Technical Report

Individual Project

Course:	ES - Engenharia de Software
Date:	Aveiro, 13-12-2024
Student:	108287: Miguel Belchior Figueiredo

Table of contents:

1 Introduction

This report outlines the development process and results of an individual project undertaken as part of the Software Engineering course curriculum. The project's goal was to implement an agile workflow to design, develop, and deploy a fully functional web-based To-Do List application hosted on Amazon Web Services (AWS).

2 Agile Workflow

Epics and User Stories

In order to implement the solution, an agile methodology was followed. The project was carried out over four different sprints. During each sprint, the focus was on delivering specific functionalities from the product backlog, ensuring that each increment added value to the overall solution. Three different agile epics were identified: **User Authentication**, **Task Management** and **Deployment**. Each of these epics represents a body of work that was broken down into smaller and specific tasks (user stories) related to user authentication, task management and deployment, respectively.

The user stories, which are informal, general explanations of a software feature written from the perspective of the end user, were structured in the following format: **“As a [user], I want to [action], so that [benefit].”**. Each user story was accompanied by acceptance criteria defined in the Gherkin Syntax - **“Given [context], When [action], Then [result]”** - which will later be discussed in the **Definition of Done** section of this report.

Epic: User Authentication

The following user stories were included in the **User Authentication Epic**: User Registration, User Login and Landing Page. As an example, the **User Login** user story is highlighted below. Bear in mind the use of acceptance criteria as well as story point estimates and priority assignment. These aspects will be further discussed in the report.

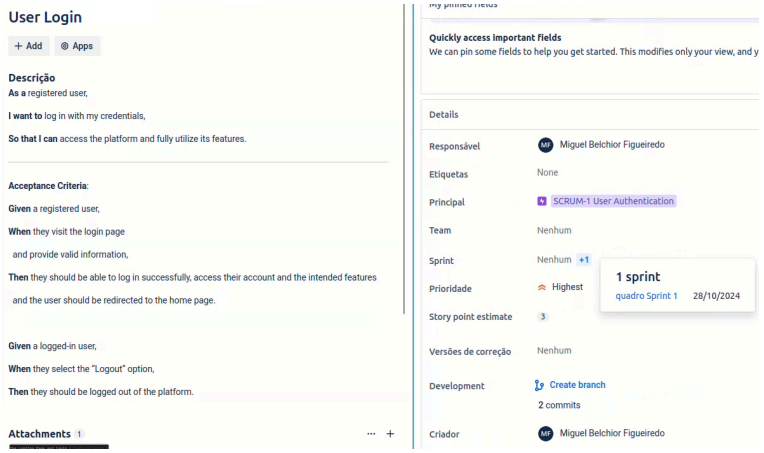


Figure 1 – User Login User Story Definition

Epic: Task Management

The following user stories were included in the **Task Management Epic**: Add New Task, Edit Task, Delete Task, Sort Tasks, Switch Task Status, View Tasks and Filter Tasks. As an example, the **Sort Tasks** user story will be highlighted below.

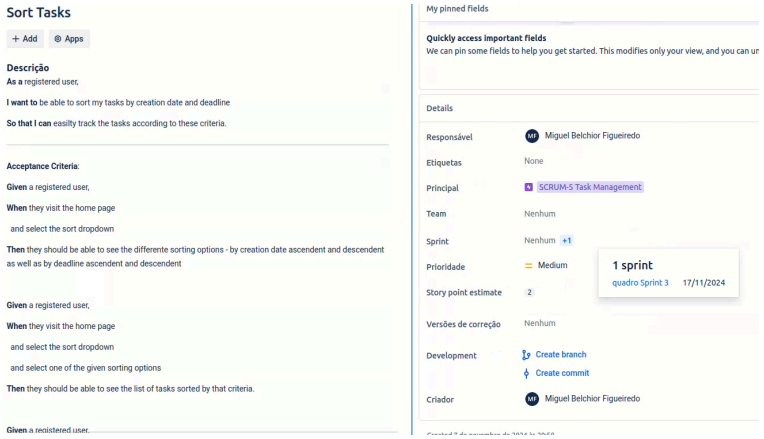


Figure 2 – Sort Tasks User Story Definition

Epic: Deployment

The following tasks were included in the **Deployment Epic**: Web Load Balancer & API Load Balancer, Frontend EC2, Backend EC2 and RDS. Unlike the previous epics, which were broken down into user stories, this epic was structured around technical tasks related to deploying the solution on AWS. Ideally, each of these deployment tasks would be linked to the earlier user stories as non-functional requirements. For example, a possible user story addressing an availability requirement for an Application Load Balancer could be: "**As a** user, **I want** the website to be available 99.99% of the time, **so that** I can rely on the platform to manage and track my tasks.". However, this was not possible due to time constraints, as the deployments were only carried out in the final sprint after the solution was fully developed.

Definition of Ready

The definition of ready outlines the conditions that must be fulfilled for a task or user story to be considered ready for inclusion in the product backlog and, consequently, ready for implementation. The definition of ready applied throughout this project is the following:

- **Independent** - The user story does not depend on another User Story(ies) to be implemented;
- **Small** - The user story must be concise, specific and focused on a single objective or functionality. It should be defined so that it cannot be further divided into smaller tasks and can be fully completed within a single sprint;
- **Estimated in story points** - The effort required to implement a user story is expressed in story points, providing a clear estimate of the overall required effort as shown in Table 1. In Table 1 it is outlined the rationale behind estimating story points, considering both the level of understanding of the task and the expected time required for its completion. It also includes an example of a user story for the simplest 1 story point user story which serves as a baseline for the other estimates - Building a landing Page. Due to the project's low

code complexity, no user stories were estimated higher than 3 story points.

How much is known about the task	**How much work/time effort will it require**	**Story points**	**Example**
Everything	Less than 1 hour	1	Building a Landing Page
Almost everything	1-3 hours	2	
Something	Half a day	3	

Table 1 – Task Effort Estimation

- **Testable with written Acceptance Criteria in the Gherkin Syntax** - The story must include clear, concise and testable acceptance criteria, to validate the completion of the solution and its alignment with the requirements.
- **Prioritization ranked appropriately in the product backlog** - Organize items of the backlog according to the project's and sprint's goals.

Definition of Done

The definition of done outlines the conditions that must be fulfilled for a product increment to be regarded as done/completed. The definition of done applied throughout this project is the following:

- **Developed** - The expected functionality or feature is fully developed.
- **Tested** - Tests for the User Story are developed and passing. These include unit testing, at controller and service layer, as well as functional testing using selenium web-browser and cucumber with a BDD approach. The unit tests are automated and integrated into an automated Sonar build pipeline, which will be discussed later. An example of this practice can be seen in the following user story: Add New Task.
- **Documented** - The API is documented using the OpenAPI specification;
- **Acceptance criteria met** - Ensure that all acceptance criteria specified in the user story are satisfied. Since the corresponding user interface tests are not automated, proof of passing tests must be provided. This proof should include a screenshot attached to the story in Jira. An example of this practice can be seen in the following user story: Add New Task;
- **Quality Gate Clearance** - Verify that the product increment meets all predefined quality standards and passes the quality gate.

The initial definition of done, established at the start of the project, was diligently adhered to during the early sprints, up until midway through the second sprint. However, due to the goal of this project being enabling students to gain practical experience in designing, developing and deploying a software solution on Amazon Web Services (AWS) and due to time constraints, the initial definition had to be revised and narrowed down to the following version:

- **Developed** - The expected functionality or feature is fully developed;
- **Documented** - The API is documented using the OpenAPI specification and is available here;

- **Acceptance criteria met** - Acceptance criteria continued to be specified for each user story. Although testing was no longer conducted, the development process remained aligned with the defined acceptance criteria to ensure functionality met the outlined requirements.
- **Quality Gate Clearance** - The final quality gates reflected the previous changes on testing phase, with coverage of 80% no longer being required. Automated Sonar Cloud builds are still enforced, with separate frontend and backend quality gates established, ensuring distinct evaluation criteria for each repository. Figure 3 highlights the successful SonarCloud builds, each of which meets the predefined criteria established in the respective quality gate.

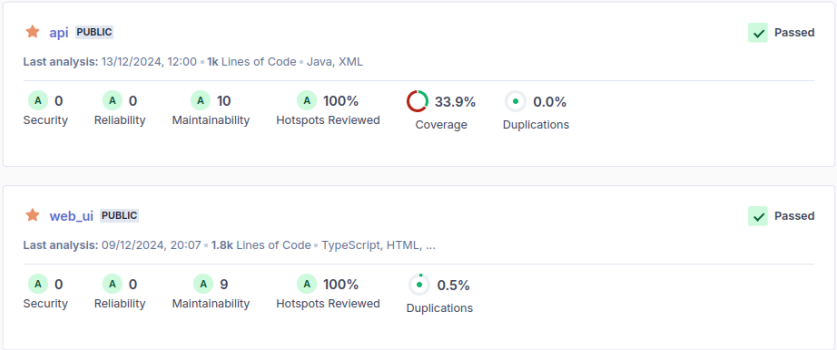


Figure 3 – Successful Sonar Cloud builds for web-ui and api repositories

Sprint Retrospectives

This section outlines the retrospectives for each sprint. For a comprehensive overview of all sprint reports considered in these sprint retrospectives, please refer to the sprints report page.

