsulated in the `UserApi` class. The base path for this class is set to "profile", as all user-related routes in the backend start with `/profile`. This includes routes for user sign-in, user creation, and addition of API keys for communication with cloud service providers.

3.2 Understanding the Login Service

In the course of utilizing our application, a user will need to input specific keys in order to establish communication with the Cloud Service Providers' (CSPs) API. This allows the user to interact with and manipulate the state of various infrastructures within the application. Additionally, the application offers the user the capability to store the state of certain infrastructures for later use. Despite any modifications made to the online infrastructure, these saved states will remain unaffected, providing a level of consistency for the user's interaction with the platform.

However, a critical limitation in the current system is the impermanence of user data. Once a user exits the project, all their associated data is lost, as there is no mechanism to track and associate that data with the user. The crux of the issue lies in the lack of user identification, making it impossible to attribute saved states or modifications to a specific user. A potential workaround for this might involve the use of cookies to maintain user sessions. However, this solution presents its own challenges. For instance, should a user's computer crash, or should they lose access to their browser, they would consequently lose access to their account and

all associated data.

This predicament underscores the necessity for a robust user authentication system. User authentication forms a crucial part of any application, providing it with the means to assign an identity to a client. Once this identity is established, the application can retrieve and display the user's saved data, thereby enhancing their experience with the platform.

As discussed in Section 2.1, our proposed solution is the implementation of a microservice architecture, with a specific microservice dedicated to user authentication. This decision is primarily driven by its potential benefits to the development team, in this case, the Cisco team. Delegating the user authentication responsibilities to a microservice enables the team to manage this aspect of the application independently. For instance, they may wish to integrate Single Sign-On (SSO) with existing Cisco accounts. This architectural flexibility offered by microservices allows them to do so without risking the stability or integrity of the overall application.

3.2.1 Big Picture

As we delve into the intricacies of the authentication service, it is beneficial to first illustrate its position within the broader application infrastructure. The following image provides a visual representation of the architecture with a focus on the authentication component:

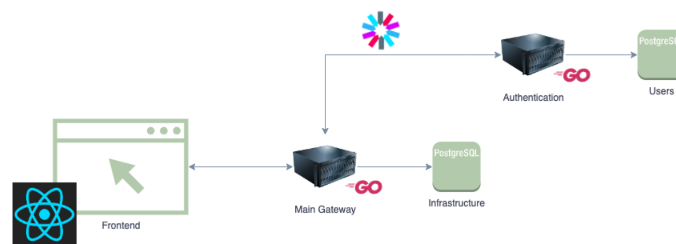

Figure 3.8: Architecture with a focus on the authentication component

Having established the structural context, it's now crucial to understand the flow of interactions between different services during the authentication process. The sequence diagram below outlines the high-level interaction pattern:

Initially, the user is presented with the login page on their browser, where they are prompted to input their credentials, i.e., the *email* and *password*. Upon submission, a request containing these credentials is dispatched from the client-side application (frontend) to the main gateway.

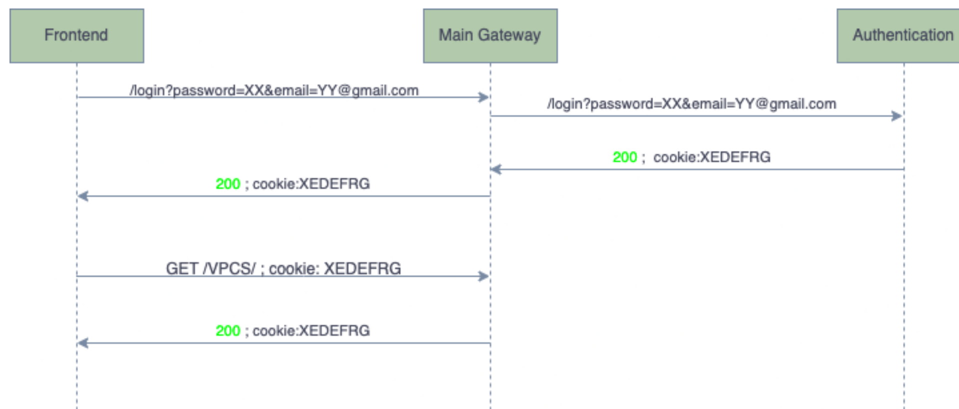

Figure 3.9: Sequence diagram outlining the authentication process

The main gateway, acting as an intermediary, relays this request to the authentication service. This service then processes the request, verifies the credentials, and sends a response back to the main gateway. This response, in turn, is relayed back to the frontend.

Incorporated within the response is a cookie, which becomes crucial for the subsequent interactions between the user and the application. Each ensuing request from the user to the main server (main gateway) must include this cookie.

The main gateway, acting as the gatekeeper, scrutinizes every incoming request for a valid cookie. If a valid cookie is detected, it extracts pertinent information such as the user ID and allows the request to proceed. However, if a valid cookie is not found, it responds with an appropriate error code, as outlined in the application's documentation. This process ensures that only authenticated users gain access to the resources and functionalities of the application.

3.2.2 Database

For storing essential information, the service utilizes a PostgreSQL database running on a Docker instance. The database diagram, depicted in Figure 3.10, showcases the following:

The database consists of a single table called the `users` table. This table is responsible for storing user data and includes the following fields:

- Email: Represents the user's email address, which serves as the username for authentication purposes.
- Password: Indicates the hashed version of the user's password. The password and salt are combined to generate the hash.

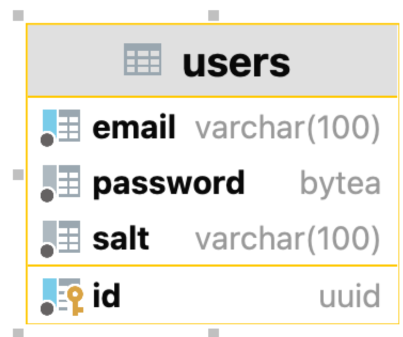

Figure 3.10: Database diagram of the Authentication microservice

- Salt: Refers to a random value employed in the computation of the password's hash. Each user possesses a unique salt.
- ID: The unique identifier assigned to the user. (data type: UUID)

3.2.3 User Authentication

When a user needs to log in, they send a request to the server following the format specified in the documentation (refer to Section 3.2.4 for details).

This request includes two elements:

- Email address
- Password (with no restrictions)

Upon receiving these elements, the server's route performs the following steps:

Check if the email address exists. If it doesn't, the server sends the response: *Either your password or email is incorrect.* This generic message is used instead of revealing whether the email address exists to mitigate potential brute-force attacks.

If the email address exists, the server concatenates the received password with the associated salt value (Sriramya & Karthika, 2015). Then, it applies the `SHA256` hash function to generate a hash. (Note: `SHA256` is considered a strong hash function at the time of writing, though stronger alternatives like `SHA512` exist, but a tradeoff between speed and security has to be made (Rahmatulloh, Gunawan, & Nursuwars, n.d.).) The resulting hash is compared with the stored hash in the database. If they match, the server proceeds to create the authentication token, as described in the following paragraph. If the hashes don't match, the server returns the previously mentioned error message along with the user's information.

Token Creation As mentioned in Section 2.3.2, we will employ JWT tokens for authentication. The token payload contains the following information:

```
{
    "exp": 1760640510,
    "id_user": "8ed44e42-8537-4c3b-80bb-e465055da34e"
}
```

The token payload includes the expiration date of the cookie (e.g., 2 hours) and the user ID, which allows us to identify the owner of the cookie. This enables us to retrieve only the user's services when interacting with other services.

Once the token is created, it needs to be signed using our private key. We utilize the HMAC-SHA256 cryptographic algorithm for generating the signature with the private key. After signing, we return the token. Other microservices only require the public key to validate tokens and authorize requests.

Public Key Sharing When it comes to sharing the public key so that other microservices can use it, several techniques come to mind:

1. Certification Authority (CA): This approach ensures that the certificate is owned by the microservice, providing a level of trust. However, frequent key rotation can be cumbersome due to the associated overhead. CAs are typically used for SSL certificates used in establishing HTTPS connections between services.
2. Manual Certificate Sharing: This involves manually copying the public key certificate to the codebase of other microservices. However, this approach presents several challenges:
    - Key Rotation: It becomes difficult to rotate keys, as the code always references the key file. If rotation is possible, it may require server downtime.
    - Access Security: Securing access to the key is essential. While anyone can view the key, preventing unauthorized individuals from replacing it with another key is crucial to avoid key forgery.
    - Scalability: Copying and pasting keys into multiple repositories can be time-consuming, especially with a large number of microservices.

    However, for small repositories or non-production versions, this manual approach may be acceptable.
3. Public Key Retrieval Route: Another approach is to store the public keys in

a dedicated API route, such as /keys, which returns the public keys. Microservices can include a 'key' tag in the token header to indicate which key to use for signature verification. This approach offers several advantages:

- Key Rotation: Rotating keys becomes seamless by discontinuing the use of a specific key (e.g., Key A) for signing new tokens and switching to other keys. Once the expiration period for the last token signed with Key A has passed, Key A can be discarded without downtime. Microservices do not need to be aware of specific keys during startup.
- Traffic and Scalability: Although this approach introduces additional traffic due to periodic requests for updated keys, it eliminates the need for microservices to individually store and manage keys. Microservices can request the latest keys as needed, ensuring access to the most up-to-date keys.

