universidade de aveiro
deti departamento de eletrónica, telecomunicações e informática

ES - 2024/2025 - **IAP Project Report**

To Do List

Task Planner App

13/12/2024

108317 Miguel Miragaia

Content

1 Introduction 3
1.1 Project Overview 3
1.2 Key Features 3
2 Development Process 4
2.1 Tools and Technologies Used 4
2.2 Development Workflow 4
3 Agile Development Process 4
3.1 Agile Methodology Overview 4
3.2 Definition of Ready 5
3.3 Definition of Done 5
3.4 Sprints 5
3.5 User Stories and Acceptance Criteria 6
..... 7
3.6 Jira and Github Integration 7
4 AWS Architecture 8
4.1 Walkthrough of Development Process 8
4.2 Diagram 12
5 Data Flow 12
5.1 Authentication Flow 12
6 Swagger Documentation 13
7 Conclsuion 13
8 Bibliography 13

Table of Acronyms

EC2	Elastic Compute Cloud
ECR	Elastic Container Registry
RDS	Relational Database Service
VPC	Virtual Private Cloud
US	User Story
CORS	Cross-Origin Resource Sharing
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
AZ	Availability Zone

Table of Figures

Figure 1- Sprint 2 Review & Retrospective at Github 6
Figure 2- Sprint 3 Review & Retrospective at Github 6
Figure 3- Task Management Epic at Github 7
Figure 4- Edit Task User Storie at Github 7
Figure 5 - Burndown Chart Sprint 4 8
Figure 6- VPC Network 9
Figure 7- AWS Security Groups 10
Figure 8- AWS Target Groups 10
Figure 9- AWS Architecture – Very Bad Implemeted and Structure due to Time Constraints 12
Figure 10- Authentication Flow 12

1 Introduction

1.1 Project Overview

This report outlines the development and deployment of a task management application that integrates several AWS services for scalable and secure infrastructure. The application utilizes AWS Cognito for user authentication, AWS EC2 and Elastic Load Balancer for hosting, RDS for database management, and Secrets Manager for secure credential storage. It supports task creation, categorization, and management via backend APIs hosted in the cloud. Agile development methodologies and tools such as Jira and GitHub were employed to streamline the development workflow, ensuring iterative progress and effective project tracking.

1.2 Key Features

- **Authentication**: Utilized AWS Cognito IDP for secure authentication and registration, including support for token exchange and user session management.
- **Task Management**: Features include task creation, categorization, editing, and deletion, all managed through secure backend APIs.
- **Scalability**: Leveraged AWS services like EC2 for hosting, Elastic Load Balancers for distributing traffic, and API Gateway for API management to ensure scalable and robust infrastructure.
- **Security**: Used AWS Secrets Manager to securely store sensitive credentials, ensuring data protection and compliance. Enabled HTTPS communication across all endpoints to safeguard user interactions.
- **Deployment**: Frontend and backend applications were containerized using Docker and deployed via AWS ECR and EC2 instances. PostgreSQL database was hosted on AWS RDS for reliable and consistent storage solutions, supporting transactional integrity and optimized performance.

2 Development Process

2.1 Tools and Technologies Used

- **Frontend**: Developed using HTML, JavaScript, and CSS.
- **Backend**: Implemented with Spring Boot.
- **Database**: PostgreSQL hosted on AWS RDS.
- **Cloud Services**:
    - VPC, Subnets, Security Groups, EC2, ECR, Elastic Load Balancer, API Gateway, AWS Cognito, Secrets Manager, RDS.
- **Version Control**: GitHub served as the repository for tracking changes in the codebase.
- **Project Management**: Jira was used for planning and managing sprints effectively.

2.2 Development Workflow

The development workflow was structured around Agile principles, emphasizing iterative progress and responsiveness to changes. It was followed a weekly sprint-based approach, where each sprint had defined goals and deliverables. Tasks were prioritized using Jira, ensuring clear visibility and accountability.

3 Agile Development Process

3.1 Agile Methodology Overview

Agile methodology was implemented to maintain a structured and iterative development approach. Each sprint was planned with clear goals, deliverables, and user stories that contributed to incremental progress. Work was divided into manageable tasks with specific acceptance criteria, ensuring a consistent development pace.

