**Chapter – 1:-**

The objectives of my dissertation are as follows:

• To develop an AI-powered onboarding assistant for navigating extensive private code repositories.

• To implement efficient embedding and retrieval methodologies for accurate code and documentation understanding.

• To build a scalable microservices backend integrated with real-time webhook-driven updates.

• To create a user-friendly interface using a Visual Studio Code extension.

• To evaluate improvements in developer productivity and code comprehension.

From the objectives, the progress till mid-semester are:-

1. Implemented a React Application as frontend for the server process
2. Integrated GitHub OAuth Framework for Authorization of Front-End Application
3. Developed APIs to communicate between Front-End and Back-End
4. Created a process to download the list of files from GitHub Repository
5. Created a Publisher process to publish the downloaded files into the RabbitMQ Queue
6. Created a Consumer process to consume the files from RabbitMQ Queue in order to perform the Embeddings for those files
7. Generated the Text Embeddings for the files published from RabbitMQ queue and stored the generated text embeddings into the ChromaDB.
8. Deployed the entire application into the AWS Cloud which is required for the future scope of the dissertation.

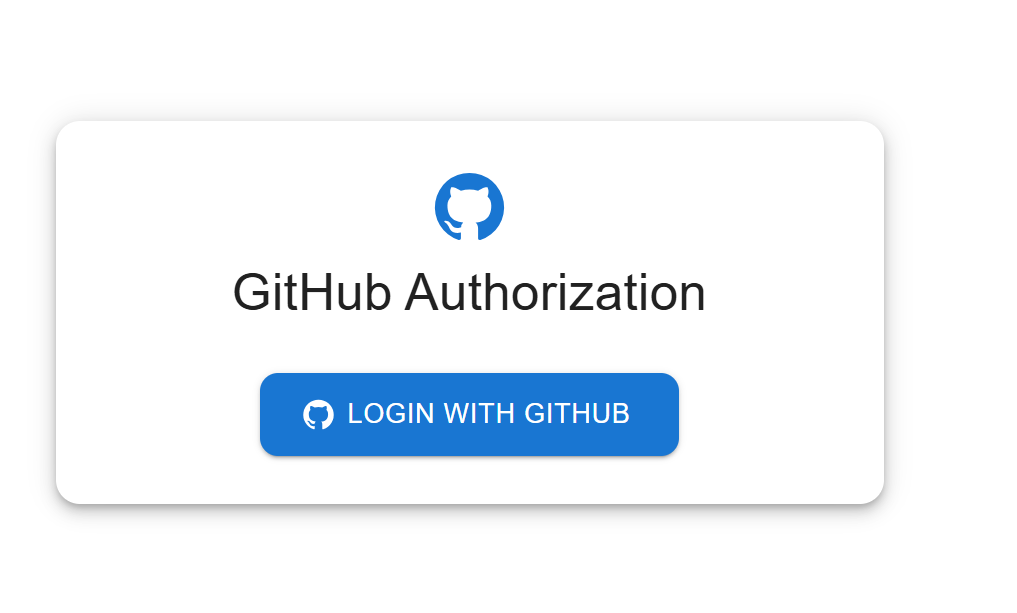
**Chapter-2:-**

In this chapter, it mainly focuses on the creation of Front-End React Application which mainly deals