Sprint 1

The **goal** of the first sprint was to implement user stories related to user authentication, a critical feature for any user-based service, as well as some basic functionality regarding task creation. Therefore, the goal for this sprint was to lay the foundational groundwork, setting the stage to add more value to the system in the upcoming sprints. The **total effort for this sprint** was estimated at 9 story points, over a 7 day period. From the user stories planned for this sprint, the one related to creating a task was not completed due to some **difficulties**. The user authentication-related user stories took longer than expected due to a learning curve with Amazon Cognito and its authentication flow. Since these user stories had a higher priority, they were prioritized, and as they consumed more time than anticipated, the task creation user story could not be delivered by the end of the sprint. As a result, this **sprint's velocity** amounted to 7 story points. Therefore, the upcoming sprints should be planned with this velocity in mind.

Sprint 2

As the core unit of a To-Do List Application is the task, the **goal** of the second sprint was to implement all task management user stories related to CRUD task operations: creating, updating and deleting a task. The **total effort for this sprint** was estimated at 6 story points, over a 6 day period. All planned user stories on this sprint were completed as the sprint progressed smoothly without significant **difficulties**. However, the sprint was not closed promptly at the sprint deadline, which resulted in an irregularity in the burndown chart where the sprint continues until November 7th, despite all the user stories having already been completed. As a result, this

sprint's velocity amounted to 6 story points. Although not reflected in the jira's sprint report or burndown chart, it's important to note that the Definition of Done (DoD) was adjusted during this sprint with the goal of improving the sprints velocity, and remained for the following sprints.

Sprint 3

The third sprint's **goal** was to implement all remaining features, which included the user stories related to viewing the tasks, changing their status, as well as enabling filtering and sorting options. Therefore, the objective was to implement all remaining functionality so that the deployment of the solution could be done in the next sprint. The **total effort for this sprint** was estimated at 7 story points, over a 10 day period. The sprint's goal, burndown chart and the list of completed issues are highlighted in Figure 4. All planned user stories were completed without significant **difficulties**, with the **sprint's velocity** totaling to 7 story points.

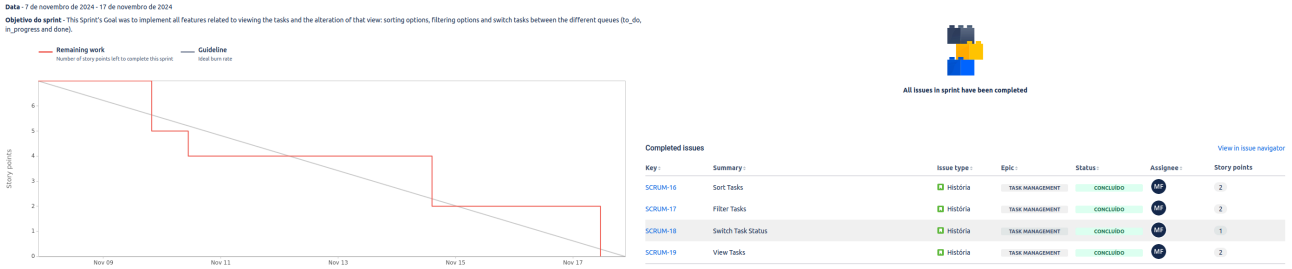


Figure 4 – Sprint Burndown Chart and Completed Issues

Sprint 4

The **goal** of the fourth and last sprint was to implement all tasks related to the deployment of the solution in Amazon Web Services (AWS). The **total effort for this sprint** was estimated at 10 story points over a 6 day period. All planned user stories were successfully completed, as the sprint proceeded without any major **difficulties**. Furthermore, the sprint was completed three days ahead of the final deadline, revealing a story point overestimation during the planning phase, which was caused by limited familiarity with AWS at that time. This resulted in a **sprint's velocity** of 10 story points.

3 Solution Technical Implementation

Architecture

The solution employs a **microservices architecture**. Each microservice is independent, has a single responsibility (separation of concerns) and should and is deployed as a separate unit, allowing an easier and better deployment and decoupling. Therefore, the change or stoppage of one of the microservices will not affect the proper functioning of the others.

The solution consists of two distinct microservices: one dedicated to the web interface and other to the API. In this report, the terms frontend and backend will be used interchangeably to refer to the web and API microservices, respectively. The API microservice could have been further divided into a task microservice and a user microservice. However, this was deemed unnecessary for the current project, as there were no user-specific features requiring managing personalized

user data, such as preferences or settings, only being required an endpoint to obtain a token using the authorization code in the authentication flow. As a result, these functionalities were consolidated into a single main API microservice.

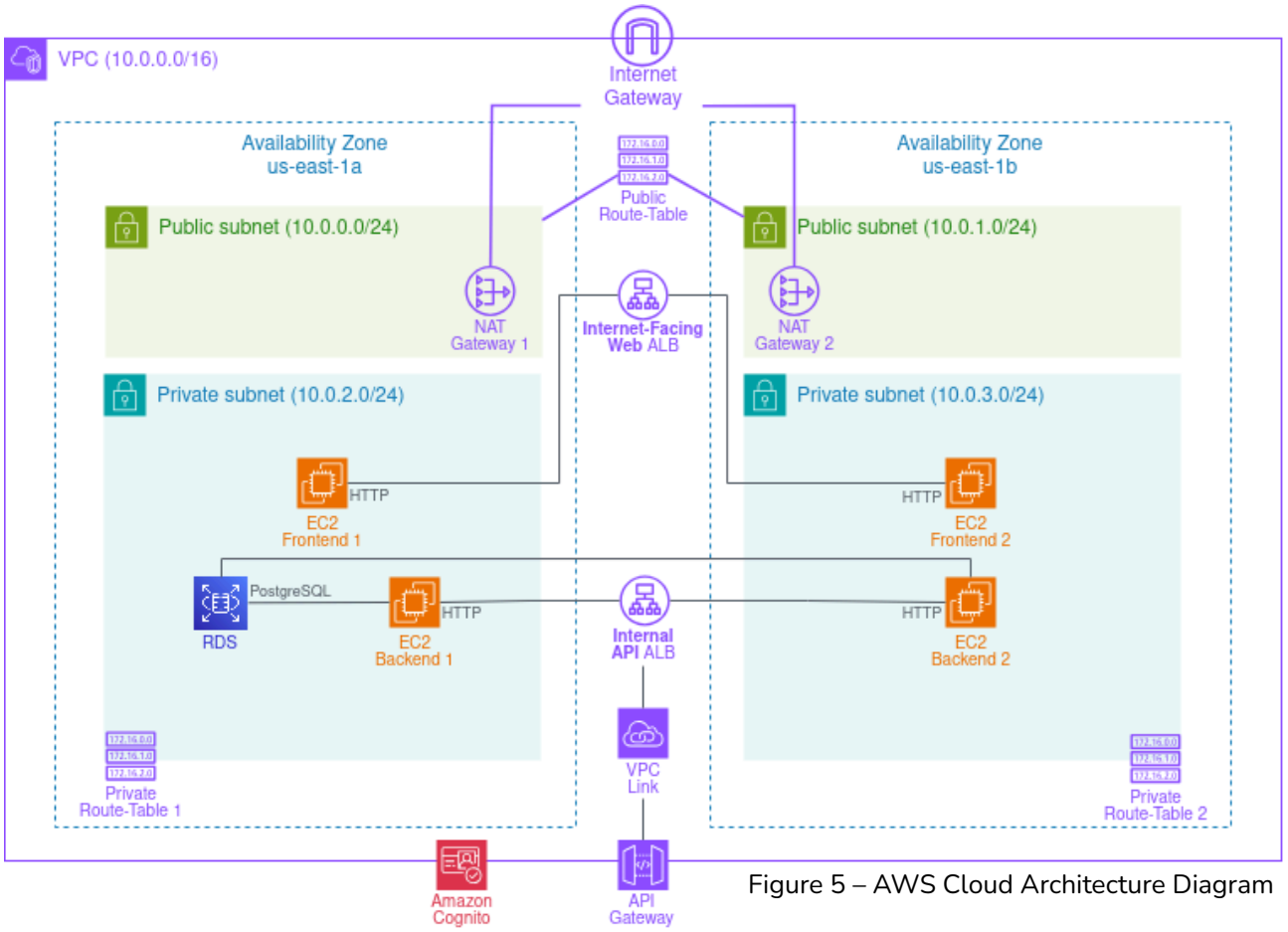


Figure 5 – AWS Cloud Architecture Diagram

Figure 5 illustrates the architecture of the solution implemented on Amazon Web Services, designed to ensure high-availability, fault-tolerance and security. The infrastructure is deployed within the AWS region **us-east-1** (US East - N. Virginia). The Virtual Private Cloud (VPC) spans over multiple availability zones – **us-east-1a** and **us-east-1b** – to provide redundancy and minimize the risk of service disruption, ensuring a resilient and reliable infrastructure, even in the event of an AZ failure or outage. Each availability zone contains a public and private subnet. In each of the private subnets, there are ec2 instances for both the frontend and backend. The frontend instances are available to the consumer through an internet-facing web Application Load Balancer (ALB), which accepts traffic over HTTPS for secure communication and forwards it over HTTP to the EC2 frontend instances.. The backend/API is exposed securely through an API Gateway, which handles requests over HTTPS. Through a VPC Link , this API Gateway routes traffic to an internal API ALB, which then forwards the requests over HTTP to the API target group containing both ec2 backend instances. Both backend instances access the same postgreSQL RDS database that is available on us-east-1a availability zone. This Amazon RDS instance is not distributed across multiple availability zones, due to this facility not being included in the free tier and having a limited budget of 50$ in aws learners lab. However, it would be ideal to perform a multi-az deployment with multiple RDS instances across different Availability Zones, allowing AWS to manage replicas and automatic failover, ensuring improved fault tolerance and performance. AWS Cognito was utilized for user management, implementing robust authentication, authorization, and accounting (AAA) mechanisms.