At the end of each sprint, a review was conducted to evaluate the outcomes against the sprint goals. This included testing implemented features, identifying challenges, and refining plans for the next sprint. Retrospectives were also carried out to assess what went well, what didn't, and what could be improved, allowing for continuous enhancement of the development process and product.

3.2 Definition of Ready

Formatted as "As a (...), I want (...) so that (...)".
Short and self-explanatory.
Defined story points.
Acceptance criteria formatted as "Given (...) when (...) then (...)".
Followed the INVEST methodology.
Estimated priority, determined by the position in the backlog.

3.3 Definition of Done

Developed.
Documented.
Compliant with Acceptance Criteria.
Merged into the central branch (dev).

These definitions were stated in the GitHub README of the project from the beginning, ensuring clarity and consistency in task management and completion standards.

3.4 Sprints

At the end of each sprint, a **Sprint Review and Retrospective** was conducted and published at https://github.com/Miragaia/ES-Individual-Project/tree/dev/docs/Sprints . These included:

- **Sprint Reviews**:
  - Evaluated the work completed against the sprint goals.
  - Summarized completed user stories and tasks, aligning them with their respective epics.
- **Retrospectives**:
  - Identified challenges and areas for improvement.
  - Listed story points categorized into "Done," "In Progress," and "To Do."
  - Highlighted improvements for the next sprint, such as addressing unresolved issues or refining processes.

This process ensured incremental progress, continuous feedback, and alignment with project objectives, ultimately enhancing both the product and the development workflow.
Here are some examples:

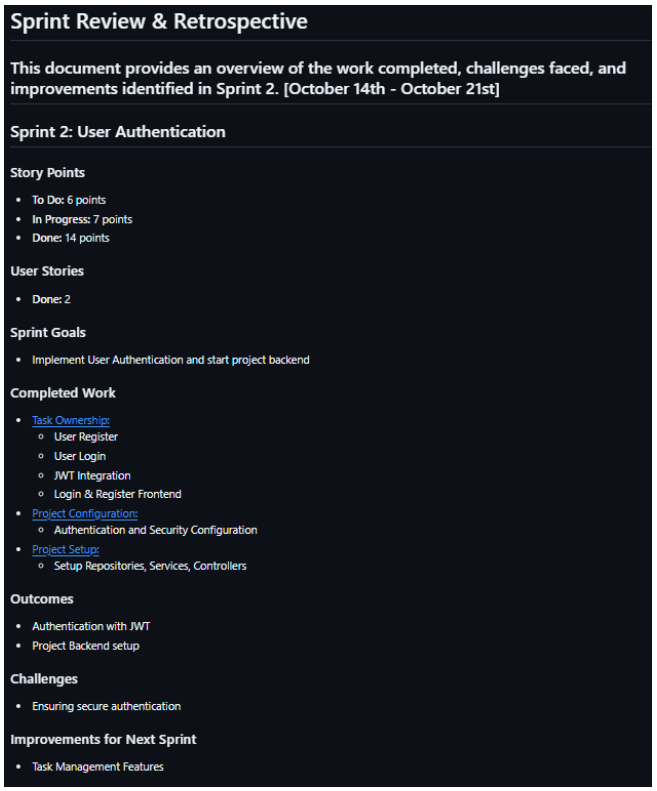

Figure 1- Sprint 2 Review & Retrospective at Github

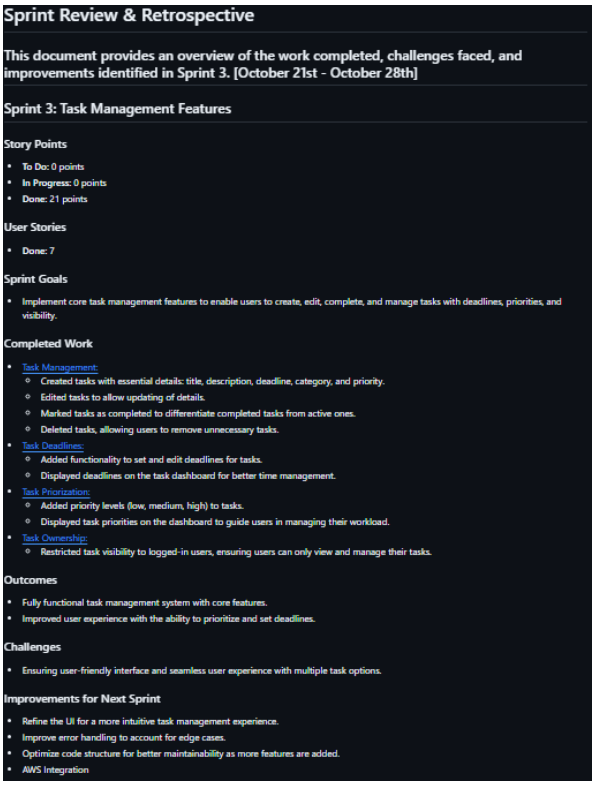

Figure 2- Sprint 3 Review & Retrospective at Github

3.5 User Stories and Acceptance Criteria

User stories were created in **Jira** to provide a structured framework for development. Each story was then linked to corresponding **GitHub branches**, ensuring traceability and alignment with the project's epics.

Epics, which group related user stories, were also documented in both Jira and GitHub (https://github.com/Miragaia/ES-Individual-Project/tree/dev/docs/Epics). Each epic included:

- **Overview**: A high-level description of the feature or functionality.
- **Goals**: The objectives the epic aimed to achieve.
- **User Stories**: A list of all stories associated with the epic.

Individual user stories followed a consistent format and included:

- **Summary**: A concise statement of the story's purpose.
- **User Story**: Formatted as *"As a [user], I want [action], so that [benefit]"*.
- **Acceptance Criteria**: Specific conditions under which the story would be considered complete, formatted as:
    - *Given [initial state], when [action], then [outcome].*
- **Implementation Details**: Key technical or functional details required for development.

Here are some examples:

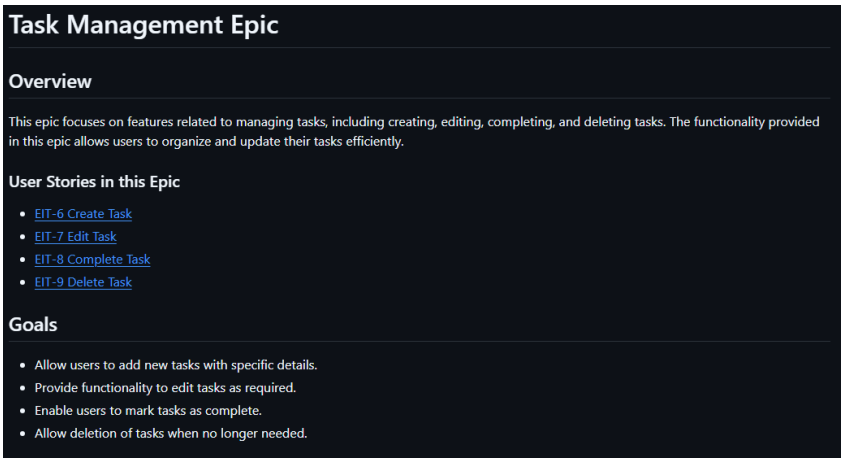

Figure 3- Task Management Epic at Github

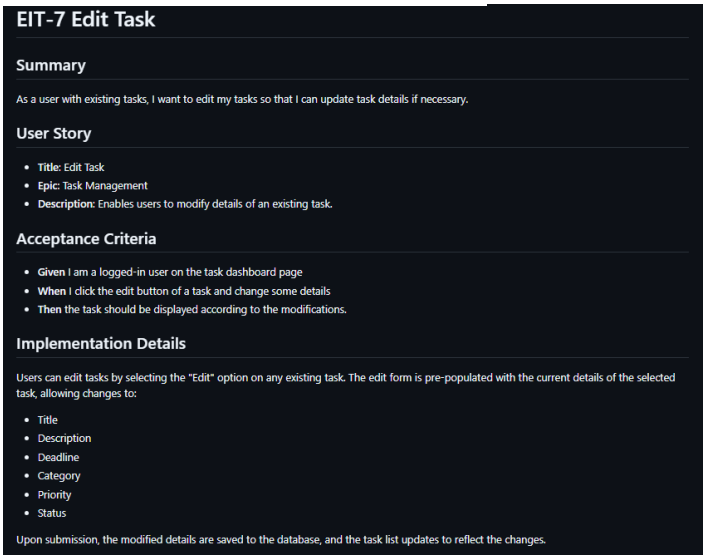

Figure 4- Edit Task User Storie at Github

3.6 Jira and Github Integration

To streamline project management and version control, **Jira** (https://es-ip-2425.atlassian.net/jira/software/projects/EIT/boards/2/timeline) and **GitHub** (https://github.com/Miragaia/ES-Individual-Project) were tightly integrated. This integration ensured seamless tracking of development progress, linking code changes directly to project tasks, and maintaining transparency in the workflow.