By adopting the public key retrieval route approach, key management becomes more flexible, allowing key rotation without disruption. However, it is important to consider potential network traffic increase and ensure that microservices periodically update their key caches to maintain current key information. In our application, we have chosen the Manual Certificate approach as we only have one microservice utilizing the tokens.

3.2.4 Detailed Documentation

Effective documentation serves as the foundation of any RESTful API. It delineates essential details such as the format, type, and the names of routes, thereby facilitating a smoother usage experience.

As discussed in Section 2.5, we utilize Swagger, a renowned software tool, to document our REST APIs. Swagger simplifies the API design process by enabling developers to build, design, document and consume RESTful web services.

In our microservice, we employ the swaggo/http-swagger package in Go, a statically typed, compiled programming language designed at Google. This package simplifies the documentation of our APIs, making the process more streamlined and efficient. To generate the routes, we create them on a specific path with the following line of code:

```
router.PathPrefix("/doc").Handler(httpSwagger.WrapHandler)
```

This code leads to the creation of API documentation accessible through the /doc/index.html page.

Figure A.11 gives a snapshot of what our Swagger documentation looks like.

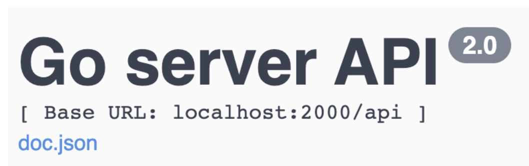

Figure 3.11: The Swagger hostname with the base path visible. It starts with `localhost:2000/api`, implying that any subsequent route (e.g., user) should follow the `/api` prefix, resulting in a complete path like `/api/user`.

The API we're working with is rather simple, consisting of just two routes. Before diving into a detailed explanation, it's imperative to note that all the API routes commence with the prefix `/api/`, as explicitly displayed in this portion of the Swagger documentation.

Now, let's delve into the specific routes:

1. `sign-in`: This route is designed to authenticate a user by signing them in. The documentation provides the following description:

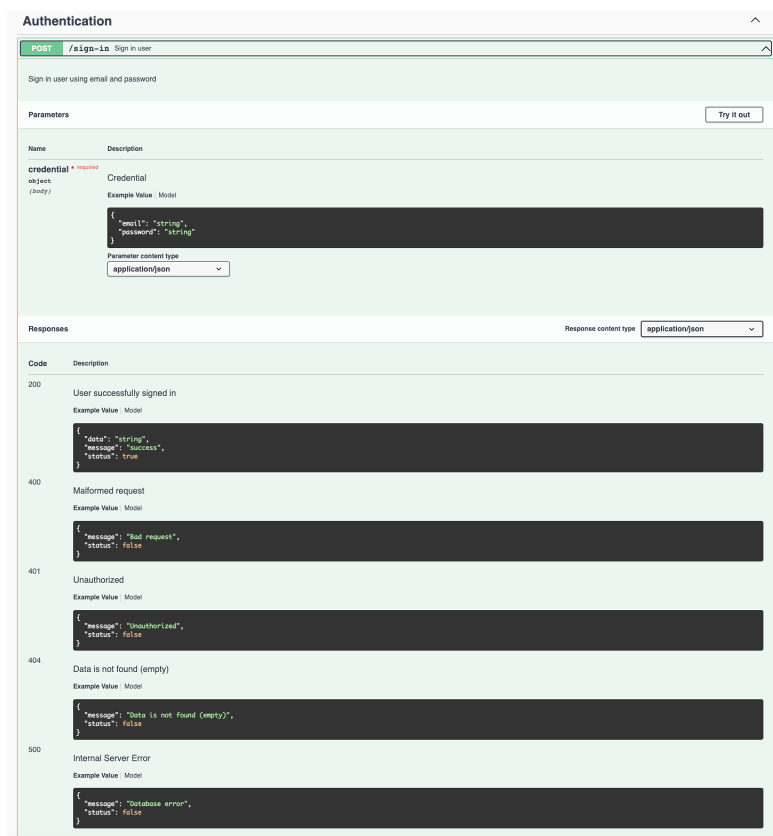

Figure 3.12: Sign-in route depiction: We can see that it utilizes a POST method. The request's body is expected to contain two key values (i.e., email and password). The response will be a code that varies depending on the parameters received and processing. The format returned is JSON, with either two or three fields. Upon a successful sign-in, the 'data' field contains the result (a string in this case). Additionally, the response includes a 'message' and a 'status' to provide comprehensive information about the response.

2. The `sign-up` route is designed to facilitate user registration using their email and password.

3.2.5 Tests

Testing our service is a crucial task, although it is not complex as we only have two routes. The tests for these routes include:

Sign-in

- When valid credentials are provided, we expect a valid code and token with the correct expiration data.
- If an incorrect email is provided, we expect a non-200 response code.
- If an incorrect password is provided, we expect a non-200 response code.
- If the request format is incorrect, we expect a non-200 response code.
- ...

Sign-up

- When valid credentials are provided, we expect the user to be created successfully.
- If a non-email address is provided, we expect a non-200 response code.
- If no password is provided, we expect a non-200 response code.
- ...

Please note that these lists are not exhaustive, and determining the number and depth of tests can be a complex task, especially when considering aspects like testing against SQL injections.

In Go, tests are performed using the *testing* package, which allows us to define tests that break if specific conditions are not met.

Here is an example test:

```
1 type SignInRequest struct {
2         Email    string `json:"email"`
3         Password string `json:"password"`
4 }
5
6 func TestSignInRightPassword(t *testing.T) {
7   // Creating the structure of the sign-in request as
            specified in the Swagger documentation
8   // enables us to easily forge requests.
```

```
9	  _, err := models.CreateUser(test_email,
	     test_password)
10	  assert.NoError(t, err)
11	
12	  // Marshal the request
13	  req := SignInRequest{
14	    Email:    test_email,
15	    Password: test_password,
16	  }
17	  body, err := json.Marshal(req)
18	  assert.NoError(t, err)
19	
20	  // Send the request
21	  resp, err := http.Post(url+"/sign-in", "application/
	     json",   bytes.NewBuffer(body))
22	  assert.NoError(t, err)
23	
24	  // Verify that the response code is 200
25	  assert.Equal(t, 200, resp.StatusCode)
26	}
```

This test code is explained as follows:

Starting from lines 1-4, we create the structure of the sign-in request as specified in the Swagger documentation, which allows us to forge requests easily. In the test function, we first create a user with the specified credentials on line 9, and we check if the user creation succeeded on line 10. Then, we forge the request using the *json* package until line 18. After that, we send the request and verify that the response code is indeed 200 on line 25.

3.3 Unravelling the Main Gateway

The Main Gateway stands as an integral service, primarily tasked with mediating interactions between the system and the APIs of CSPs. It is designed to fetch and store the services available from these CSPs, ensuring their configurations are readily accessible for the frontend components of the system.

To understand the role of the Main Gateway more clearly, let's dissect the way it processes an incoming request. Its handling methodology is broken down into six distinct steps:

1. Verification of the validity of the included cookie via a middleware approach.
2. Extraction of relevant user details, such as the user's ID, from the validated cookie.
3. Parsing the request for essential information, often encapsulated in the form of query parameters.
4. Depending on the route, a determination is made: if the route's function involves fetching services from a CSP, the user-associated keys are retrieved to facilitate the requests. In contrast, if the function involves returning certain structures from the database, no key is necessary and the request is passed directly to the controller.
5. The model is then solicited to procure the relevant information.
6. Finally, the gathered data is formatted as per the specifications outlined in the documentation and returned.

Each of these steps plays a vital part in fulfilling the duties of the Main Gateway. For a more comprehensive understanding, let's delve deeper into these steps in the following sections.

3.3.1 Authentication Via Cookie Verification

As outlined in Section 3.2, our system employs JSON Web Tokens (JWT) to handle user authentication. For the sake of bolstering security and mitigating the risk of Cross-Site Scripting (XSS) attacks, we store these JWT tokens within cookies. Each time a request is received, it undergoes a validation check to ensure the accompanying cookie is legitimate.

Middleware in Go The Go `mux` router is equipped with middleware functionality. This feature allows for a piece of code to be executed between the initial request and the specific code tasked with handling that request. In our setup, every request we receive is evaluated for the presence of valid tokens. If validation is successful, control is passed to the next handler for processing the route (usually this is the route's controller, but it could also be a logger middleware, for instance).

The middleware process is visually represented in the flow diagram shown in Figure 3.13.