Authentication Flow with Amazon Cognito

Amazon Cognito was selected as the identity platform to implement authentication, authorization, and accounting (AAA) mechanisms, providing a robust solution for managing user access and identity. A single user-pool has been created in the us-east-1 AWS region to authenticate and authorize users to the web app and API. The Figure below also displays the JWKS endpoint, which provides the public keys required to validate JWT signatures. These keys ensure the tokens are genuine and were issued by the Cognito user pool. Email and password authentication was implemented using Cognito, which sends a verification email to the user.

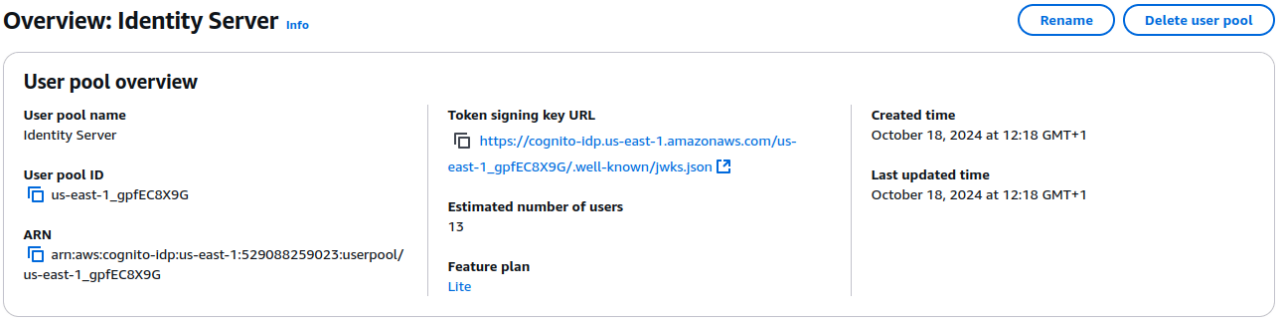


Figure 6 – AWS Cognito User Pool

In Amazon Cognito, the following app client was registered, representing an application capable of interacting with the Cognito user pool for authentication purposes. The app client was assigned a **Client ID**, which will be used in the application's web app to identify the application making the authentication request. The **Client Secret** will be used in the backend in combination with the Client ID for the authentication flows, which will be discussed next.

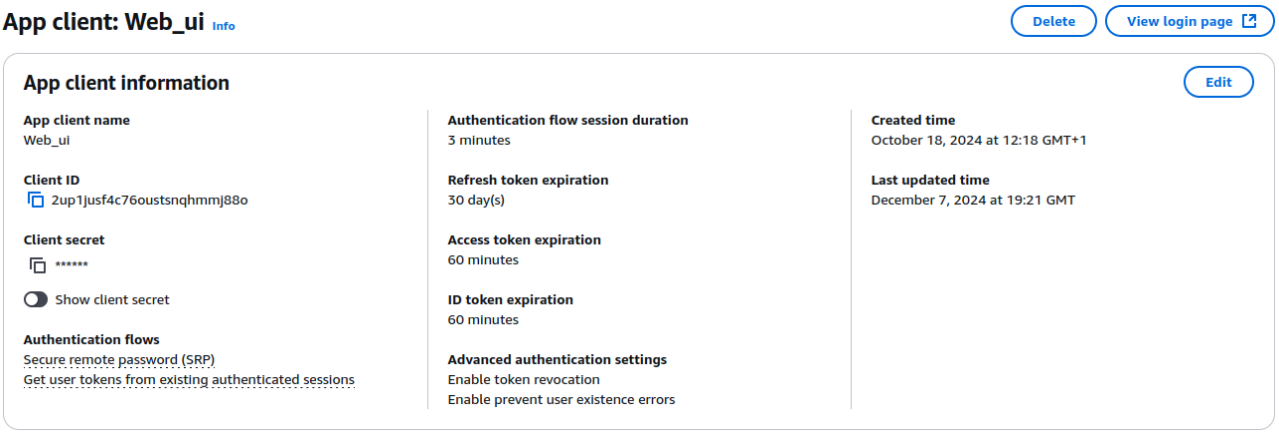


Figure 7 – Web App Client

The OAuth 2.0 authorization code grant type, or *auth code flow* was chosen as the authentication flow and works as follows:

1. The web application starts the process by redirecting the user to the AWS Cognito login hosted-ui endpoint, which is built as follows. https://**<AUTH_DOMAIN>**/login?**response_type=code**&**scope=email+openid**&**client_id=<CLIENT_ID>**&**redirect_uri=<REDIRECT_URI>**
2. Upon successful authentication, the authorization server (Cognito) issues an authorization code and redirects the user back to the application using the specified redirect URI - this redirect URI was previously configured in Cognito and uses the website's HTTPS URL, as shown in Figure 22 of appendix. Therefore, Cognito redirects the user to the following page: https://**<REDIRECT_URI>**?**code=<AUTHORIZATION_CODE>**
3. The web application exchanges the authorization code for an access token by making a

request to the sign-in endpoint of the API.

4. The api then exchanges the authorization code for a JWT access token by performing a post request to the token issuer endpoint: **https://<AUTH_DOMAIN>/oauth2/token**. The request includes:
   - the authorization header **Basic *Base64Encode(<CLIENT_ID> : <CLIENT_SECRET>)***
   - a content type of ***application/x-www-form-urlencoded***;
   - the following body parameters: **grant_type=authorization_code**, **code=<AUTHORIZATION_CODE>**, **redirect_uri=<REDIRECT_URI>** and **client_id=<CLIENT_ID>**.
5. Finally, the API returns to the frontend a JWT access token, which is required for all subsequent API requests. Upon each request, the token will be verified using the previously discussed JWKS URL ensuring the token is valid and has not been tampered with.

VPC

The main VPC, **IAP-vpc,** contains across both availability zones a public and a private subnet. In **us-east-1a**, the present subnets are **IAP-subnet-public1-us-east-1a** (public) and **IAP-subnet-private1-us-east-1a** (private), while in **us-east-1b**, the present subnets are **IAP-subnet-public2-us-east-1b** (public) and **IAP-subnet-private2-us-east-1b** (private). As previously mentioned both public subnets share the same route table **IAP-rtb-public** , while each private subnet is associated with its own dedicated route table: **IAP-rtb-private1-us-east-1a** and **IAP-rtb-private2-us-east-1b.**

Name	Subnet ID	State	VPC	Availab...	Route table
IAP-subnet-private1-us-east-1a	subnet-04fbd607140bcd56c	Available	vpc-01e89686aff50c0d1 | IAP-vpc	us-east-1a	rtb-04f98e5ef167a0c3c | IAP-rtb-private1-us-east-1a
IAP-subnet-public1-us-east-1a	subnet-0ea7e1993bb82f1bb	Available	vpc-01e89686aff50c0d1 | IAP-vpc	us-east-1a	rtb-08bb8318f59178228 | IAP-rtb-public
IAP-subnet-public2-us-east-1b	subnet-08eaa621cee1811c9	Available	vpc-01e89686aff50c0d1 | IAP-vpc	us-east-1b	rtb-08bb8318f59178228 | IAP-rtb-public
IAP-subnet-private2-us-east-1b	subnet-065bfbf5b6d099374	Available	vpc-01e89686aff50c0d1 | IAP-vpc	us-east-1b	rtb-03eb5218b77221a02 | IAP-rtb-private2-us-east-1b

Figure 8 – IAP VPC subnets

Figure 9 illustrates the previously mentioned route tables. From further inspection of these route tables it is clear that the public subnets have a direct route to an internet gateway to access the public internet while the private subnets do not, being required a NAT device for its resources to access the public internet. It's also important to note that all NAT gateways must be in a public subnet as they need to communicate with the Internet and therefore need a route to the Internet Gateway. Additionally, every route table includes a local route, which is used to enable communication within the VPC. As a result, any traffic destined for a target within the VPC (10.0.0.0/16) is covered by the local route, and therefore is routed within the VPC. All other traffic from the subnet is routed through the internet gateway for public subnets or the NAT gateway for private subnets, depending on the subnet type.

Destination	Target	Destination	Target	Destination	Target
0.0.0.0/0	igw-0b10408eb2de9cfe9	0.0.0.0/0	nat-0c218a2ceeb14932a	0.0.0.0/0	nat-0d343e240eff62bda
10.0.0.0/16	local	10.0.0.0/16	local	10.0.0.0/16	local

Figure 9 – IAP-rtb-public, private1-us-east-1a and private2-us-east-1b route tables

Web Application Load Balancer

Elastic Load Balancing automatically distributes the incoming traffic across multiple targets, in this case, the frontend EC2 compute instances that are spread across multiple availability zones, ensuring efficient load distribution and high availability. The web ALB – referred to as WebALB in the architecture diagram and as **iapALB** in the attached aws figures – will serve as the entry point

for users accessing the web interface and will forward the incoming requests to the respective frontend EC2 instance. Therefore, the respective security group **ALB Security Group** must allow inbound HTTPS traffic on port 443 from the clients on the internet (0.0.0.0), as well as outbound HTTP traffic through port 80 to the frontend Security Group, which as the name suggests is the security group associated with the frontend EC2 instances.