Jira Integration:

- User stories, tasks, and bugs were created and tracked in Jira.
- Each user story was associated with a specific **Jira issue** using unique identifiers (e.g., EIT-47).
- Sprint boards in Jira organized tasks into categories such as *To Do*, *In Progress*, and *Done*, visually representing progress at any given time.
- **Burn-down charts** in Jira provided a clear overview of remaining work, ensuring that sprint timelines were on track.

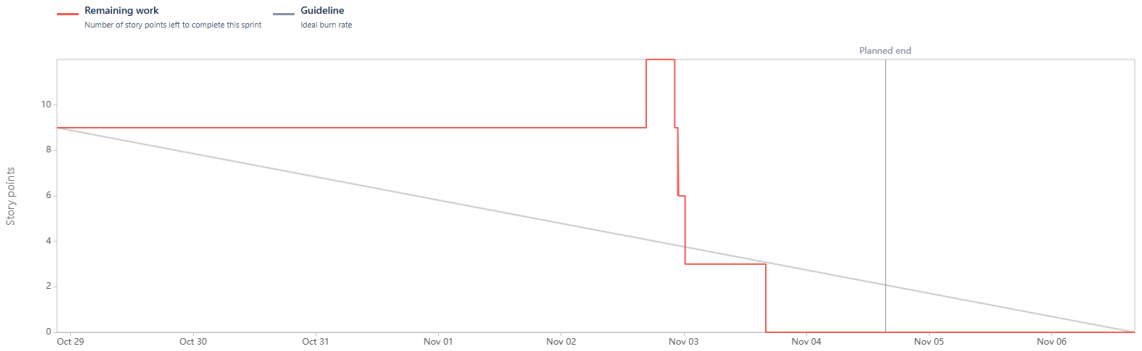

Figure 5 - Burndown Chart Sprint 4

GitHub Workflow:

- Feature branches were created in GitHub for each Jira issue, named according to the Jira ticket ID (e.g., EIT-47-RDS-Configuration).
- Pull requests (PRs) from feature branches were submitted and linked back to their respective Jira tasks for review.

Branching Strategy:

- A **feature branching model** was used:
  - **Feature Branches** (EIT-XX-US-Name): Created for individual tasks, bugs or user stories.
  - **Development Branch** (dev): Served as the integration branch where completed features were merged after passing review and testing.
  - **Main Branch** (main): Held stable, production-ready code. Merge from dev to main occurred only after the last successful (although at the end I know that I should have done it after each Sprint Review and Retrospective).
- Each branch underwent a **pull request (PR) process**, where code was reviewed for adherence to acceptance criteria, coding standards and met the **Definition of Done**.

4 AWS Architecture

4.1 Walkthrough of Development Process

1. Initial Development and Dockerization

- **Frontend Development:**
  - The UI was built using HTML, CSS, and JavaScript with functionality for authentication, task and category management, and was encapsulated into a Docker container using Nginx.
- **Backend Development:**
  - Built and containerized with **Docker** a Spring Boot application to handle API calls for task management, authentication, and category operations.
  - RESTful endpoints were structured under paths such as /api/auth, /api/tasks, and /api/categories.

- **Database Setup:**
    - Developed initial database schemas using PostgreSQL, that were encapsulated into a Docker container

2. AWS Integration

Virtual Private Cloud (VPC):

- **VPC Name:** ToDoVpc
- **IPv4 CIDR:** 10.10.0.0/16
- **Subnets:**
    - Two public subnets (10.10.0.0/26, 10.10.0.64/26) for Internet-facing resources.
    - Two private subnets (10.10.0.128/26, 10.10.0.192/26) for backend and database.
- **Routing:**
    - **Internet Gateway**: todo-internet-gateway attached to the VPC.
    - **Route Tables**: Public and private route tables for routing traffic appropriately.
        - Public Route Table associated with public subnets, routes all 0.0.0.0/0 traffic through the Internet Gateway.
        - Private Route Tables for secure communication between backend and database.