It's important to note that some routes may bypass the cookie verification step. This typically includes routes that are frequently accessed and not associated with

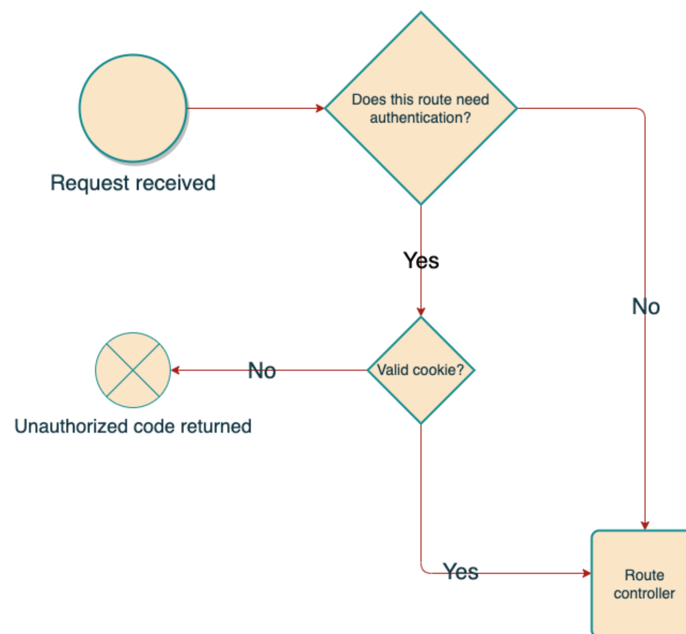

Figure 3.13: Flow diagram depicting request handling in the main gateway

user data, such as documentation routes (although we utilize a separate router for this, if we weren't, the same principle would apply). Another example would be OPTIONS routes which are commonly sent and do not carry a cookie.

Validating the Cookie As mentioned earlier, we employ the Manual Approach for key sharing, meaning the key file is securely stored in the repository. For efficiency, this key is loaded into a variable at the start of the service, which expedites subsequent key verifications. When a request is received, the included cookie undergoes a rigorous validation process consisting of two primary steps:

1. Firstly, we ascertain that the signing method isn't "None". This is a vital security measure as if it were set to "None", malicious parties could potentially forge cookies at will (as discussed in Paragraph 2.3.2).

2. Secondly, we leverage a library function to confirm the validity of the token. This function calculates the signature of both the header and body using the key and specified algorithm, then returns the claims, which contain the body of the token.

If the validation process identifies the cookie as invalid, a 401 status code (Unauthorized) is returned, in accordance with the HTTP status code specifications found here. If, however, the cookie is validated successfully, the system proceeds to serve the next layer by executing:

```
next.ServeHTTP(w, r)
```

In our context, this 'next layer' refers to the controller.

3.3.2 Extracting Relevant Information to Handle the Request

Four types of information can be needed:

1. **Token information**: These are the information stored in the JWT token, such as the user ID. As the body is just base64 encoded of the actual set of information, we just have to decode it and we can use its content.
2. **Path parameters**: These are parameters in the path, essentially used to get some specific thing of a specific thing. For instance, if we want to get the route table of a VPC, the route is going to be: `/vpc/{vpc-id}/route-table` and it will return the route table of the VPC. To get the VPC ID here, we are using the mux router. In the route, we specify the parameter using the curly brace notation and then in the controller, we can just do:

```
vpcId := mux.Vars(r)["vpc-id"]
```

3. **Query parameters**: As the RFC specifies that the body of a GET request should not influence the content returned, we have to use query parameters. For instance, if we want to get the VPCs of a specific region, we have to use the route with the **region** parameter: `/vpcs?region=us-west-2`. To fetch the element, it is as easy as doing this:

```
region := r.URL.Query().Get("region")
```

4. **Body parameters**: In a POST request, the body is important. For instance, to perform reachability testing as will be explained later, we pass a body to the route. To use this body, we have to "unmarshal" it. First, we create a structure representing the body and then unmarshal it. This has the benefit of also checking the syntax and ensuring that all the required parameters are present. In this service, we do not have a POST request, but we do create POST requests toward the reachability service. Here's an example when having a body for the reachability testing:

```
type HopStatus struct {
    Id          string `json:"id"`
    CanTransmit bool   `json:"canTransmit"`
    Reason      string `json:"reason"`
}
```

```
type Response struct {
    Reachable bool        `json:"reachable"`
    Reason    string      `json:"reason"`
    Path      []HopStatus `json:"path"`
}

var response Response
err = json.NewDecoder(resp.Body).Decode(&response)
if err != nil {
    u.Respond(http.StatusInternalServerError, w,
    u.Message(false, "Error decoding response"))
    return
}
```

The reachability service returns a list of `HopStatus` (we will see it later). The "Decode" operation, which is the equivalent of the Unmarshal operation, decodes the response body and stores it in the previously created `response` object. After extracting the relevant information, we can proceed to process it using the obtained objects or values.

3.3.3 Fetching and Returning the Data

Having verified the legitimacy of the request and obtained all the necessary information for processing, we can proceed to fetch the services.

There are two types of routes: routes that directly fetch services from the CSP's API, used to retrieve the currently deployed topology, and routes to fetch services stored from a previous topology.

Online Topology By default, all paths use the online infrastructure unless an *infrastructureId* is specified. For example, to retrieve the paths of the VPCs currently online, the route path would be: `/vpcs`.

We utilize the AWS Go SDK v2 to communicate with the API. In order to establish communication, we need to fetch the services and create an AWS configuration. This **config** stores information such as API keys and the region for the query.

The data is then returned in the format specified in our API documentation, which may differ from the format used by AWS as we may not require all the fields stored by AWS. To address this, we have created multiple methods to convert AWS-like

Figure 3.14: Main Gateway database structure

structures to our own structures. For example, `CreateTagsFromAwsTags` takes a list of `AwsTags` as input and returns our `Tags` structure.

Once the data has been fetched, we simply return it to the user.

Offline Topology Offline routes can be identified by the `saved` suffix in their path, followed by the *infrastructureId*. For instance, the following route is responsible for returning the VPCs saved in the infrastructure with the given *infrastructureId*: `/vpcs/saved/{infrastructureId}`.

To retrieve this data, we must be able to save the infrastructures. Another route is responsible for this task, it will populate the tables shown at Figure 3.14

Infrastructures This table maps the infrastructure ID to a profile. The profile table is used to store information such as the name and description of the saved

infrastructure. We utilize this infrastructure ID in all subsequent requests. Additionally, the infrastructure ID is stored in all other tables to indicate from which infrastructure the service has been saved.

Apart from that, the tables are fairly straightforward.

Sample Data To facilitate the development process, fake (sample) data is utilized. This data is stored in an SQL file, which, upon startup, inserts all the sample data into the database. This allows us to easily reset the state of the database in case of any bugs or issues.

3.3.4 Testing Procedures

We conduct various tests to verify the acceptance of valid tokens, rejection of non-valid tokens, and evaluate the behavior of our services when integrated with a valid infrastructure.

Token Validation Tests To validate the handling of tokens, we perform GET requests on test routes that require authentication. Our middleware is invoked during these tests to verify the authenticity of the token provided. The following scenarios are tested:

- No Token: We send a GET request without including a token in the request header. The middleware should reject the request and return an authentication error.
- Invalid Token: We send a GET request with an incorrect or expired token. The middleware should detect the invalid token and reject the request, returning an authentication error.
- Valid Token: We send a GET request with a valid token. The middleware should successfully authenticate the token and allow access to the requested route.

By conducting these tests, we ensure that our token verification mechanism functions correctly, allowing authorized access and denying unauthorized access to our services.

Testing Infrastructure In our sample file, we've incorporated an infrastructure that is initiated during the database startup. This infrastructure is leveraged to verify if the routes intended to fetch the Virtual Private Clouds (VPCs) associated with a given `infrastructureId` indeed retrieve all of them. This typically

involves reviewing the number of services obtained, a process we apply across all service types.

Testing live infrastructure, on the other hand, using CSP's API, presents a more complex challenge since the number of current VPCs in the cloud is not really known. A potential workaround could involve using a **Terraform** script that deploys an infrastructure and subsequently verifies whether the routes are correctly returning the intended services. Nevertheless, I propose this task should be integrated into a Continuous Integration (CI) pipeline due to its larger-scale testing requirements.

Continuous Testing with Pre-commit Hooks To maintain a high level of code quality, we employ pre-commit hooks that automatically execute tests on each code commit. This ensures that any changes or additions to the codebase undergo thorough validation before being merged into the main branch. By incorporating continuous testing practices, we mitigate the risk of introducing potential bugs or vulnerabilities into our services.

3.3.5 Throttling of APIs by CSPs

When fetching services from CSPs, there are limits on the number of requests that can be made concurrently. This limitation limits our ability to parallelize requests, resulting in noticeable consequences when visualizing the online infrastructure. Loading the infrastructure takes longer due to the inability to parallelize requests. However, when working with saved infrastructures, no delays are observed.

3.4 Exploring the Reachability Server

The Reachability Server is a key component of our application, serving as the mechanism for conducting reachability testing. Our goal is designing an algorithm that can theoretically gauge the reachability between different services.

The algorithm should be capable of traversing from one hop to the next, querying the service at each stage to determine if, from a configuration standpoint, the subsequent hop is accessible. The algorithm will be explained using the basic topology depicted in Figure 3.15.

Our initial focus will be on understanding the workings of the service. Following this, we will delve into the specifics of the algorithm.