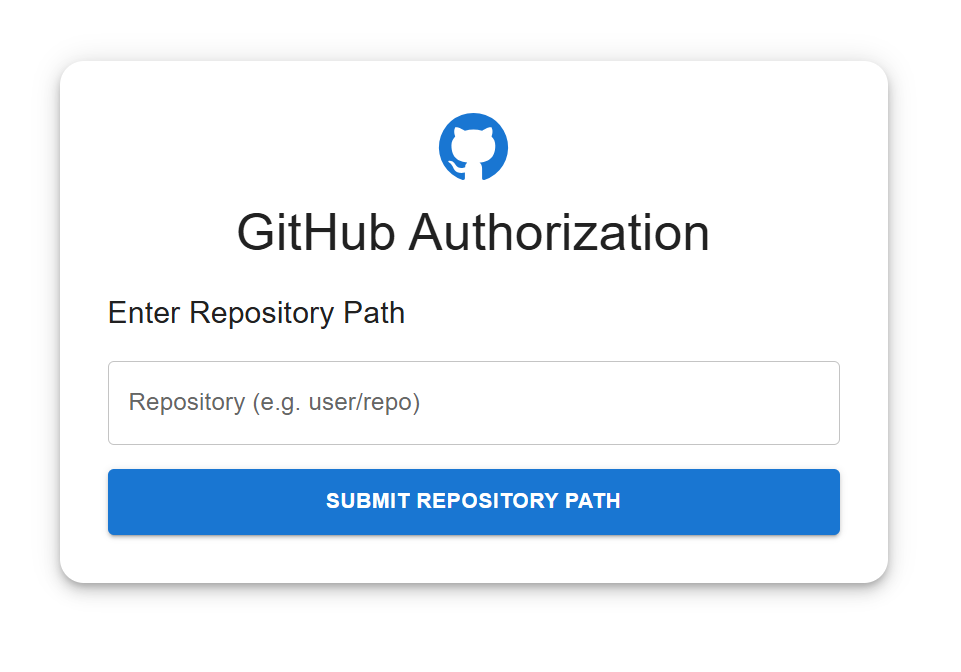
on the Login Page and Repository Configuration Page

The below are the screens that I have implemented as part of the server-side process.

1. **Screen-1:-** As the react app starts, this is the starting page.



1. **Screen-2:-** Once user is authorized, the below is the screen, where user can configure the GitHub Repository, on which the Repository needs to be vectorized.

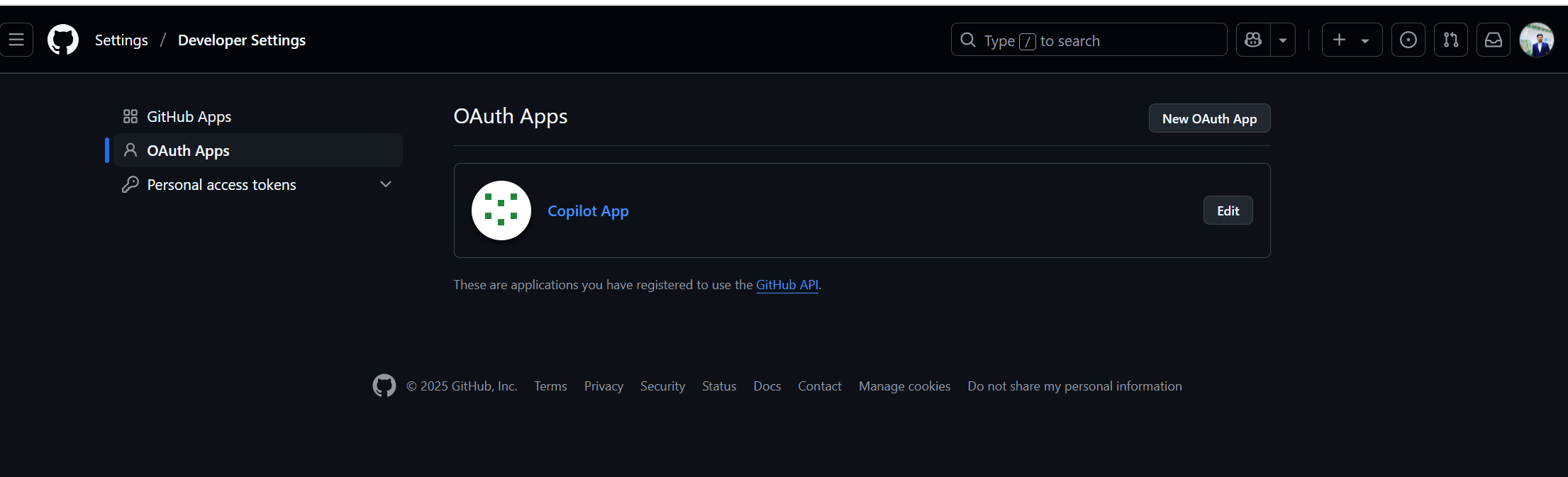


User can configure the Repository path in the above text box, on which the vectorization needs to be done.