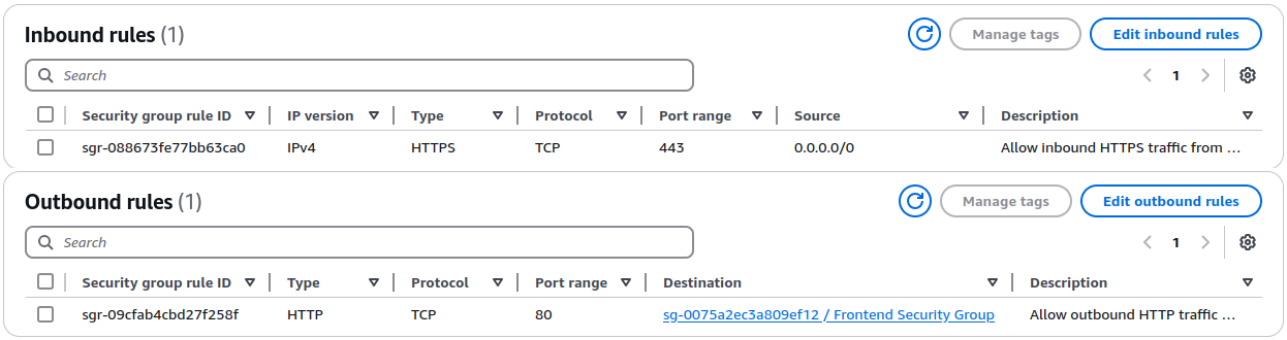


Figure 10 – Web ALB Security Group Rules

The Web ALB is deployed across the us-east-1a and us-east-1b availability zones, with a listener configured for HTTPS on port 443 that forwards requests to the Frontend Target Group. As the name implies, the Frontend Target Group routes the requests over HTTP on port 80 to two registered targets – one for each frontend EC2 instance in the respective availability zones.

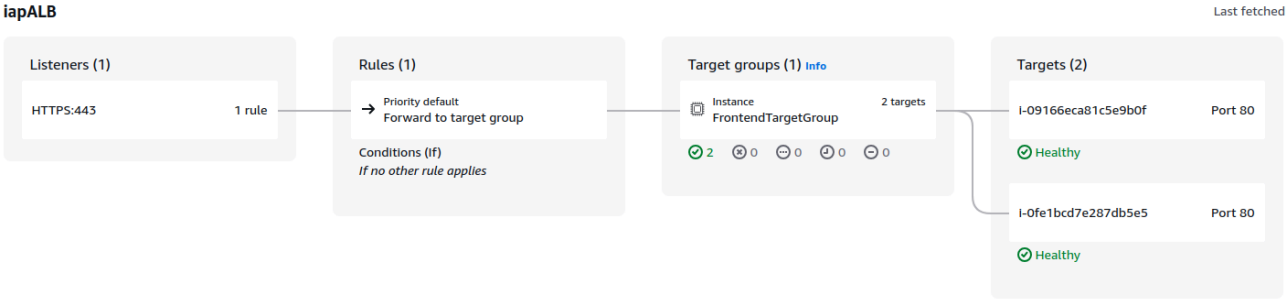


Figure 11 – Web ALB Resource Map

In order to enable HTTPS communication, the SSL/TLS certificate provided by professor Rafael Direito has been used. Since the domain of the certificate, **es-ua.ddns.net,** doesn't match the load balancer's domain, the IP addresses of the load balancer have been mapped to es-ua.ddns.net in the /etc/hosts file, thereby bypassing DNS resolution on the respective computer and allowing to access the website without encountering the "net::ERR_CERT_COMMON_NAME_INVALID" error.

EC2 Frontend

Amazon Elastic Compute Cloud (EC2) instances, which provide on-demand, scalable computing capacity in the AWS Cloud, have been used to provide the web interface to the final user. To deploy the web interface on the EC2 instances, the process begins by assigning the **LabRole** IAM Role to the instance and establishing a connection via AWS Session Manager, eliminating the need to use SSH for access. Upon connection, necessary preparations are made including installing Docker to set up the environment for container execution. The web interface, packaged as a Docker image on Docker Hub, is then pulled and the container executed within the virtual machine. The container is configured to expose the designated port in compliance with the security group rules. The Dockerfile used to run the Angular frontend employs a multi-stage build process. The first stage builds an Angular application for production using a Node.js base image.

In the second stage, it uses an NGINX base image to serve the built application, configuring NGINX with a custom configuration. This multi-stage approach results in a smaller and more secure final Docker image.

```
# Stage 1: Build Angular App
FROM node:lts-alpine3.20 AS build
WORKDIR /app
COPY . /app
RUN npm install
RUN npm run build --prod
```

```
# Stage 2: Serve the app with NGINX
FROM nginx:stable-alpine3.20-perl
COPY nginx.conf /etc/nginx/nginx.conf
COPY --from=build /app/dist/web-ui/browser
                  /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

As previously stated, both frontend EC2 instances are located in the private subnets of their respective availability zones. These instances are assigned only private IPv4 addresses and are associated with the same security group, the **Frontend Security Group**. The security group includes an inbound rule allowing HTTP traffic on port 80 from the previously mentioned web application load balancer security group, and an outbound rule permitting internet access through 0.0.0.0/0, which corresponds to the NAT Gateway destination. This setup enables the EC2 instances to access external resources for tasks such as installing packages, setting up Docker, and pulling Docker images.

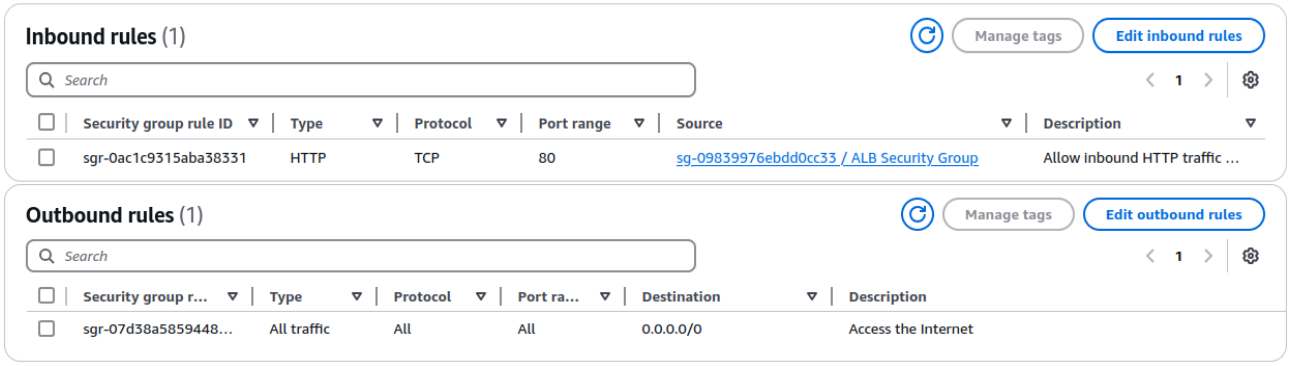


Figure 12 – Frontend Security Group

API Application Load Balancer

In addition to the internet-facing Web Application Load Balancer, which manages inbound traffic from users accessing the web interface, the architecture also includes an **internal API Application Load Balancer**. This load balancer spans the same availability zones as the Web ALB, ensuring high availability and fault tolerance. However, unlike the Web ALB the API ALB is an **internal application load balancer**, meaning it is not accessible from the public internet.

The nodes of the **Web ALB**, being an internet-facing load balancer, have public IP addresses. The DNS name of the Web ALB is publicly resolvable to the public IP addresses of its nodes, allowing it to route requests from clients over the internet to the EC2 frontend resources in the VPC. In contrast, the nodes of the **API ALB** are part of an internal load balancer and only have private IP addresses. The DNS name of the internal API ALB is publicly resolvable to the private IP addresses of its nodes. Therefore, the API ALB can only route requests from clients with access to the VPC, which is facilitated through the use of the previously discussed VPC Link. This conclusion is supported by Figure 13.

Network interface ID	Subnet ID	VPC ID	Availability Zone	Security group n...	Public IPv4 address	Primary private IPv4 address
eni-03cebf8245adc8c0d	subnet-04fbd607140bcd56c	vpc-01e89686aff50c0d1	us-east-1a	API ALB Security Group	–	10.0.2.50
eni-095602e0b0583e582	subnet-0ea7e1993bb82f1bb	vpc-01e89686aff50c0d1	us-east-1a	ALB Security Group	3.224.70.119	10.0.0.186
eni-0c99a46666a25283c	subnet-08eaa621cee1811c9	vpc-01e89686aff50c0d1	us-east-1b	ALB Security Group	54.204.247.51	10.0.1.197
eni-08ff8686582d7d0ad	subnet-065bfbf5b6d099374	vpc-01e89686aff50c0d1	us-east-1b	API ALB Security Group	–	10.0.3.250

Figure 13 – ALB Network Interfaces

Since the VPC Link resource connects the routes from the API Gateway to the internal API Application Load Balancer, the **API ALB Security Group** must allow inbound HTTP traffic on port 80 from the **VPC Link Security Group** and allow outbound HTTP traffic through port 80 to the **Backend Security Group**.

Inbound rules (1)

Manage tags | Edit inbound rules

Search

Security group rule ID	Type	Protocol	Port ra...	Source	Description
sgr-0b974a539074e9c88	HTTP	TCP	80	sg-031c7d90571858ad6 / VPC Link Security Group	Allow inbound HTTP traffic fr...

Outbound rules (1)

Manage tags | Edit outbound rules

Search

Security group rule ID	Type	Protocol	Port ra...	Destination	Description
sgr-0338f81ac90b614bf	HTTP	TCP	80	sg-076c1e3d2028ed711 / Backend Security Group	Allow outbound HTTP traffic ...

Figure 14 – API ALB Security Group Rules

The API ALB is deployed across the us-east-1a and us-east-1b availability zones, with a listener configured for HTTP on port 80 that forwards requests to the **ApiTargetGroup**. As the name implies, the **ApiTargetGroup** routes the requests over HTTP on port 80 to two registered targets – one for each backend/api EC2 instance in the respective availability zones.

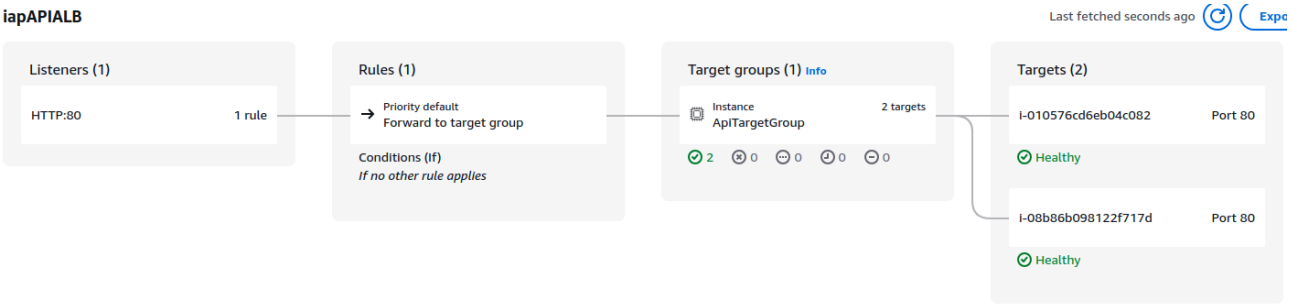


Figure 15 – API ALB Resource Map

EC2 Backend

Most of the configuration details discussed for the frontend EC2 instances also apply to the backend EC2 instances with the main differences residing on its security group and the utilized Dockerfile, which will be explained next. For the same reasons mentioned previously, this dockerfile also follows a multi-stage approach. The first stage uses a JDK image to build the application and generate the JAR file, while the second uses a smaller JRE image to package the application by copying the JAR file from the build stage. An entrypoint is configured to run the Spring java application. It's of utmost importance to note that, for security reasons, only the final JAR is copied during the package stage, with the .env file and any other files being excluded from the final image. A dockerignore file is also included to ensure this. Therefore, the environment file will be provided at container runtime in each EC2 virtual machine, based on the predefined application.properties, to prevent the exposure of sensitive details such as the client secret.