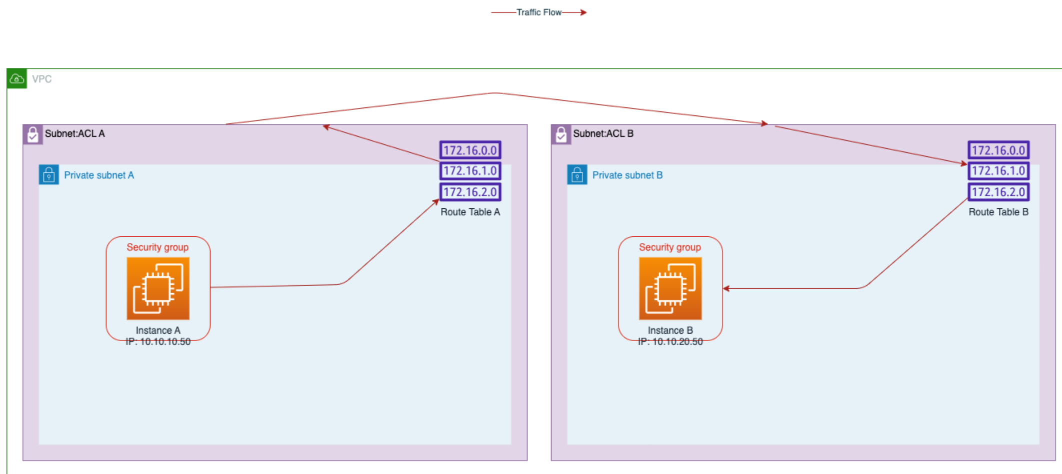

Figure 3.15: This basic topology presents a Vpc, which encompasses two subnets. Every subnet is linked with an ACL and a route table, and each one houses an EC2 instance secured by a security group. The red arrow illustrates the route from instance A (`10.10.10.50`) to instance B (`10.10.20.50`).

3.4.1 Service Technology Considerations

Contrary to our earlier services, this particular service necessitates a focus on algorithmic design. A sophisticated way to implement these algorithms is by employing the Object-Oriented programming paradigm, such as polymorphism (for example, polymorphism allows objects of different types to be treated as objects of a parent type, offering a way to simplify complex problems). Unfortunately, the Go programming language lacks full-fledged object-oriented structures. It only supports structures and methods within these structures and permits a degree of hierarchy extension, but this is not straightforward or comprehensive.

Fortunately, our decision to employ a microservices architecture offers us flexibility. One of the significant advantages of this architecture is that each microservice can utilize its own language and follow its own operational methods as long as it presents an accessible API.

In this context, we have chosen to use the **Python** programming language for the reachability testing. Python was selected for its expressiveness, extensive range of available libraries, and its robust support for the object-oriented approach.

Python's Web Framework Utilizing a web framework to process the incoming requests to our Python service is a necessity. There is a multitude of such requests, with perhaps the most widely known being *Flask*. However, for the purpose of this project, the selection of **FastAPI** has been made. This is due to its status as one of

the quickest web frameworks in Python, and the similarity it bears to the workings of Flask.

In FastAPI, we employ `uvicorn` to activate the web server. Subsequently, a route can be established in the following manner:

```python
app = FastAPI()


class ReachabilityQuery(BaseModel):
    source: dict
    destination: dict
    sgSource: list
    sgDestination: list
    protocol: str
    port: int
    ttl: int


@app.get('/reachability')
def reachability(rawReachability : ReachabilityQuery,
     authorization: str = Header(None)):
```

As can be observed in line 14, the path for the route from the app object (originally created in line 1) is being defined. We instruct that upon encountering the path `/reachability`, the function `reachability` should be invoked. This function has arguments, which are "typed" because FastAPI generates automatic documentation at launch. This ensures that the documentation correctly specifies the type. In our scenario, there are two arguments: the reachability structure and the authorization token.

Both of these elements will be clarified in due course.

Automated format verification is carried out, and any requests lacking the authorization header (or the reachability for that matter) are halted and a status indicating a malformed request is returned.

3.4.2 High level view of the Algorithm

Consider the provided pseudocode, which effectively illustrates the core idea of our algorithm (Backes & et al., 2019):

While this captures the overall workings of the algorithm, it's essential to consider potential edge cases.

Algorithm 1 Reachability Algorithm

Require: $sourceService, destinationService, token, reachabilityInformation, ttl$

1: $currentPath \leftarrow []$
2: $currentService \leftarrow sourceService$
3: **while** $currentService \neq destinationService$ **and** $ttl > 0$ **do**
4: $idNextHop \leftarrow$ currentService.NextHop$(destination = destinationService, currentPath, reachabilityInformation)$
5: $currentService \leftarrow$ fetchService$(idNextHop, token)$
6: $currentPath$.append$(currentService)$
7: $ttl \leftarrow ttl - 1$
8: **end while**

Broadly, the algorithm is straightforward:

1. We initiate with a source and a destination service, setting the source service as the current service. Every routing service is equipped with a 'NextHop' method; therefore, we call this method for the service, which subsequently returns the ID of the next hop.
2. We then fetch the service using this ID.

Each of these steps will be elaborated on in the subsequent subsections.

3.4.3 Determining the Next Hop

The determination of the next hop is an important task within our algorithm. Designing the appropriate model is crucial to facilitate the subsequent expansion for new services. The UML employed in this microservice is presented in Figure 3.16.

Before delving into the operations of each service concerning the next hop procedure, it's worthwhile to acquaint ourselves with the structure of the UML. There exists a general, abstract class known as `Service`, which encapsulates an ID. Indeed, each service is associated with a unique identifier. We then bifurcate services into two main types: the `RoutingService` and the `BlockingService`. Below are their descriptions:

- A `RoutingService` features a 'nextHop' method, tasked with determining the next hop. 'RoutingService' does not refer to a physical entity, but rather symbolizes the idea of routing. These services essentially steer the packet towards its next stop. For instance, a Subnet qualifies as a routing ser-

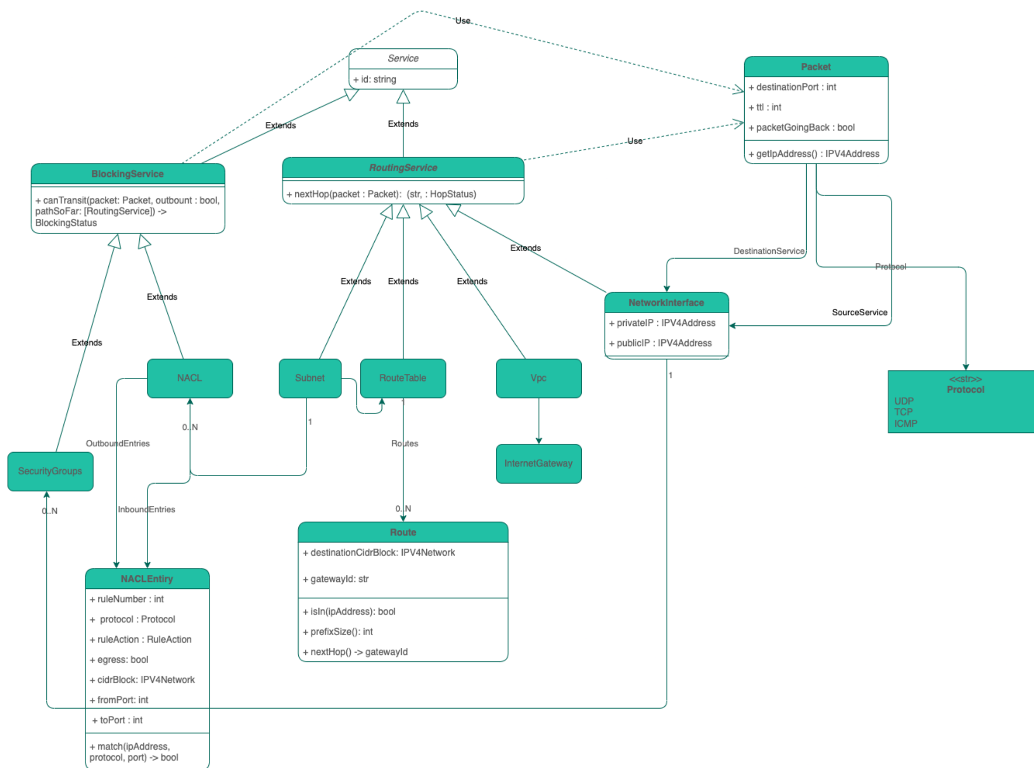

Figure 3.16: A portion of the UML utilized by the reachability service to execute the reachability algorithm. This primarily focuses on the routing aspect, with any redundant information omitted, which is why most classes don't display any fields/methods.

vice. While a subnet is purely an abstract concept and there isn't a physical 'subnet' that manages a packet, exiting a subnet necessitates going through certain services. Thus, we classify Subnet as a routing service. Moreover, Routing services are entrusted with the duty to inspect their blocking services to confirm if the traffic is authorized, thereafter furnishing the ID of the next hop.

- The `BlockingService` embeds a 'canTransit' method. This method scrutinizes the packet along with some other information and yields a `BlockingStatus` (though not depicted in the figure, it records whether it's been blocked and the reason behind it). Services that inherit from this abstract class comprise firewalls, access control lists, policies, among others.