**Chapter-3:-**

In this chapter, it mainly focuses, on the authorization of GitHub Credentials.

1. I have created a GitHub OAuth Application called Copilot Application
2. In that application, where we can configure HomePage URL and Authorization Callback Url
3. The below are the screenshots for reference.



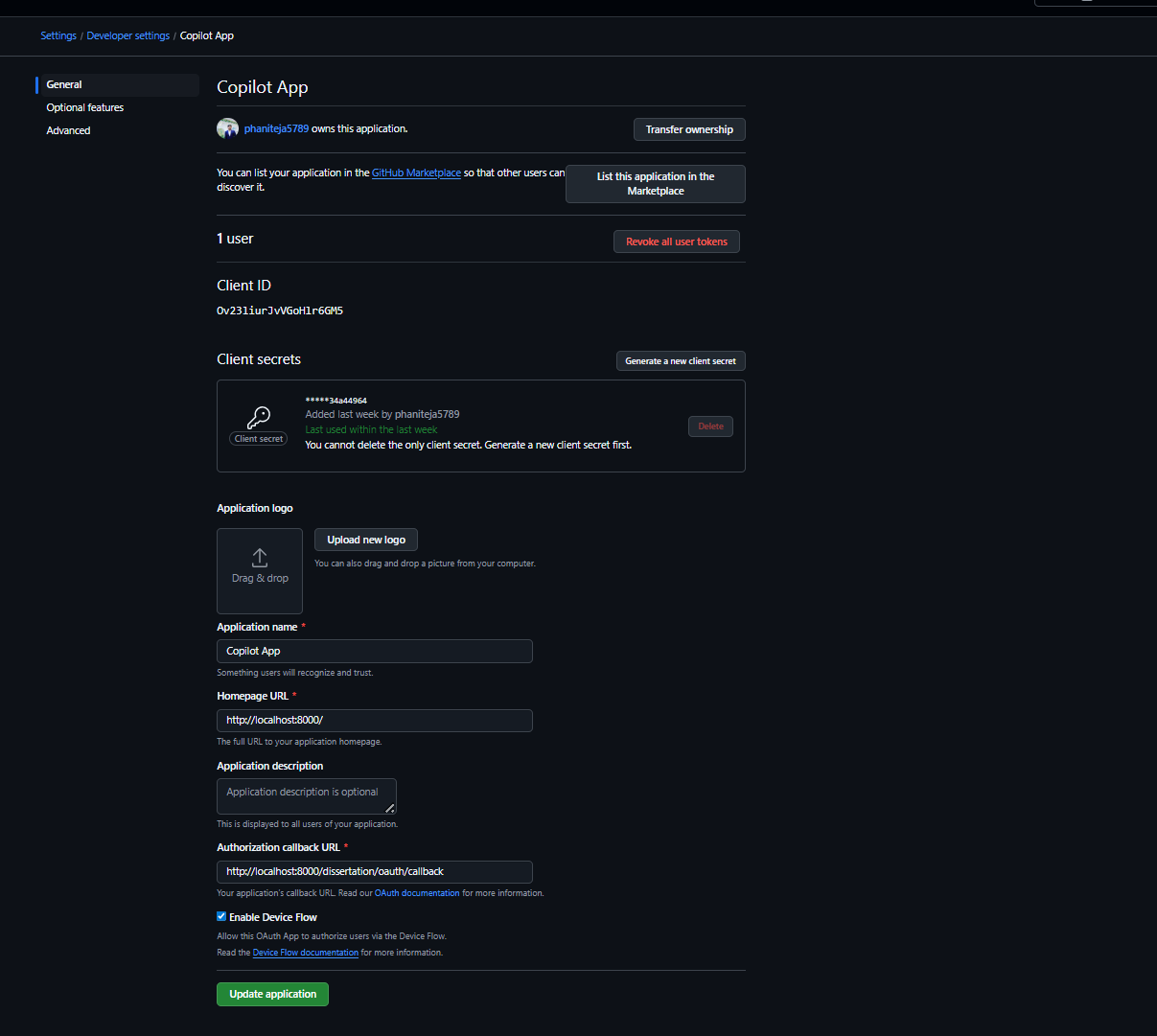
Created an OAuth App called Copilot App.