```
# Stage 1: Build the JAR file
FROM eclipse-temurin:17-jdk-alpine
                            AS build
WORKDIR /app
COPY . .
RUN ./mvnw clean package -DskipTests
```

```
# Stage 2: Package Stage - jre
FROM eclipse-temurin:17-jre-alpine
COPY --from=build /app/target/api-
0.0.1-SNAPSHOT.jar /app/runner.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar",
                "/app/runner.jar"]
```

As previously mentioned, the backend/api EC2 instances aren't exposed the same way as the frontend ec2 instances, which are accessed via a simple application load balancer over HTTPS. Nevertheless, an application load balancer, the API ALB, it's still necessary to distribute the incoming traffic across the EC2 targets in both availability zones. This is accounted for in the **Backend Security Group**. The security group includes an inbound rule allowing HTTP traffic on port 80 from the API ALB security group, and an outbound rule permitting internet access through 0.0.0.0/0, which corresponds to the NAT Gateway destination.

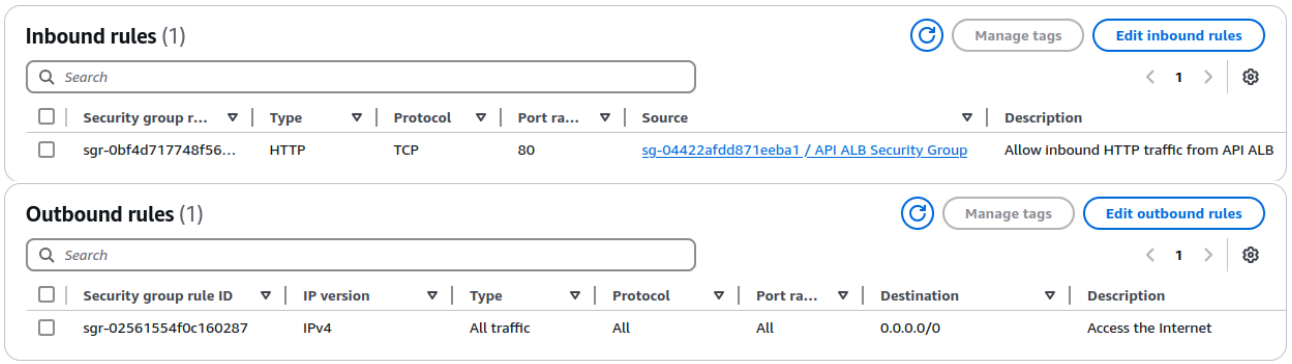


Figure 16 – Backend Security Group

Furthermore, the API requires access to the RDS database, which is ensured by the **ec2-rds-1** security group. This security group was created automatically by aws upon creation of the postgres database using Amazon Relational Database Service (RDS), based on the specified EC2 instance intended for database connectivity. This security group contains only an outbound rule that allows connections on port 5432 to **postgres-database-1** from any instances to which the security group is attached. This security group is attached to both backend EC2 instances.

Figure 17 – ec2-rds-1 Security Group

API Gateway

Amazon API Gateway is an AWS service for creating, publishing, maintaining, monitoring, and securing REST, HTTP, and WebSocket APIs at any scale. Additionally, API Gateway offers an extra layer of security with a multitude of built-in features such as rate limiting, which helps mitigate DDoS attacks. It's often paired with Amazon Cognito user pools or Lambda authorizer functions in order to control access to the API. However, in this project, access control was implemented at the application level within the API code, rather than using Lambda authorizers. Therefore, the API Gateway is present here primarily to expose the API over HTTPS. Since Amazon API Gateway does not support unencrypted (HTTP) endpoints, it assigns by default an internal domain to the API that automatically uses the Amazon API Gateway certificate, ensuring

secure communication. The API Gateway HTTPS endpoint is shown in Figure 25 of the appendix.