Next, let's dive deeper into each service and discern how they implement the `nextHop` or `canTransit` methods uniquely. Note that the pseudo code for each service's algorithm is provided either in the text if it is relevant or in Section A.

EC2 Instance and Network Interface A crucial point to understand is that EC2 instances, and generally virtual machines in the cloud, don't possess network connectivity until they're connected to a network interface.

Consequently, the concept of an EC2 instance (i.e., VM) here is irrelevant, and we will focus exclusively on the network interface.

The Network Interface serves as a routing service, implying it needs to determine the next hop. Each network interface is tied to one or more instance-level firewalls (for example, security groups).

The algorithm to determine if the security group permits the packet to transit will be discussed in the next paragraph. If it does, we examine whether the packet is directed towards our network interface (essentially, we verify if the traffic is inbound). If it is, we return our ID to indicate that we've arrived. On the other hand, if the packet is not destined for our interface, we return the subnet ID because it signifies that we're exiting the subnet.

Security Group A security group functions as the firewall for EC2 instances, thereby classifying it as a blocking service. Security Groups are stateful entities. They utilize the list of services already encountered along the path. If our own ID appears in the list of services, we understand that the security group has already approved the traffic for entry or exit, so we can authorize the traffic to enter or leave. However, the actual process isn't quite as simple.

Algorithm 2 Network Interface Next Hop

Require: securityGroup, packet
Ensure: blockStatus or subnet_id

```
1: blockStatus ← securityGroup.canTransit(packet)
2: if blockStatus = blocked then
3:     return blockStatus
4: else if packet.get_destination_ip() ≠ self.ip then
5:     return subnet_id
6: else
7:     return self.id
8: end if
```

Consider the topology depicted in Figure 3.15 for instance. In AWS, both instance A and instance B might be protected by the same security group, perhaps a security group that allows outbound SSH but no inbound traffic. When A attempts to SSH into B, the security group permits outbound traffic as per its rules, and adds its ID to the witnessed services list. But when the packet reaches B, the security group sees its own ID in the list of visited services and assumes that it has already accepted the packet, thus allowing SSH inbound traffic, even though it should have blocked it as no rules permit it. Addressing this issue equates to solving the uniqueness problem of the ID added to the list of services. As a workaround, we include a tuple of values in the list of services. This tuple comprises both the ID of the security group and the ID of the associated network interface. As one network interface is linked to only one instance, it is a unique value and thus ensures uniqueness (note that it's the network interface that calls the security group, so it's straightforward to verify it before calling).

Having addressed the statefulness problem, the remainder of the process is relatively straightforward. We just need to sift through the inbound or outbound rules of the security group and whenever we find a match for a rule, we accept the packet. If no rule matches, we "block" it. Keep in mind that security groups only have allow rules, with a deny default at the end.

Subnet The Subnet is a routing service associated with a NACL. Similar to the EC2 instance, it checks whether the NACL allows the traffic to transit. If the traffic is allowed, it returns the ID of the route table.

NACL The NACL is a blocking service known as the Subnet layer firewall and operates in a stateless manner, making it simpler compared to security groups. Unlike security groups, NACLs have both allow and deny rules. The algorithm

is straightforward: it iterates over the rules ordered by their number, and the first rule that matches the packet determines its action (Allow/Block).

Route Table Routing plays a vital role in achieving reachability. The route table serves as a routing service responsible for specifying the next service that will handle the packet. In cloud service providers (CSPs), routing is more complex than traditional IP routers, as the output can be various services or identifiers such as the word "local" for traffic meant to remain within the VPC.

The algorithm for the route table is relatively simple: it performs a longest prefix match on the routes and returns the next hop associated with the winning rule. Throughout this process, we extensively utilize the ipaddress library, which provides convenient utilities for determining if an IP falls within a CIDR range.

3.4.4 Retrieving the Service Using Its Identifier

Once we've ascertained the identifier of the next hop, our next task is to retrieve its configuration. During the initial request, it isn't feasible to include the configurations for all existing services due to scalability issues in larger infrastructures, and the fact that many of these services might not even be employed.

Referring to Figure 3.17, the reachability service is accessible only through the Main Gateway. The Main Gateway, in turn, is the sole entity capable of accessing the configuration of services, either via the CSPs APIs or its own database. Hence, the reachability service must contact the Main Gateway to retrieve the service configuration using the identifier supplied by the nextHop method. However, the Reachability service lacks context regarding the infrastructure on which it is performing the reachability test (e.g., online, offline, infrastructure ID, etc.).

Incorporating the Infrastructure ID in the Request Naively One possible solution involves the Main Gateway passing an identifier to the reachability service during the initial request. This identifier is then included in all subsequent requests made to the Main Gateway by the reachability service. This process is depicted in Figure 3.18.

While this solution does work, it's important to remember that the reachability microservice operates as a REST API. It operates under a contract, whereby it receives certain inputs and provides a specific output. The output in this scenario is a list of services the packet has traversed. However, if a malicious actor were to interfere, problems might arise, as shown in Figure 3.19.

There are two primary issues with this approach:

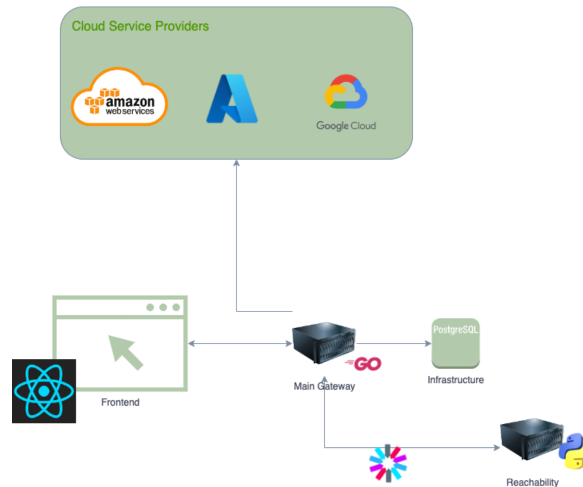

Figure 3.17: Architecture focusing on the reachability interaction

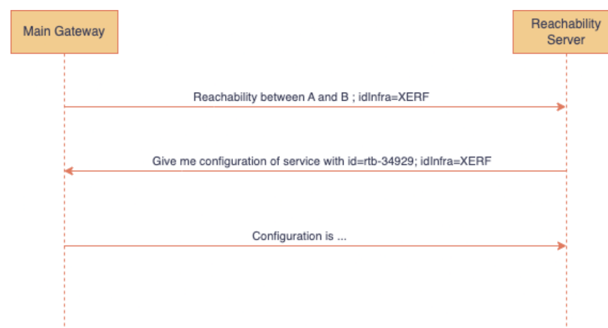

Figure 3.18: Traffic between the main gateway and the reachability service. The main gateway sends an id (actually it will be several id, e.g. id_user...) that the reachability microservices will then use to query subsequent services configurations.

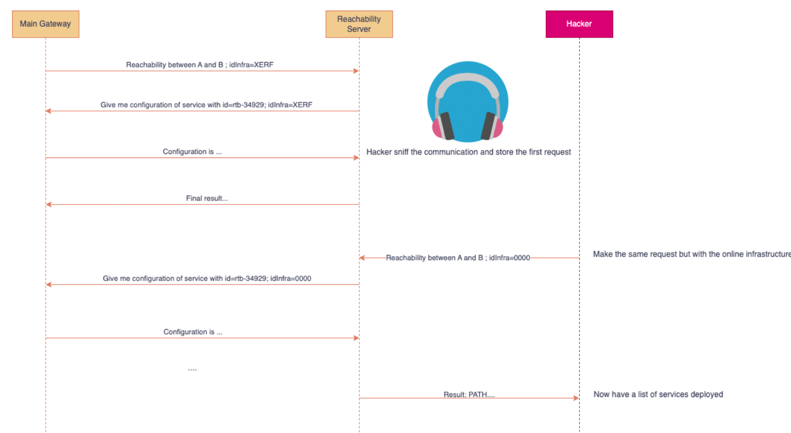

Figure 3.19: Potential security breach when using IDs as context for the reachability service to retrieve configurations

1. The Main Gateway sends configurations even when it is not the one performing the reachability testing.
2. The infrastructure ID can be changed freely.
3. Optional: The lack of encrypted traffic can be problematic. Even if the traffic was encrypted, thus preventing the hacker from intercepting requests, the hacker could still gain service IDs via other vulnerabilities (for instance, a frontend vulnerability) and write the most simplified configuration (firewall allowing everything). Then use the online infrastructure to avoid having to guess the ID.

These issues can be addressed by implementing JWT tokens. Here's how the new algorithm would work:

The Main Gateway generates a token containing the context information of the infrastructure used for the reachability test (e.g., infrastructure ID). This token is then signed using a symmetric algorithm, ensuring that the Main Gateway is the only entity capable of verifying the token's signature and creating tokens.

When the reachability service receives a request, the token is included, and the microservice utilizes this token when requesting a service configuration. This abstraction enables the reachability service to perform the test without knowing the infrastructure's context. Before returning the configurations, the Main Gateway verifies the token's signature and, if valid, sends it.