From this Application, we can fetch the Client ID and Client Secret ID which will be used for Authorization.

In this application, we need to configure the HomePage URL and Authorization Callback URL

HomePageURL 🡺 From which endpoint it receives the request for Authorization

Authorization Callback URL 🡺 Once the credentials has been authenticated and authorized, the endpoint it redirects to further continue with the application.



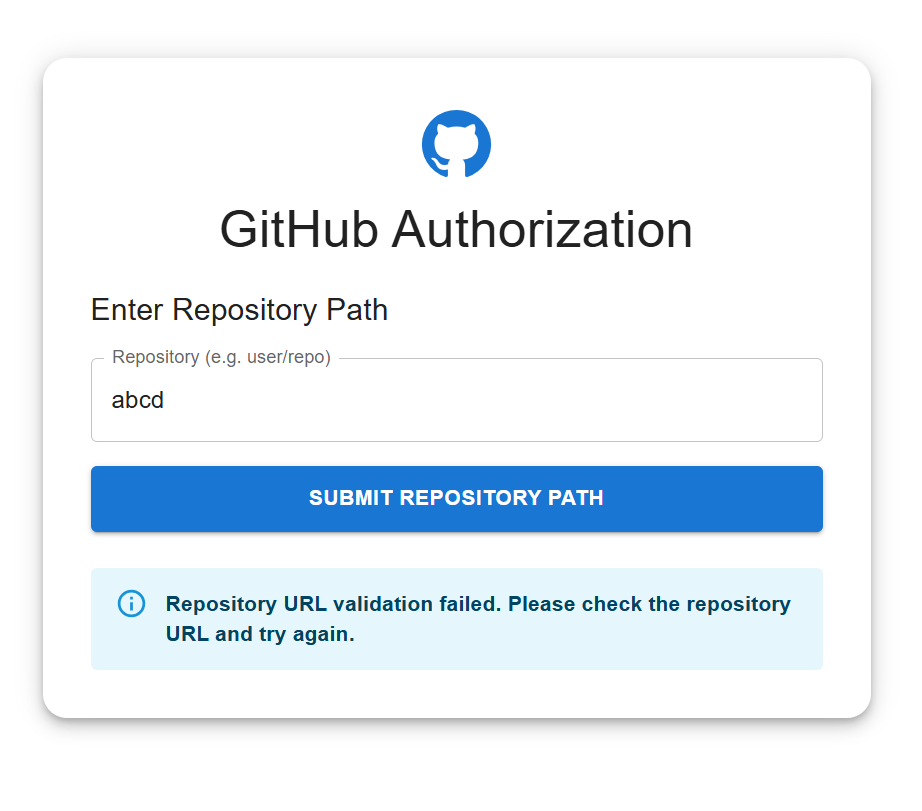
The main usage of the GitHub OAuth Application is we shouldn’t valid a user credentials. We are redirecting to the GitHub for Authorization.

**Chapter-4:-**

In this chapter, I have created Back-End APIs for communication between Front-End and Back-End and GitHub Authorization.