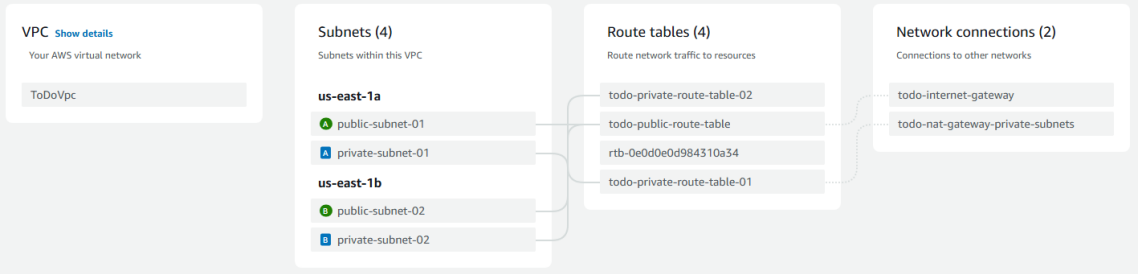

Figure 6- VPC Network

Security Groups:

- **Frontend EC2 Security Group (ec2-frontend-sg):**
    - **Inbound Rules:**
        - HTTP (80) from anywhere (0.0.0.0/0).
        - HTTPS (443) from anywhere (0.0.0.0/0).
        - SSH (22) restricted to the administrator's IP.
    - **Outbound Rules:** Allow all traffic.
- **Backend EC2 Security Group (ec2-backend-sg):**
    - **Inbound Rules:**
        - HTTP (8080) from the Load Balancer.
    - **Outbound Rules:** Allow all traffic.
- **Load Balancer Security Group (todo-alb-sg):**
    - **Inbound Rules:**
        - HTTP (80), HTTPS (443) from anywhere (0.0.0.0/0).
    - **Outbound Rules:** Forward to backend EC2.
- **RDS Security Group (rds-database-sg):**

- **Inbound Rules:**
  - PostgreSQL (5432) from private subnets only.

Name	Security group ID	Security group name	VPC ID	Description
–	sg-092d4c4a174c86c78	rds-database-sg	vpc-025ca0741866a20f1	Security group for RDS database
SSH-Only-SG-hE1H...	sg-0bd5f247068de13ef	SSH-Only-SG-hE1HWeZA	vpc-025ca0741866a20f1	Security group allowing SSH access only
SSH-Only-SG-BeZJ...	sg-04af4234d5c8f4eff	SSH-Only-SG-BeZJ6tPJ	vpc-025ca0741866a20f1	Security group allowing SSH access only
–	sg-0f8bbdd3f0a2b0015	todo-alb-sg	vpc-025ca0741866a20f1	Http and https for ALB
–	sg-01a054365afb1d8a6	ec2-backend-sg	vpc-025ca0741866a20f1	launch-wizard-1 created 2024-12-07T...
–	sg-01c15ac274cb921e2	default	vpc-025ca0741866a20f1	default VPC security group
–	sg-0156f6b654d13b370	ec2-frontend-sg	vpc-025ca0741866a20f1	launch-wizard-1 created 2024-12-07T...

Figure 7- AWS Security Groups

Elastic Compute Cloud (EC2)

- **Frontend EC2 Instance:**
  - **Name:** EC2-frontend
  - **Instance Type:** t2.micro
  - **Public IP:** 44.223.86.210
  - **Purpose:** Hosts the frontend using Docker and Nginx.
- **Backend EC2 Instance:**
  - **Name:** EC2-backend
  - **Instance Type:** t2.micro
  - **Public IP:** 54.152.249.200
  - **Purpose:** Hosts the Spring Boot backend using Docker.

Interaction: Both EC2 instances are registered as targets in their respective target groups for the Load Balancer.

Load Balancers:

Two Application Load Balancers distribute incoming traffic.

- **Frontend Load Balancer (todo-alb):**
  - Routes:
    - HTTP (80) and HTTPS (443) traffic to the frontend target group (frontend-target-group).
    - Routing Rules:
      - / → Frontend target group.
- **Backend Load Balancer (backend-alb):**
  - Routes:
    - HTTP (80) traffic to the backend target group (api-backend-target-group).