This approach effectively solves the two problems in the following ways:

- The token creation process ensures that a hacker cannot represent a user

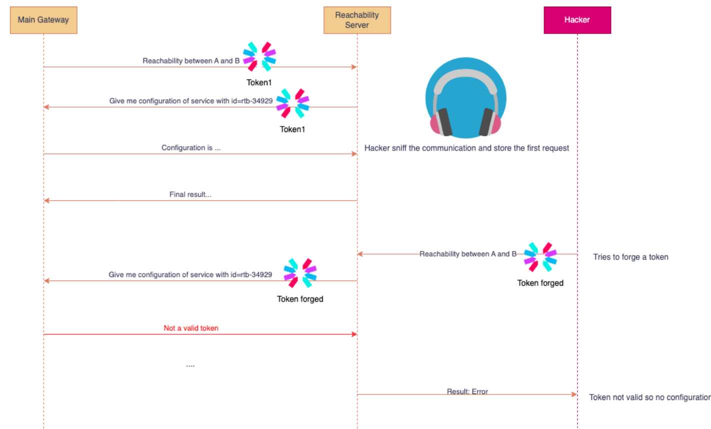

Figure 3.20: Token-based reachability query. The token carries context information, and the hacker cannot fabricate a token.

to obtain a configuration, as the Main Gateway will not send a configuration without a valid token. Also, tokens carry an expiration date, rendering stolen tokens useless after a certain period.

- Modifying the ID of the user or infrastructure is no longer feasible, as it would invalidate the signature.

The ultimate solution is illustrated in Figure 3.20.

3.4.5 Returned Values and Their Applications

The API endpoint delivers three distinct values as its response:

- **Reachability Status:** The first output indicates whether the two services in question are reachable. This binary status acts as a swift gauge of connectivity.
- **Reason for Unreachability:** If the services are not reachable, the API further elaborates on the cause of this unreachability. For instance, this could be due to the Time-To-Live (TTL) reaching zero, suggesting that the packet has traversed too many hops without reaching its destination, or due to a firewall blocking the traffic, indicating a network security measure at play.
- **Packet's Path:** Finally, the API illustrates the path taken by the packet, regardless of its success in reaching the destination. This information, depicted for each hop, includes the service's ID, whether or not the service forwarded the packet, and the reason behind its decision. For example, a specific rule might permit the packet's forwarding (e.g., "Rule 31 allows

it") or conversely, the packet might be blocked due to a lack of any rules permitting its passage (e.g., "No rule allowing it").

The frontend application could use this information to visually map out the packet's path on a network graph, enabling users to understand the packet's journey better. The returned values provide valuable insights, offering diverse possibilities for handling and interpreting the data according to the specific needs of the application.

3.4.6 Testing

The testing process involved initially setting up infrastructures using Terraform, followed by querying the reachability between two services. The infrastructure specifically delineates the ground truth, enabling us to evaluate various scenarios to see whether the service accurately returns the correct conclusion and path. It's crucial to note that since the service cannot be independently tested, those are conducted within the Main Gateway.

It's also important to mention that the algorithm doesn't cater to all scenarios and services. As of the release of this report, it functions only for internal VPC communications across subnets and for EC2 instances (i.e., network interfaces).

4 | Conclusion and Future Prospects

4.0.1 Looking Forward

As we move forward, the Cisco team is committed to enhancing this application by integrating more mainstream CSPs and adding additional AWS services. A minor refinement required is the adaptation of the frontend theme to mirror Cisco's service color scheme. Furthermore, when dealing with very large infrastructures, they will add filters to enable users to selectively view the relevant parts of the infrastructure.

Potential Future Developments The potential for further development of this application is vast and exciting. Here are a couple of prospective ideas:

- Introduce a feature that allows users to compare saved infrastructures and identify differences in configurations. This could help detect changes such as deleted routes in a route table.
- Implement a functionality that accepts Terraform projects as input to create an infrastructure. This could provide companies with a visual representation of what their deployment would look like before incurring costs associated with actual deployment.

The horizon for this application is expansive, considering its foundational dependence on rapidly evolving cloud technologies.

4.0.2 Conclusion

This project has successfully laid the foundation for an application capable of displaying and testing reachability across CSPs. Although currently only operational with AWS, the design philosophy has always been centered on ensuring seamless integration with other CSPs in the future.

The choice of technology, such as the microservices architecture, ensures efficient maintenance and further enhances the expandability of the application. This architecture also allows future projects to make use of the services without delving into their internal workings, and rely solely on their APIs.

The features deployed during this academic year have been thoroughly demonstrated with the help of screenshots provided in Appendix A. These illustrations offer a comprehensive view of the user interface and the various functionalities that have been integrated into the application.

This extensive development process, spanning across six separate Git repositories, was housed within the GitLab group. The following table (Table 4.1) and figure (Figure 4.1) summarize the effort invested across these repositories, reflecting the diverse technology stack employed and the substantial amount of work undertaken to realize this project.

Git Repository	Lines of Code	Lines of Documentation	Number of Files	Commits
Infrastructure code in Terraform	99	14	8	6
Study of the different CSP's SDK	748	203	6	32
Frontend web app	3369	176	75	65
Login Service	1108	151	21	11
Main Gateway	5473	1026	54	60
Reachability Service	606	232	26	9
Total	**11403**	**1802**	**190**	**183**

Table 4.1: Summary of work performed across Git repositories

Despite the significant progress made thus far, this project's vision extends beyond the current capabilities. The choices and decisions made throughout the development process were backed by a forward-thinking strategy, ensuring the application's compatibility with the integration of additional CSPs and services.

Ultimately, while the application is currently operational and boasts some key features, significant work still remains before it can be deployed publicly. Nevertheless, the robust foundation established in this project will undeniably facilitate its future expansions and success.

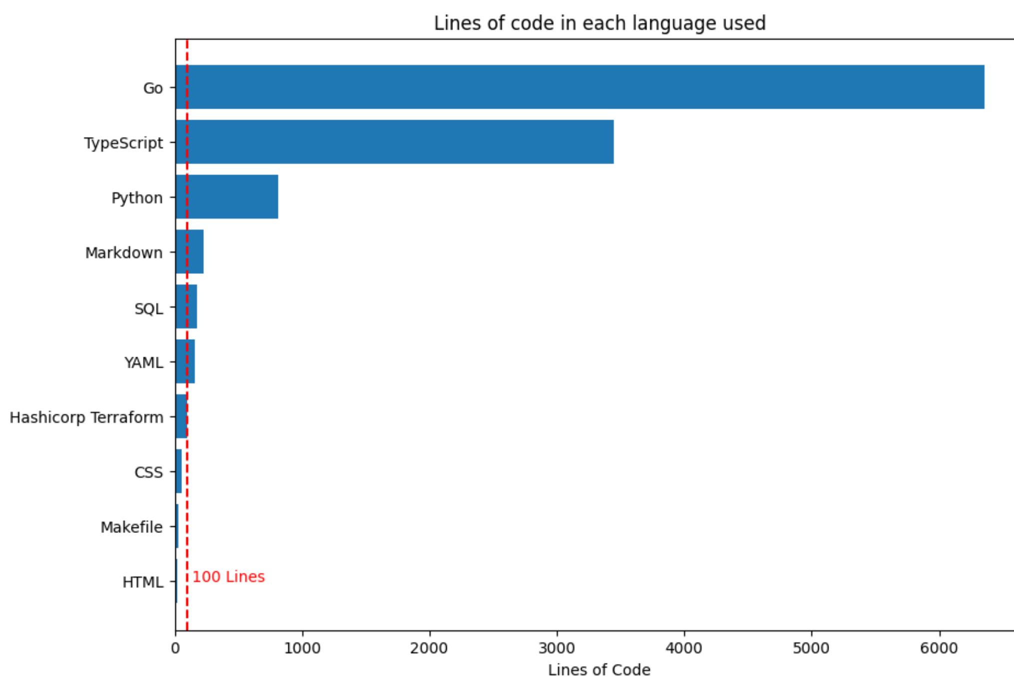

Figure 4.1: Stack of languages and for each the number of lines coded for this application

Bibliography

Ahmed, S., & Mahmood, Q. (2019). An authentication based scheme for applications using json web token. In *2019 22nd international multitopic conference (inmic)* (p. 1-6). doi: 10.1109/INMIC48123.2019.9022766

Artac, M., Borovssak, T., Di Nitto, E., Guerriero, M., & Tamburri, D. A. (2017). Devops: Introducing infrastructure-as-code. In *2017 ieee/acm 39th international conference on software engineering companion (icse-c)* (p. 497-498). doi: 10.1109/ICSE-C.2017.162

Backes, J., & et al. (2019). Reachability analysis for aws-based networks. In *Computer aided verification. cav 2019* (Vol. 11562). Springer. Retrieved from https://doi.org/10.1007/978-3-030-25543-5_14

Cheney, D. (2020, February). The zen of go. Retrieved 2023-06-09, from https://dave.cheney.net/2020/02/23/the-zen-of-go

Davis, A. (2022, October 5). *The pros and cons of a monolithic application vs. microservices.* Retrieved 2023-06-09, from https://www.openlegacy.com/blog/monolithic-application

Kubernetes. (2023). *Kubernetes concepts - overview.* Retrieved 2023-06-09, from https://kubernetes.io/docs/concepts/overview/

Musib, S. (2019, November 17). Restful api documentation made easy with swagger and openapi. *Medium.* Retrieved 2023-06-09, from https://medium.com/swlh/restful-api-documentation-made-easy-with-swagger-and-openapi-6df7f26dcad

Patil, K., & Javagal, S. D. (2022). React state management and side-effects â a review of hooks. *Yearbook of the Association for Computational Linguistics.*

Rahmatulloh, A., Gunawan, R., & Nursuwars, F. M. S. (n.d.). Performance comparison of signed algorithms on json web token. In *Iop conference series: Materials science and engineering* (Vol. 550). IOP Publishing Ltd. doi: 10.1088/1757-899X/550/1/012031

Sriramya, P., & Karthika, R. A. (2015, July). Providing password security by salted password hashing using bcrypt algorithm. *ARPN Journal of Engineering and Applied Sciences*, *10*(13).