As part of Implementation, I have created 4 main APIs

1. /dissertation/login 🡺
   1. This API is mainly used to receive the Request from React Application.
   2. Post receiving the Request, it will redirect to the GitHub OAuth App for Authorization.
2. /dissertation/oauth/callback 🡺
   1. This API is used to communicate between GitHub OAuth Application, which I have created in earlier chapter and backend server post its authorization.
   2. Once the request for Authorization is successful, GitHub OAuth Application will redirect back to this endpoint with the GitHub OAuth Token which is needed for further process on the Repository level.
   3. For example, reading the information from the GitHub Repository, Creation of Web-Hooks on the Repository.
3. /dissertation/set\_repo 🡺
   1. This API is used to set the GitHub Repository name in the React Application Text-Box.
4. /dissertation/repo/configuration🡺
   1. This API will be used once the GitHub Repository is given by the user.
   2. In this API, the Request will contain certain parameters as part of Request Body.
      1. GitHub OAuth Token 🡺 This OAuth Token will be used for validating permissions on the GitHub Private Repository Path.
      2. Private GitHub Repository Path 🡺 The Repository on which the vectorized needs to be done.
   3. As part of this API, the sequence of execution as follows.
      1. Creation of JWT from the OAuth Token 🡺 The JWT is useful in order to verify the user has valid permission to access this /dissertation/repo/configuration API or not. Because not all users require a permission to do the vectorization. Since it’s a costlier operation
      2. Once the User has valid permissions to access to this resource. Then the GitHub Repository will be validated. Whether the given path is a valid GitHub URL or not. If it’s not a valid URL, then the error message is shown on the React Application as below.



* + 1. Once the valid GitHub Repository URL has been given, the GitHub OAuth Token will be validated against the repository. Whether the user has permission to read the data from the repository or not.
    2. Once the user has valid permission to read the files from the repository. Then the files will be read from the Repository.

**Chapter-5:-**

In this chapter, the main focus is on reading the files from the provided Repository.

All the validations and permissions has been authenticated against the user OAuth Token.

Now from the GitHub Repository, I will be getting the list of files present in that Repository and Publishing the names of the files into the RabbitMQ Process.

**Chapter-6:-**

In this chapter, the main focus is on communicating between Back-End Server to the Process which is used for Vectorization.

I have created a separate Python Process called **Embeddings\_Publisher.py**

In this process, we will download the file into the local working directory.

Once the file is downloaded from the GitHub, we will publish the downloaded file into another RabbitMQ Channel.

**Chapter-7:-**

In this chapter, the main focus is on reading the files from the RabbitMQ channel and doing the vectorization on the downloaded file

I have created a separate Python Process called **Embeddings\_Consumer.py**

In this process, we will read the files from the RabbitMQ process published by the Embeddings\_Publisher.py

Once the file read by the process, we will do the Text Embeddings on the file and store the generated text embeddings into a Chromadb directory.

**Chapter-8:-**

In this chapter, the main focus is on creating the Dockerfiles for the process and deploying them into the AWS Cloud.

The main server side implementation contains 5 process.

1. React Application 🡺 The React Application which is running on Nginx Server
2. FastAPI Application 🡺 The FastAPI process which is running on UviCorn Server
3. RabbitMQ Process 🡺 The RabbitMQ process which is required for Asynchronous communication between Process
4. Embeddings\_Publisher.py 🡺 This is a Python Application which is used to publish the downloaded files from the GitHub Repository and publish the information into the RabbitMQ Queue.
5. Embeddings\_Consumer.py 🡺 This is a Python Application which is used to consume the files from the RabbitMQ Queue and Vectorize the file and store the Embedded result into the ChromaDB Collection.

The main reason of using the process are keeping in consideration about the Scalability.

Since the Embeddings\_Publisher and Embeddings\_Consumer takes time to process the files from the GitHub Repository.

Since, it’s a microservice based implementation, each service needs to be scaled up individually which has no dependency between the process.

So, that is the reason, moved everything into the Docker based Implementation.

So, there are total 5 process in this server-side implementation.