The API Gateway configuration includes only a single route: **ANY /{proxy+}**. This route allows API Gateway to accept any HTTP method for any path under the root path, while using a proxy resource defined by the greedy path variable **{proxy+}** to forward the requests to the backend. However, as previously mentioned the EC2 backend instances are not available through the previously mentioned load balancer which serves as the access point for the frontend instances, but instead through an internal application load balancer. By utilizing a VPC link it was possible to establish a private integration that connects the API Gateway routes to the internal API Application Load Balancer, a private resource within the VPC. The API ALB then forwards the requests to the **ApiTargetGroup** target group, which includes both EC2 api/backend instances.

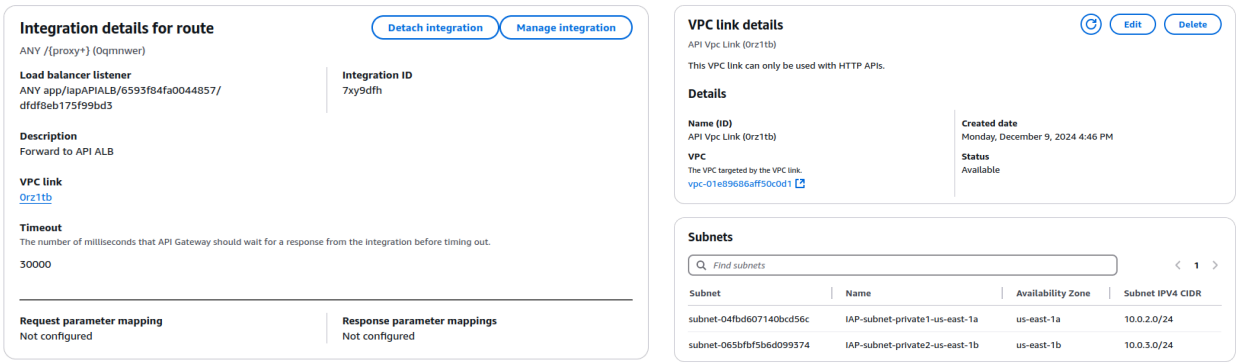


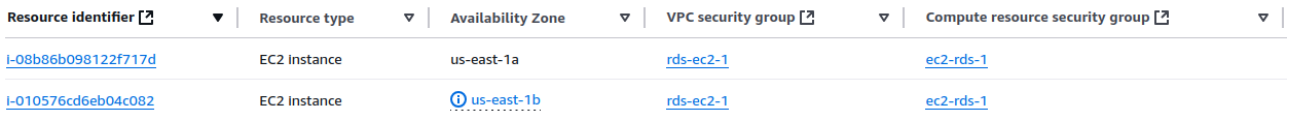
Figure 18 – API Gateway integration,on the left, and VPC Link details, on the right

The VPC link is associated with a dedicated security group, the **VPC Link Security Group**, which includes an inbound rule allowing HTTP traffic on port 80 as well as an outbound rule allowing HTTP traffic through port 80 to the API ALB Security Group, which is the security group associated to the internal API ALB.

Figure 19 – VPC Link Security Group

RDS

Amazon Relational Database Service (RDS) was used to set up, operate, and scale a postgreSQL relational database in the AWS Cloud. The RDS instance was configured with a subnet group of private subnets **IAP-subnet-private1-us-east-1a** and **IAP-subnet-private2-us-east-1b**. This Amazon RDS instance is not distributed across multiple availability zones, due to this facility not being included in the free tier and having a limited budget of 50$ in aws learners lab. However, it would be ideal to perform a multi-az deployment with multiple RDS instances across different Availability Zones, allowing AWS to manage replicas and automatic failover, ensuring improved fault tolerance and performance. As a result, the database is hosted as a single instance in the **us-east-1a** availability zone, which is connected to the two api EC2 instances located in the different availability zones. These EC2 instances must have the **ec2-rds-1** security group.



Resource identifier	Resource type	Availability Zone	VPC security group	Compute resource security group
i-08b86b098122f717d	EC2 instance	us-east-1a	rds-ec2-1	ec2-rds-1
i-010576cd6eb04c082	EC2 instance	us-east-1b	rds-ec2-1	ec2-rds-1