Tarraf, S., Cesarini, A., & Hughes, D. (2021). *To the multi-cloud and beyond.* Accenture Oracle, Business Group. Retrieved from https://www.accenture.com/_acnmedia/PDF-157/Accenture-Multi-Cloud-and-Beyond.pdf

A | Appendix

A.0.1 UI of the web app

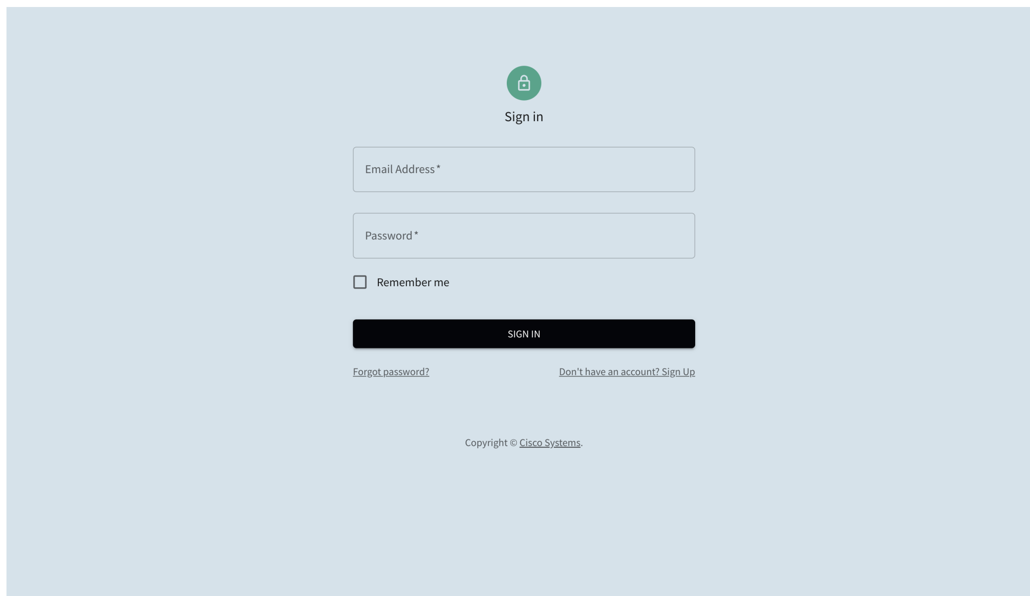

Figure A.1: User login interface. This interface initiates user login by forwarding requests to the authentication microservice and retrieving a token for further interactions.

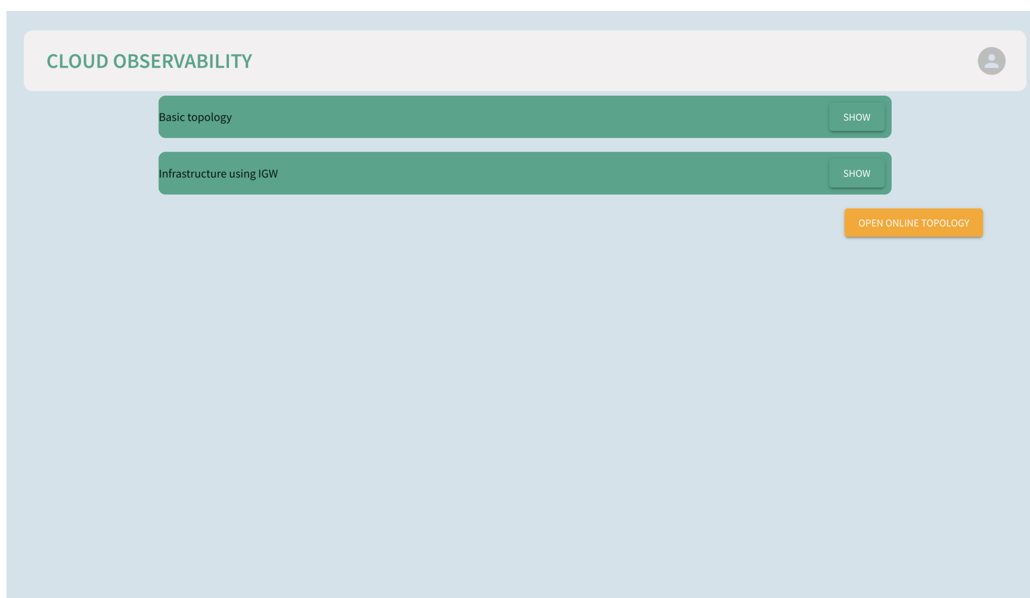

Figure A.2: Infrastructure selection interface displaying two saved infrastructures. Each infrastructure is identified by a name. The yellow button enables viewing the deployed online topology. A user settings option is available on the top right corner of the toolbar.

Figure A.3: Main page showcasing your infrastructure on the right, with visible details of three VPCs, including their CIDR range and name (if tagged). Subnets are shown along with their CIDR range and last digits of their ID (with an option to display their name). The left pane contains two tabs, for viewing clicked service details and performing reachability testing.

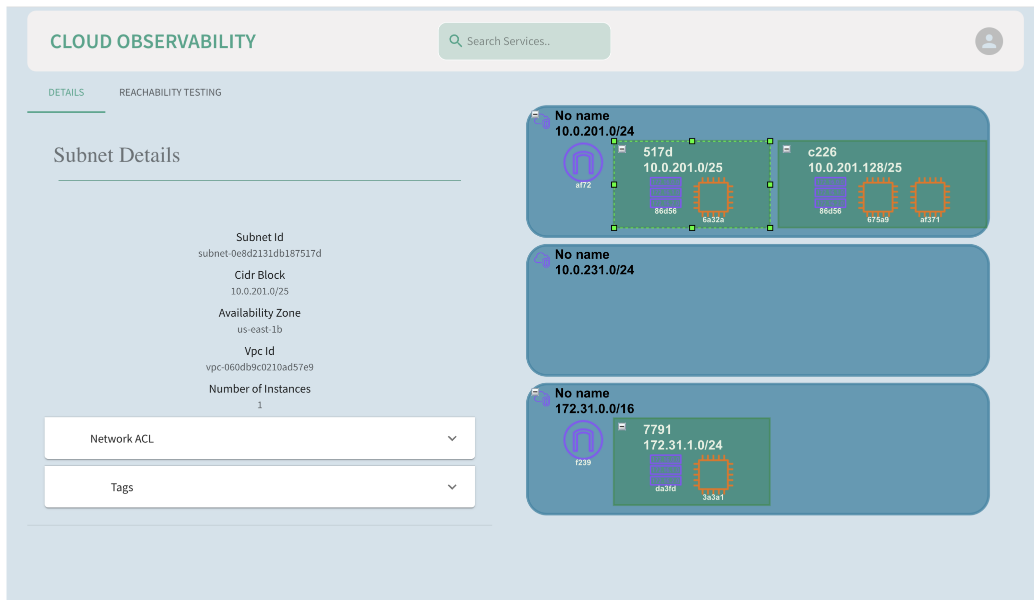

Figure A.4: Subnet detail view, displaying relevant service information upon clicking on a subnet. Top of the interface presents details such as the AZ, ID, etc., while the bottom segment lists tags and associated network ACL with inbound and outbound rules.

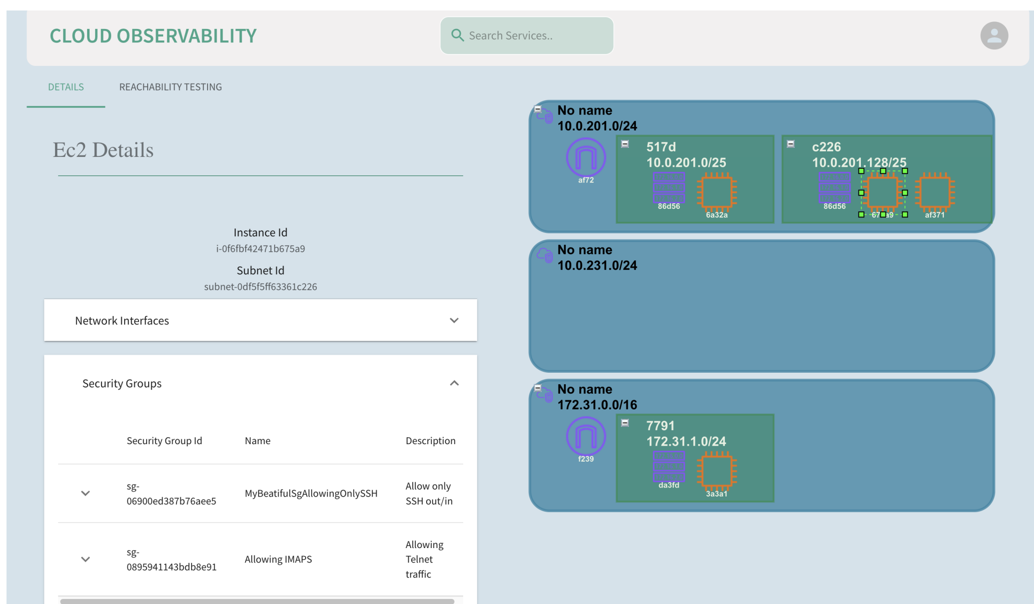

Figure A.5: Detailed view of an EC2 instance, displaying network interfaces and associated security groups along with their respective rules.