1. React Application 🡺 FrontEnd
2. FastAPI 🡺 BackEnd
3. RabbitMQ 🡺 Used for Asynchronous Communication between Process
4. Publisher 🡺 Publishes the information into the RabbitMQ Channel
5. Consumer 🡺 Consumes the information from the RabbitMQ Channel

For each process a separate Dockerfile has been created.

The created Dockerfiles are as below names.

1. Dockerfile.frontend 🡺 Which containerize the React Application and runs the Nginx Server
2. Dockerfile.backend 🡺 Which containerize the FastAPI endpoints and runs the Uvicorn Server
3. RabbitMQ 🡺 Which runs the RabbitMQ server
4. Dockerfile.publisher 🡺 Which containerize the Python Process responsible for Publisher
5. Dockerfile.consumer 🡺 Which containerize the Python Process responsible for Consumer

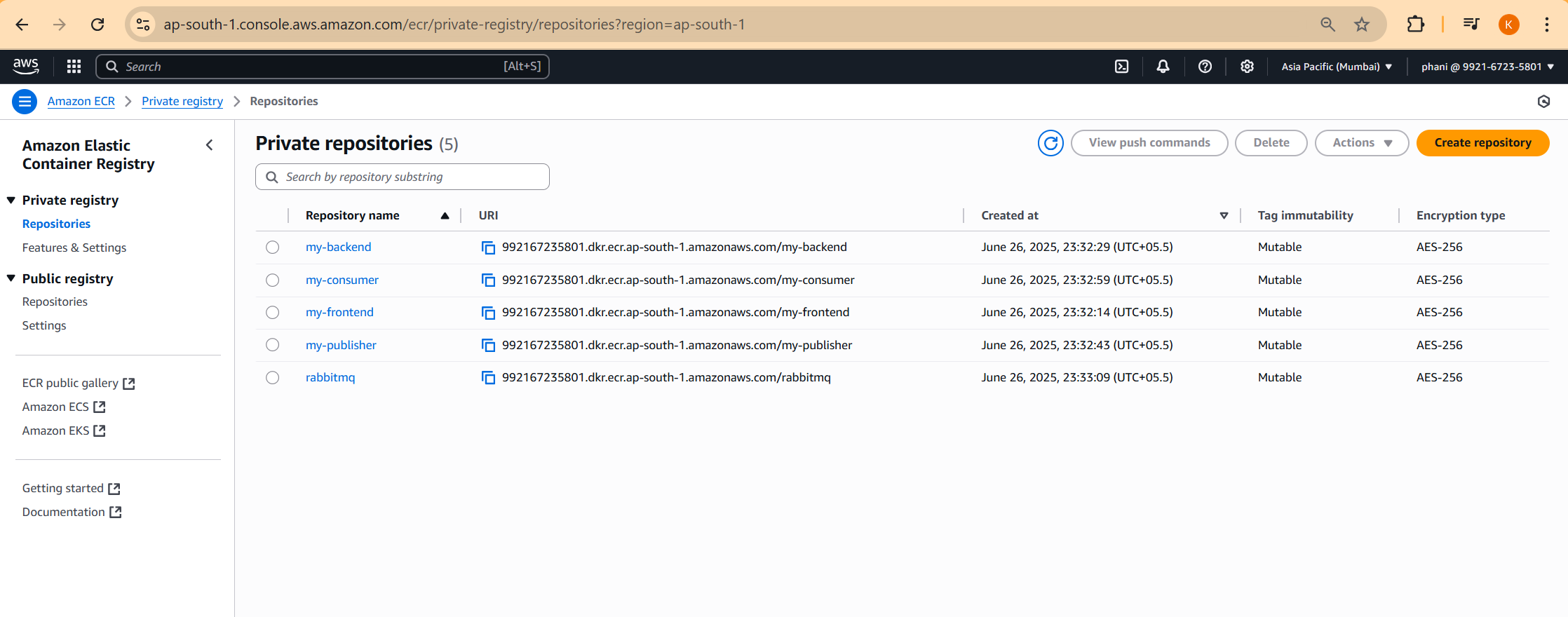
The below is the sequence of Execution once the Dockerfiles has been created.