Figure 20 – RDS Connected Compute Resources

The RDS instance has the following security group attached to it: **rds-ec2-1**. This security group allows the EC2 instances with **ec2-rds-1** security group attached to them to connect to the database through port 5432.

Figure 21 – rds-ec2-1 Security Group

4 Implementation Details and Conclusion

All features outlined in the solution specified on the project description have been successfully implemented, with some remarks regarding the sorting and the filtering functionalities. Firstly, all sorting and filtering operations are provided by the API. Secondly, the dashboard inherently divides and displays tasks into columns labeled TO_DO, IN_PROGRESS, and DONE based on their completion status. Consequently, a separate filter for tasks by completion status is unnecessary, as this categorization is built into the interface. Similarly, a sorting option by completion status is redundant, as tasks are already organized into these predefined columns.

The project's goal of designing, developing, and deploying a secure and robust software solution on Amazon Web Services (AWS) has been accomplished. This achievement reflects a comprehensive approach that integrated agile development methodologies, leveraging epics and user stories to efficiently manage project tasks and ensure iterative progress. The final solution includes a fully functional RESTful API, a user-friendly Web UI, and seamless interaction with a relational database, all deployed following best practices in cloud security and containerization.

5 References and resources

Reference Materials

- Internal Classical Load Balancer - aws docs
- Set up proxy api gateway - aws docs
- Set up VPC links for HTTP APIs in API Gateway - aws docs
- AWS Academy Modules
- AWS Academy Learners Lab

Project Resources

- Github Organization
- Jira Project
- API Documentation using the OpenAPI specification
- Frontend Docker Image
- Backend Docker Image

6 Appendix

Amazon Cognito

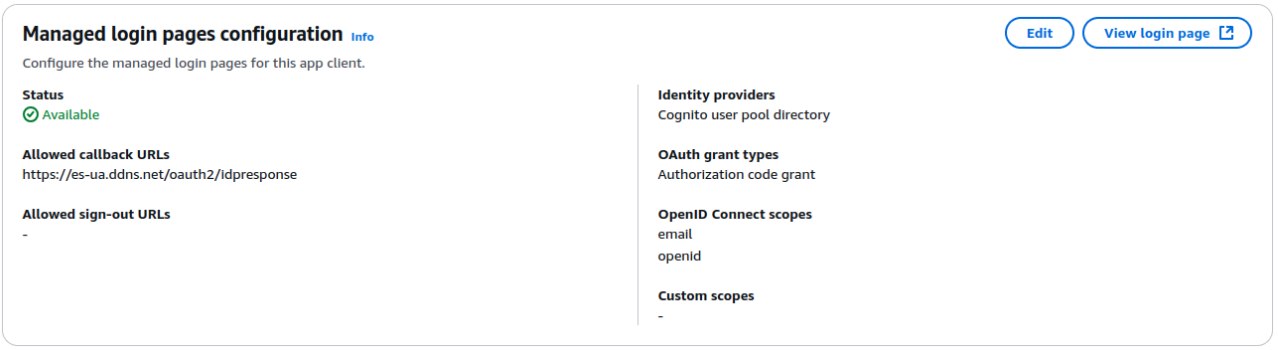


Figure 22 – Amazon Cognito Login Page Configuration

VPC

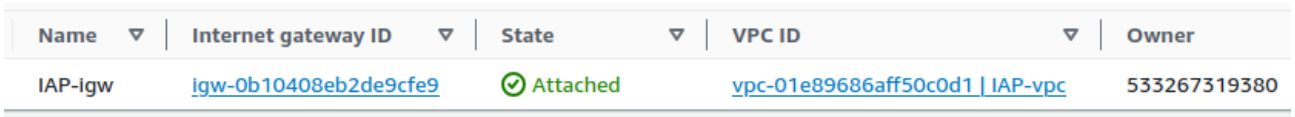


Figure 23 – Internet Gateway

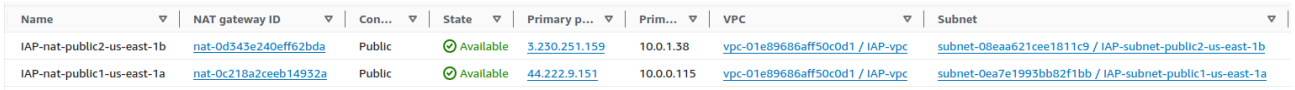


Figure 24 – NAT Gateways

API Gateway

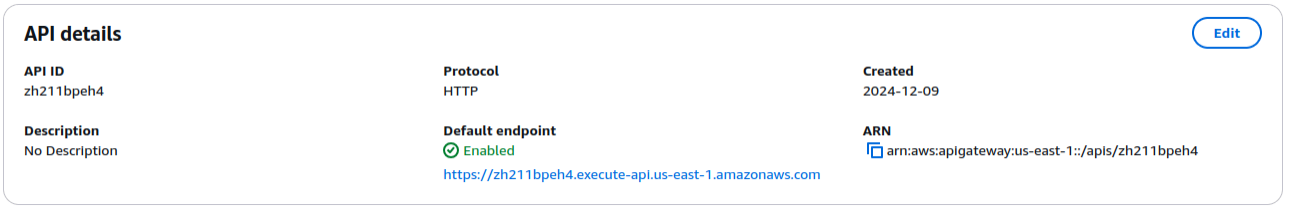


Figure 25 – API Gateway Configuration

Web Application Load Balancer

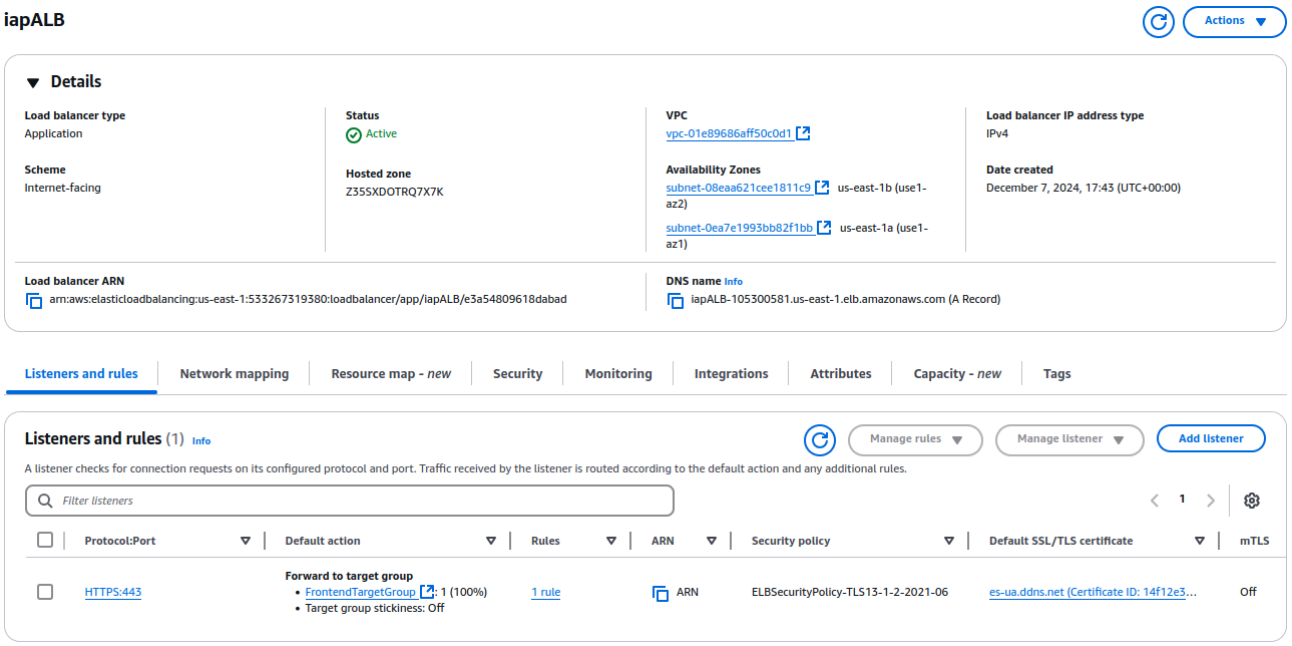


Figure 26 – Web ALB Configuration

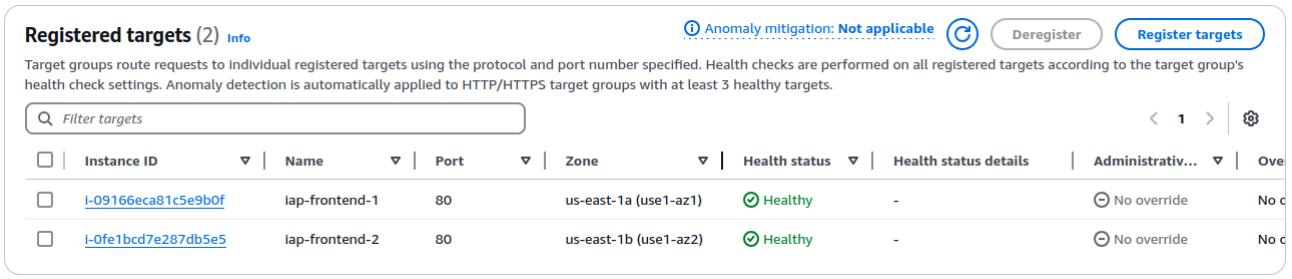


Figure 27 – Frontend Target Group registered targets

API Application Load Balancer

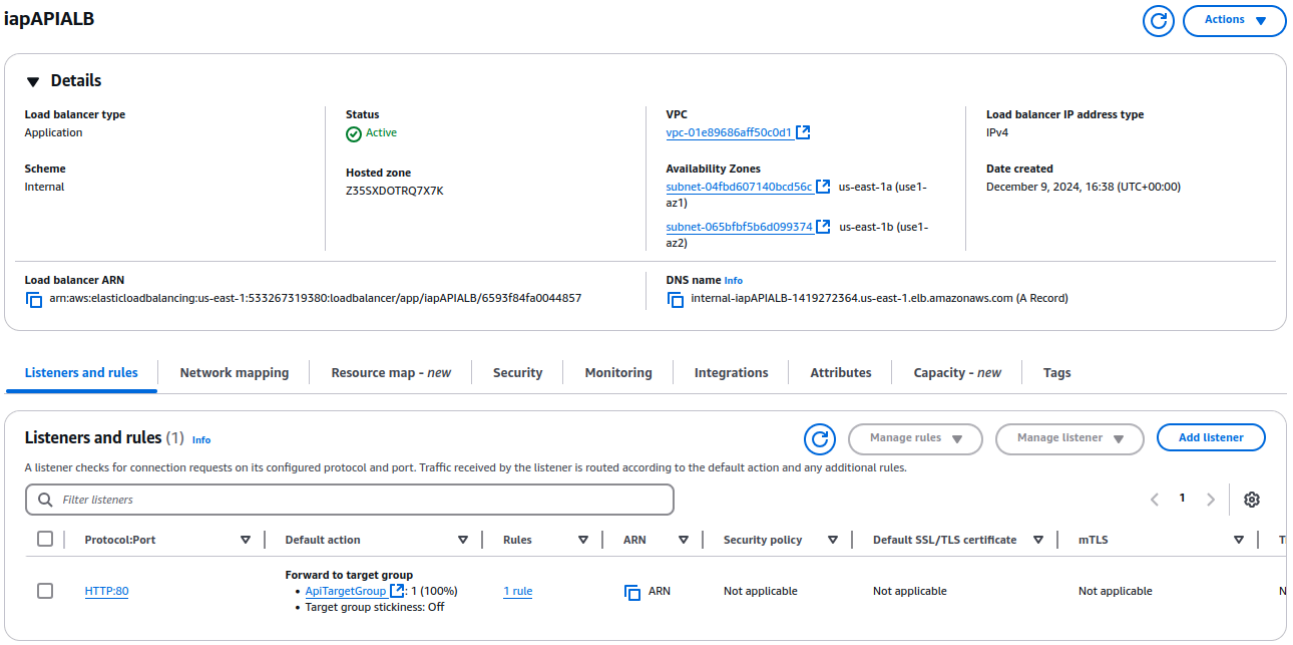