Figure A.6: Reachability test interface, where "source" and "destination" are selected by clicking the hand icon next to them. User then chooses the protocol and port, upon which a request is sent and the path is displayed. In this instance, a successful path is shown, with individual steps detailing why the traffic was forwarded.

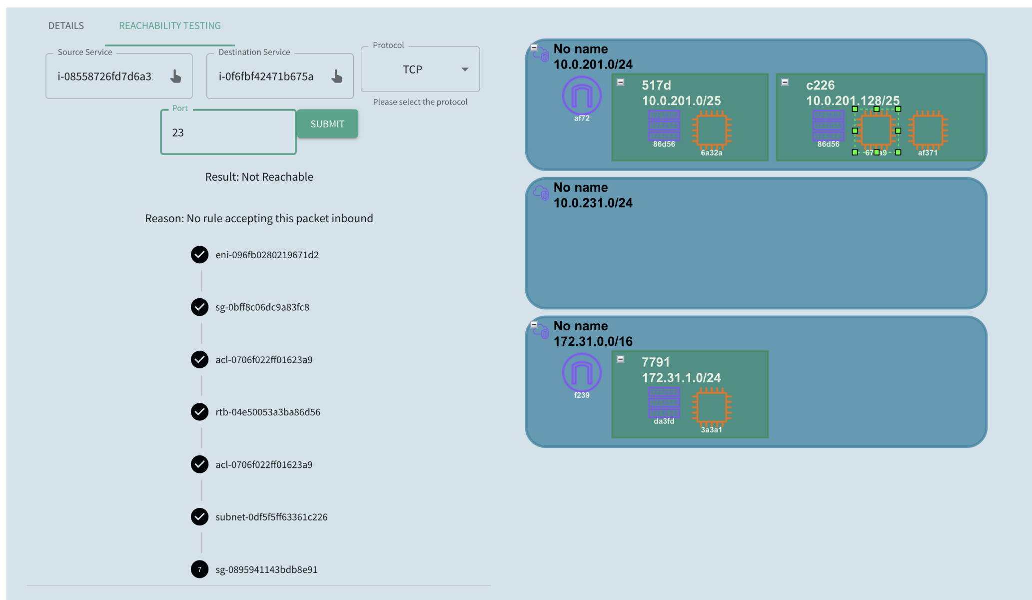

Figure A.7: Display of an unreachable path, showcasing the reason for unreachability and the final step without a check mark, indicating that the packet was not forwarded.

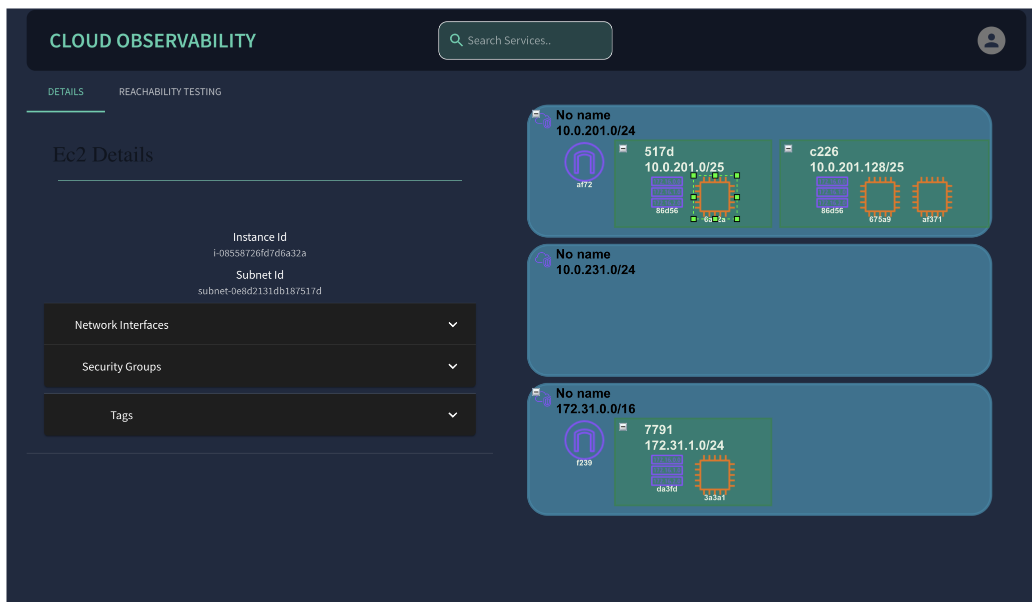

Figure A.8: Interface view in dark mode, activated by toggling the dark mode button in the settings. This mode adjusts all primary and secondary colors. The same procedure is used to apply Cisco colors.

Algorithm 3 Security Group Transit Permission

Require: securityGroup, packet, inbound
Ensure: blockStatus
1: **if** inbound = True **then**
2: $permissionArray \leftarrow$ securityGroup.inboundPermission
3: **else**
4: $permissionArray \leftarrow$ securityGroup.outboundPermission
5: **end if**
6: $blocked \leftarrow$ True
7: **for all** $element$ **in** $permissionArray$ **do**
8: **if** $element$.match(packet) = True **then**
9: $blocked \leftarrow$ False
10: **break**
11: **end if**
12: **end for**
13: $blockStatus \leftarrow$ BlockStatus(blocked)
14: **return** $blockStatus$

Algorithm 4 Nacl Can Transit Algorithm

Require: nacl, packet, egress
Ensure: blockStatus

1: **if** egress = True **then**
2: $ruleArray \leftarrow$ nacl.egressRules
3: **else**
4: $ruleArray \leftarrow$ nacl.ingressRules
5: **end if**
6: $blocked \leftarrow$ False
7: Sort $ruleArray$ by their number
8: **for all** $rule$ **in** $ruleArray$ **do**
9: **if** $rule$.match(packet) = True **then**
10: $blocked \leftarrow rule.Action = "Block"$
11: **break**
12: **end if**
13: **end for**
14: $blockStatus \leftarrow$ BlockStatus(blocked)
15: **return** $blockStatus$

Algorithm 5 Route Table Algorithm

Require: packet, routeTable
Ensure: nextHop

1: $nextHop \leftarrow$ null
2: $routes \leftarrow$ routeTable.routes
3: Sort $routes$ by their prefix length in descending order
4: **for all** $route$ **in** $routes$ **do**
5: **if** $route$.prefix.contains(packet.destinationIP) = True **then**
6: $nextHop \leftarrow route$.nextHop
7: **break**
8: **end if**
9: **end for**
10: **return** $nextHop$

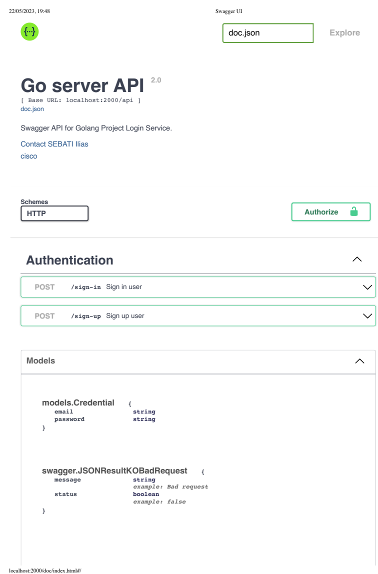

Figure A.9: Page 2 (showcasing the routes)

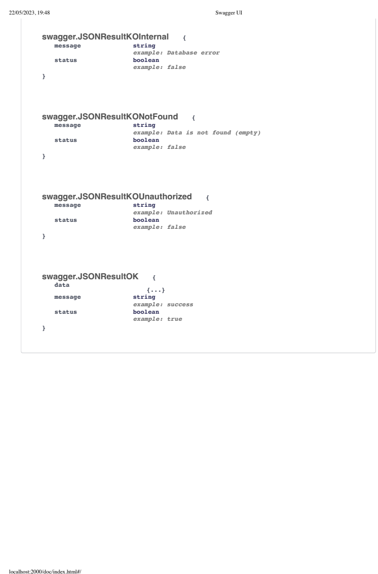

Figure A.10: Page 2 (showcasing the models)

Figure A.11: PDF format of the interactive Swagger documentation typically viewed in the browser