1. Created the Docker Images for each process using the above mentioned Dockerfiles.
2. Created a Docker Network
3. Created a Docker container for the FrontEnd, BackEnd, RabbitMQ Process
4. Created a Docker Volume which is required for Publisher and Consumer containers.
5. Created a Docker Container for Publisher Process and stored the files into the Docker Volume. This is required, since its acts a Shared Directory between Publisher and Consumer. Since Publisher will write the files into the Shared Directory and Consumer will be reading the files from the Shared Directory.
6. Created a Docker container for Consumer Process, which reads the files from the Docker Volume.

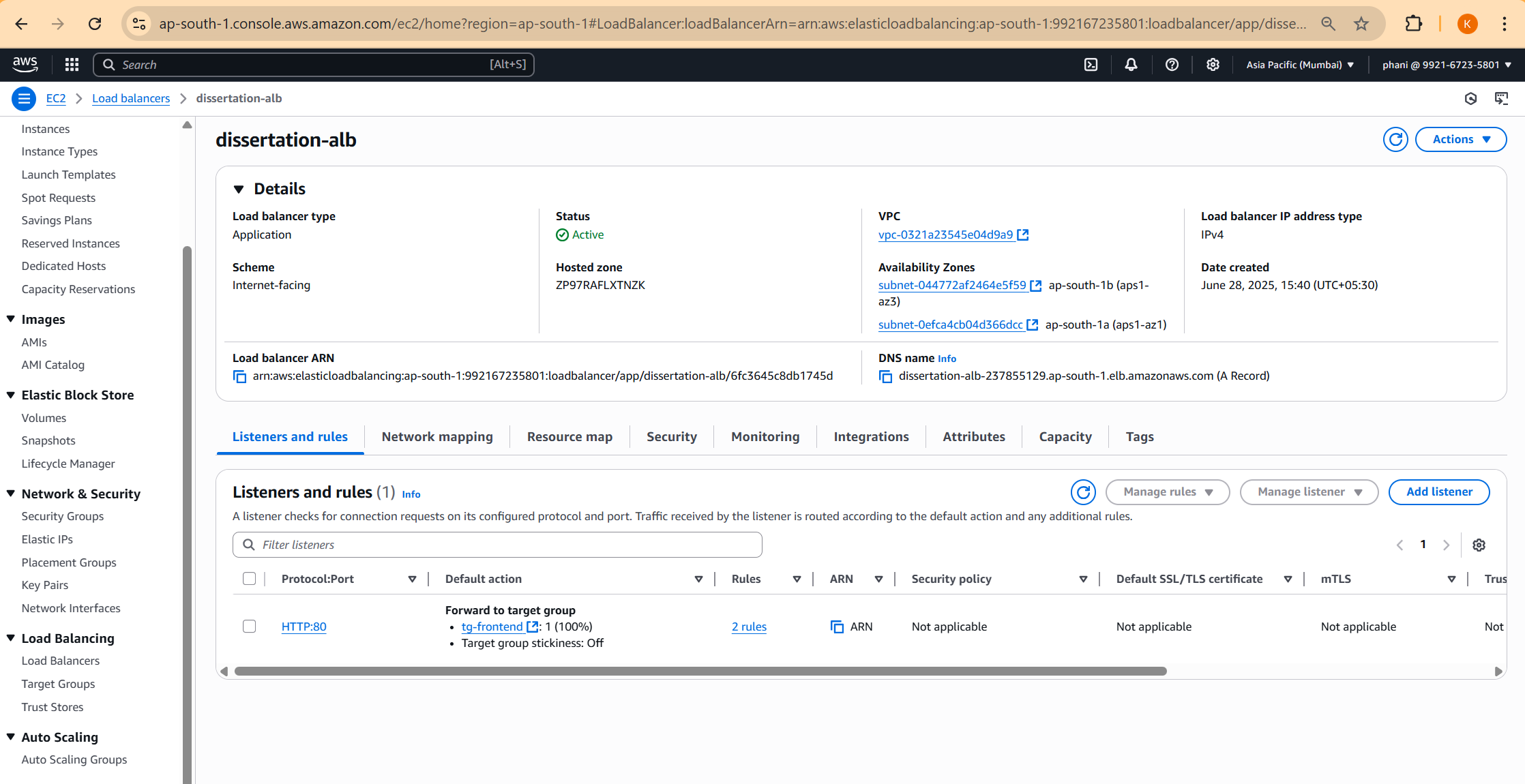
Now, deployed this entire process into the AWS.