Target groups (3) Info

Filter target groups

Name	ARN	Port	Protocol	Target type	Load balancer	VPC ID
api-backend-target-group	arn:aws:elasticloadbalancin...	80	HTTP	Instance	backend-lb	vpc-025ca0741866a20f1
backend-target-group	arn:aws:elasticloadbalancin...	8080	HTTP	Instance	todo-alb	vpc-025ca0741866a20f1
frontend-target-group	arn:aws:elasticloadbalancin...	3000	HTTP	Instance	todo-alb	vpc-025ca0741866a20f1

Figure 8- AWS Target Groups

Authentication with Cognito

- **Cognito Configuration:**
  - App Client Name: ToDoList
  - Authentication and registration are managed through Cognito's pre-built UI.
  - Tokens (ID, Access, Refresh) are retrieved post-authentication and exchanged via the ***"/api/auth/exchange"*** backend endpoint.
- **Backend Integration:**
  - Cognito client ID, secret, and endpoints are securely stored in AWS Secrets Manager.
  - The backend verifies tokens, retrieves user claims, and registers new users if they don't already exist in the system database.

Amazon RDS

- **Instance Name:** todolist-db
- **Database Engine:** PostgreSQL
- **Configuration:**
  - Hosted in a private subnet.
  - Connected to the backend EC2 via private IP.
  - Secrets stored in AWS Secrets Manager.

Note: Should be deployed across multiple AZ but due to cost constraints was not possible.

API Gateway

- **Name:** todo-backend-lb-api
- **Routes:** Configured to forward /api/* requests to the backend Load Balancer.
- **Proxy Integration**
  - **Backend Integration:** The API Gateway is configured as a proxy to forward requests to the backend Load Balancer (backend-lb).
- **CORS Policies:** Reflect backend CORS configurations to allow communication from https://es-ua.ddns.net.

ECR (Elastic Container Registry)

- **Frontend Repository:** frontend-repo
- **Backend Repository:** backend-repo
- Docker images are built locally, pushed to ECR, and pulled by EC2 instances during deployment.

How These Components Work Together

1. **Frontend Access:**
   - Users access the application via https://es-ua.ddns.net routed through the frontend Load Balancer to the frontend EC2.
2. **Backend Communication:**
   - API requests (e.g., /api/auth/exchange) from the frontend are routed via the Load Balancer to the backend EC2.
3. **Database Connection:**

- Backend communicates securely with RDS for storing and retrieving tasks.

4. **Authentication:**
   - Users authenticate through Cognito, and tokens are validated by the backend.

Challenges

1. Configuring CORS policies for seamless frontend-backend communication.
2. Debugging Load Balancer health checks to ensure traffic routing.
3. Deployment was very time consuming, building the images locally, push to ECR, and pulled by EC2 instances takes a long time. Terraforms should have been used to avoid it.

4.2 Diagram

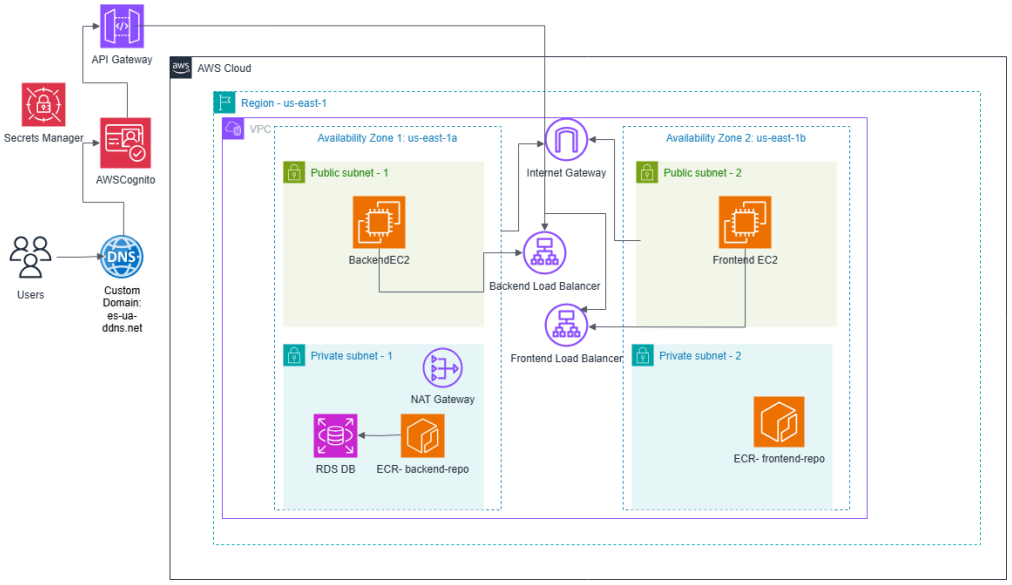

Figure 9- AWS Architecture – Very Bad Implemeted and Structure due to Time Constraints