Figure 28 – API ALB Configuration

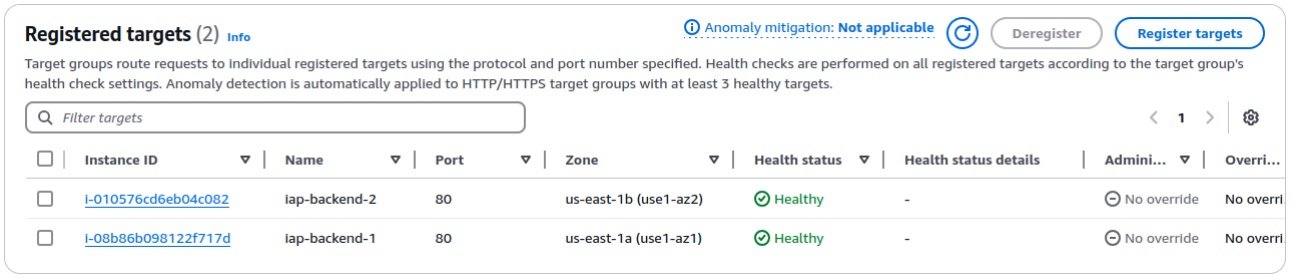


Figure 29 – API Target Group registered targets

EC2 Frontend

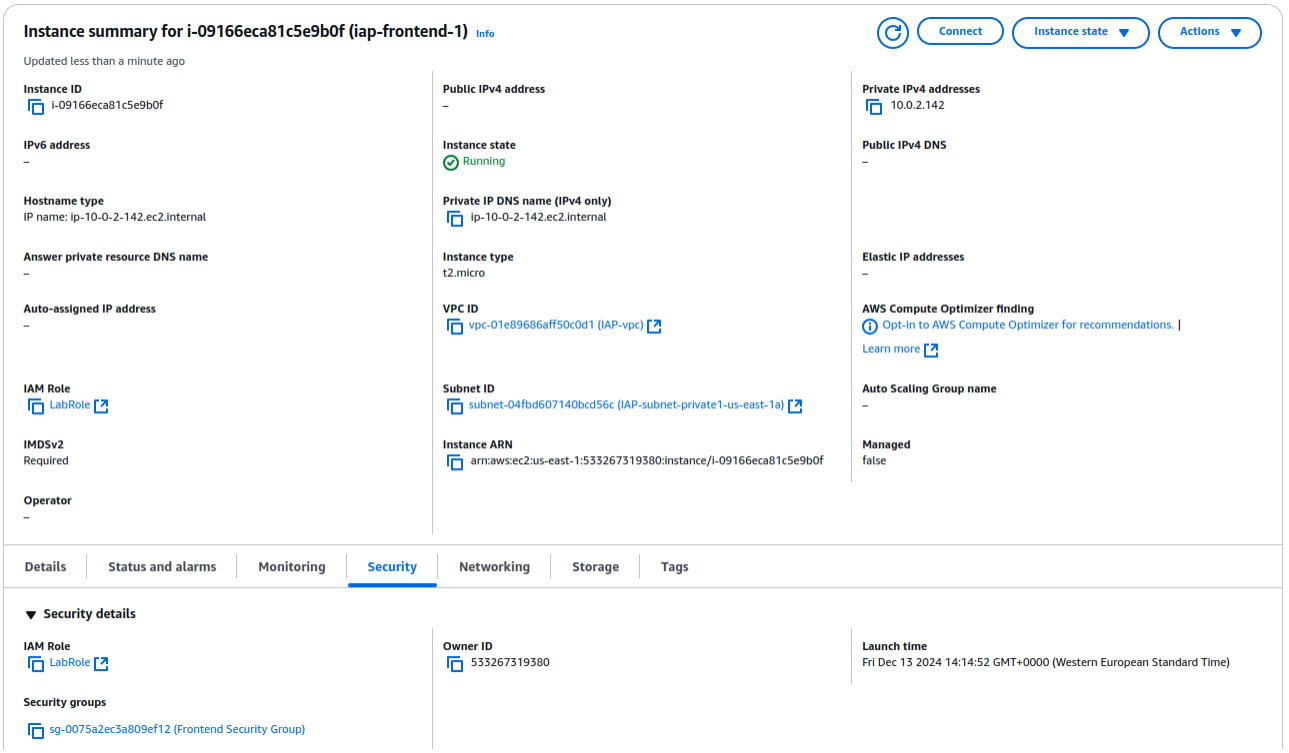


Figure 30 – First frontend EC2 instance configuration

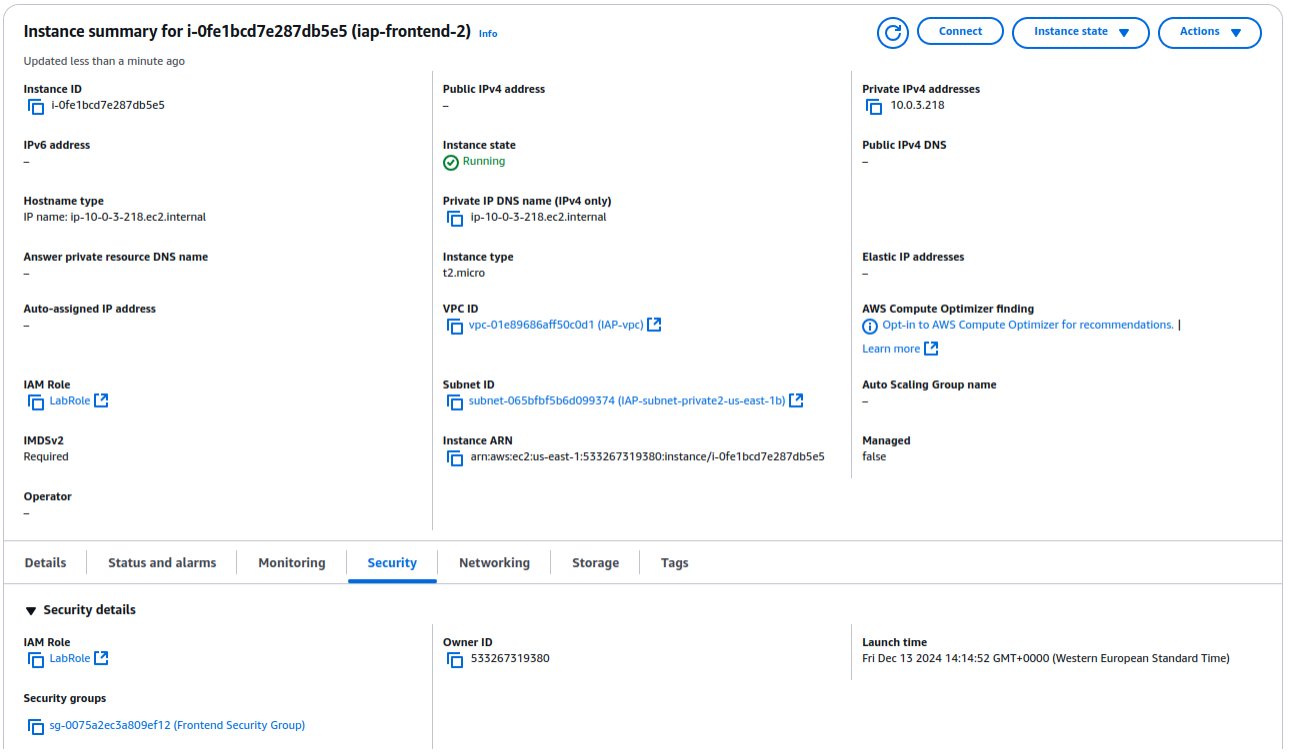


Figure 31 – Second frontend EC2 instance configuration

EC2 Backend

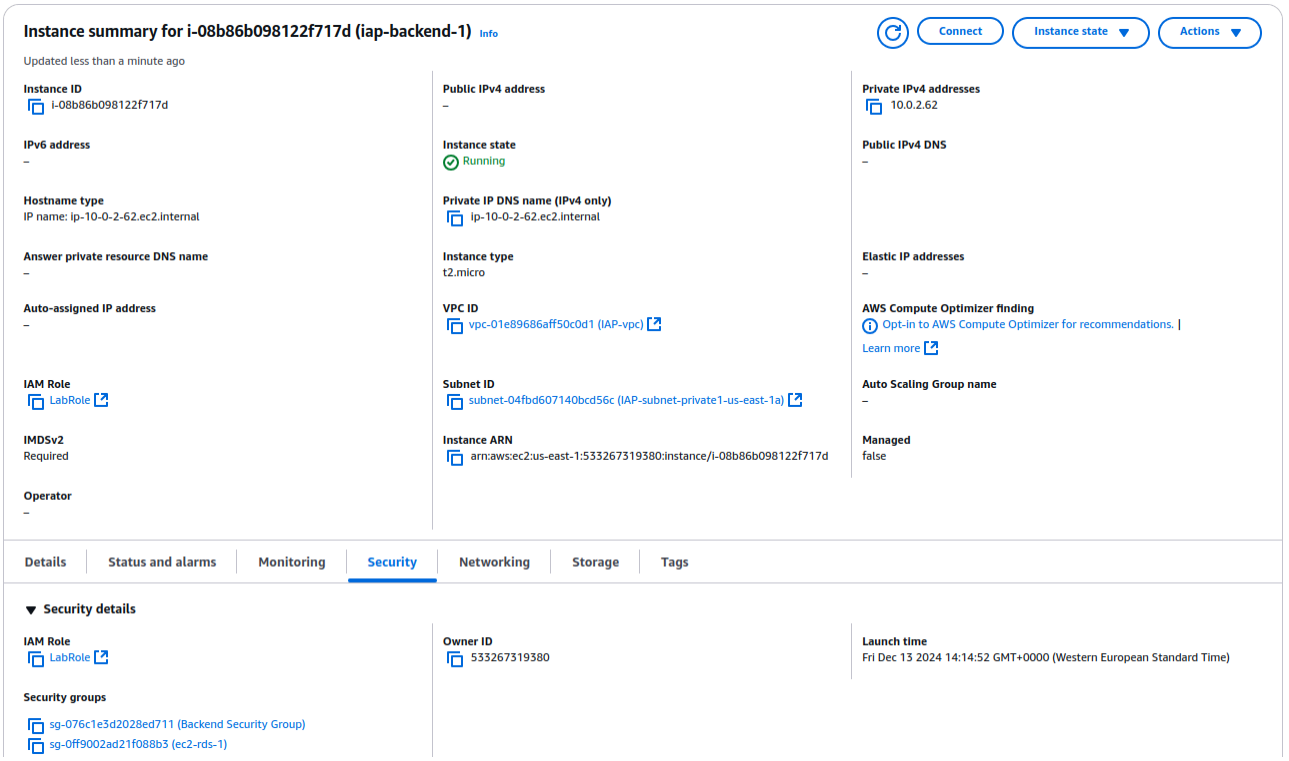


Figure 32 – First backend EC2 instance configuration

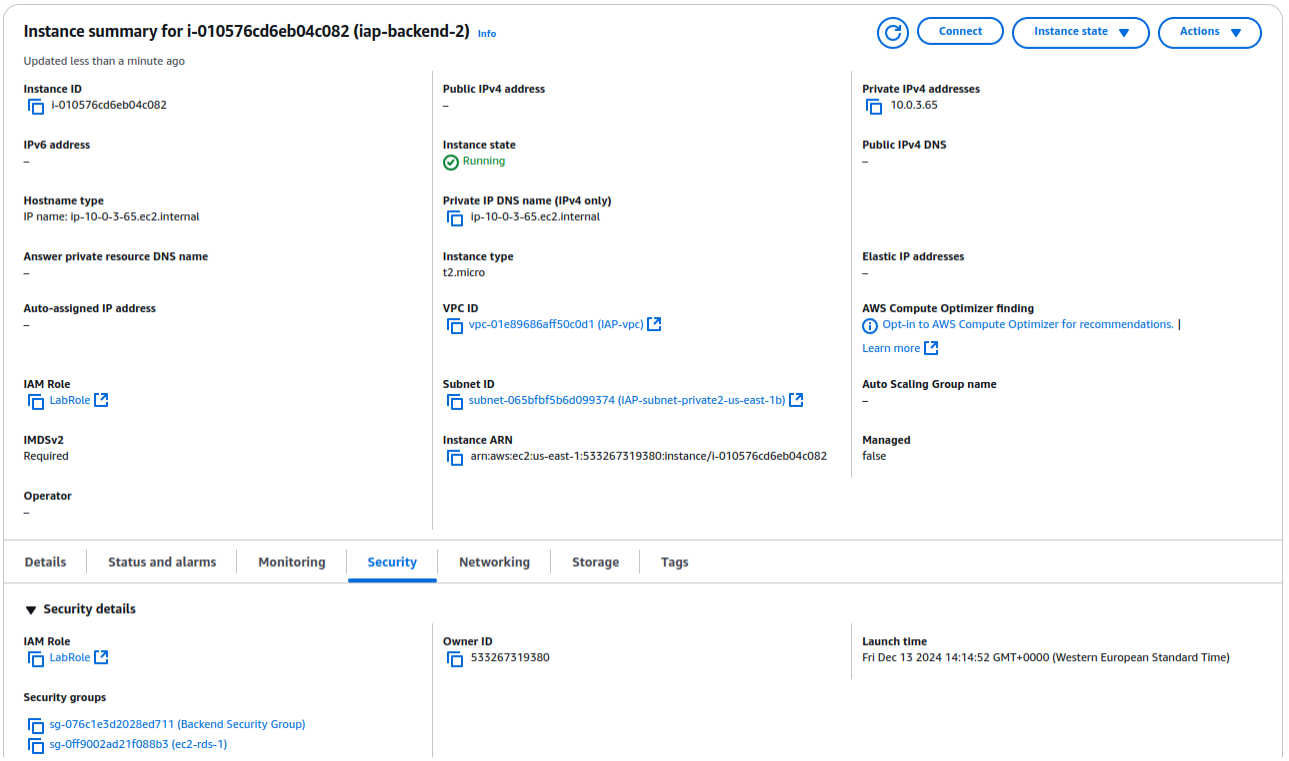


Figure 33 – Second backend EC2 instance configuration

RDS

postgres-database-1

Summary

DB identifier postgres-database-1	**Status** Available	**Role** Instance	**Engine** PostgreSQL
CPU 4.18%	**Class** db.t3.micro	**Current activity** 0.00 sessions	**Region & AZ** us-east-1a

Connectivity & security | Monitoring | Logs & events | Configuration | Maintenance & backups | Data migrations - *new* | Tags | Recommendations

Connectivity & security

Endpoint & port

Endpoint
postgres-database-1.cdymcyg64fiq.us-east-1.rds.amazonaws.com

Port
5432

Networking

Availability Zone
us-east-1a

VPC
IAP-vpc (vpc-01e89686aff50c0d1)

Subnet group
private subnet group

Subnets
subnet-04fbd607140bcd56c
subnet-065bfbf5b6d099374

Network type
IPv4

Security

VPC security groups
rds-ec2-1 (sg-03a37ddc999f32e6a)
Active

Publicly accessible
No

Certificate authority Info
rds-ca-rsa2048-g1

Certificate authority date
May 26, 2061, 00:34 (UTC+01:00)

DB instance certificate expiration date
December 09, 2025, 19:33 (UTC+00:00)

Figure 34 – RDS summary

Instance

Configuration

DB instance ID
postgres-database-1

Engine version
16.3

RDS Extended Support
Disabled

DB name
mydb

License model
Postgresql License

Option groups
default:postgres-16 In sync

Amazon Resource Name (ARN)
arn:aws:rds:us-east-1:533267319380:db:postgres-database-1

Resource ID
db-FDEGSWSALQE3EAFONL7EU7XKQQ

Created time
December 09, 2024, 19:34 (UTC+00:00)

DB instance parameter group
default.postgres16 In sync

Deletion protection
Disabled

Architecture settings
Non-multitenant architecture

Instance class

Instance class
db.t3.micro

vCPU
2

RAM
1 GB

Availability

Master username
postgres

Master password
****

IAM DB authentication
Not enabled

Multi-AZ
No

Secondary Zone
-

Storage

Encryption
Not enabled

Storage type
General Purpose SSD (gp2)

Storage
20 GiB

Provisioned IOPS
-

Storage throughput
-

Storage autoscaling
Enabled

Maximum storage threshold
1000 GiB

Storage file system configuration
Current

Performance Insights

Performance Insights enabled
Turned on

AWS KMS key
aws/rds

Retention period
7 days

Figure 35 – RDS Configuration