5 Data Flow

5.1 Authentication Flow

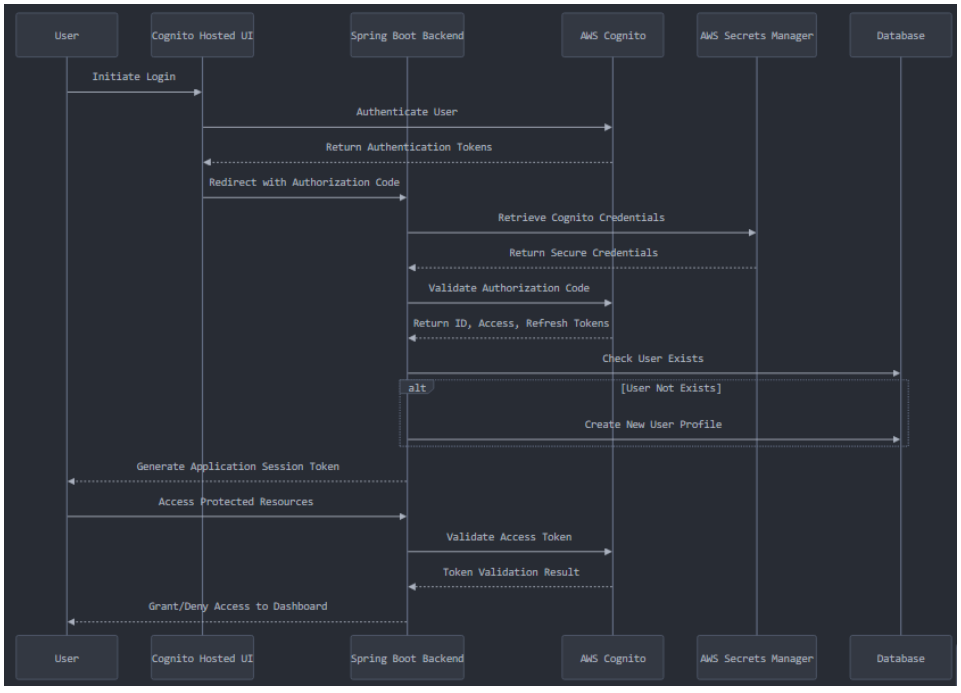

Figure 10- Authentication Flow

6 Swagger Documentation

To ensure the API is well-documented and accessible, Swagger was used to provide a clear and interactive interface for exploring and testing the application's endpoints. The complete API documentation was exported to a file named ***"api-docs.json".***

7 Conclsuion

The development of this project provided a comprehensive understanding of building a scalable task management application using AWS services such as EC2, RDS, Cognito, and API Gateway. Agile methodologies ensured an iterative approach, addressing challenges like CORS configuration, API Gateway integration, and Docker deployments while fostering continuous learning. This foundation enables future enhancements like multi-AZ database deployment and advanced analytics, emphasizing the value of cloud technologies in creating secure, reliable applications.

8 Bibliography

AWS Documentation: Amazon Web Services. Official Documentation for AWS Services. Retrieved from https://docs.aws.amazon.com

OpenAI ChatGPT: OpenAI. ChatGPT: Language Model for Assistance and Research. Retrieved from https://openai.com

draw.io: Diagrams.net - Online Diagram Software and Flowchart Maker. Retrieved from https://app.diagrams.net

Stack Overflow: Community for Developers. Various solutions retrieved from https://stackoverflow.com

Jira Documentation: Atlassian. Jira Software Documentation for Agile Project Management. Retrieved from https://www.atlassian.com/software/jira

GitHub Documentation: GitHub. Platform for Version Control and Collaboration. Retrieved from https://docs.github.com

Swagger/OpenAPI: Swagger. API Documentation Tools and OpenAPI Standards. Retrieved from https://swagger.io