The below AWS services are used as part of this deployment.

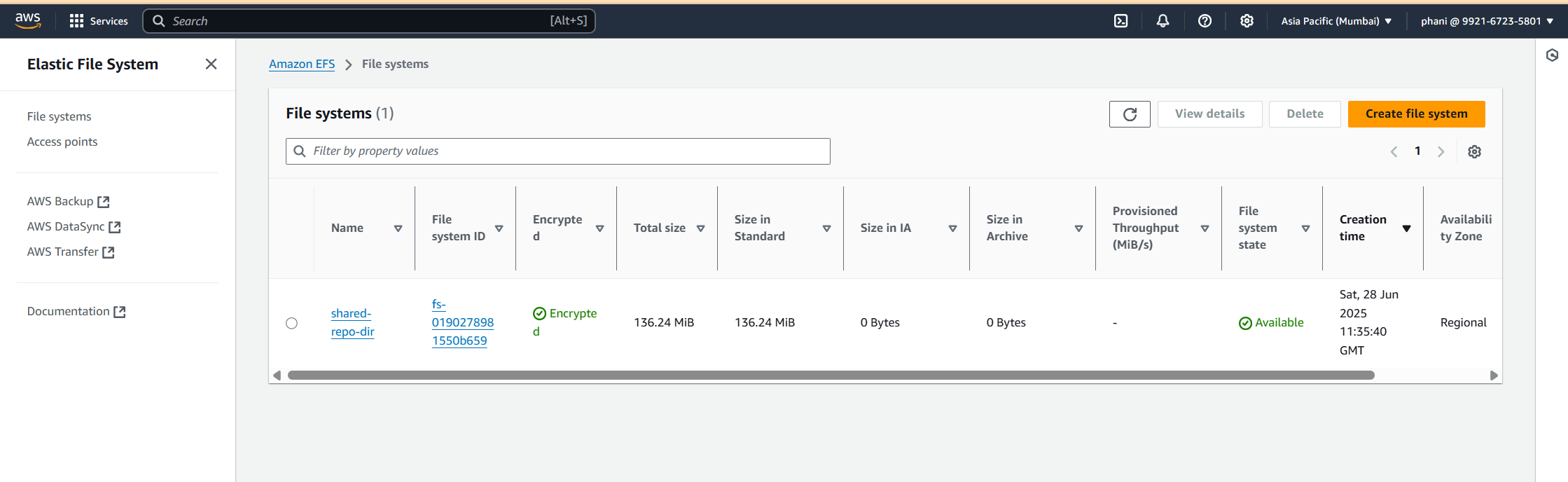
1. AWS ECR 🡺 Elastic Container Registry
   1. I have used this service in order to store the Docker Images which are created earlier.
   2. I have created 5 Repositories, where each repository stores each process Docker Image.
   3. Docker Image created from Dockerfile.frontend will be stored under “my-frontend” ECR Repository
   4. Docker Image created from Dockerfile.backend will be stored under “my-backend” ECR Repository.
   5. Docker Image created from RabbitMQ Image will be stored under rabbitmq ECR Repository.
   6. Docker Image created from Dockerfile.publisher will be stored under “my-publisher” ECR Repository
   7. Docker Image created from Dockerfile.consumer will be stored under “my-consumer” ECR Repository.



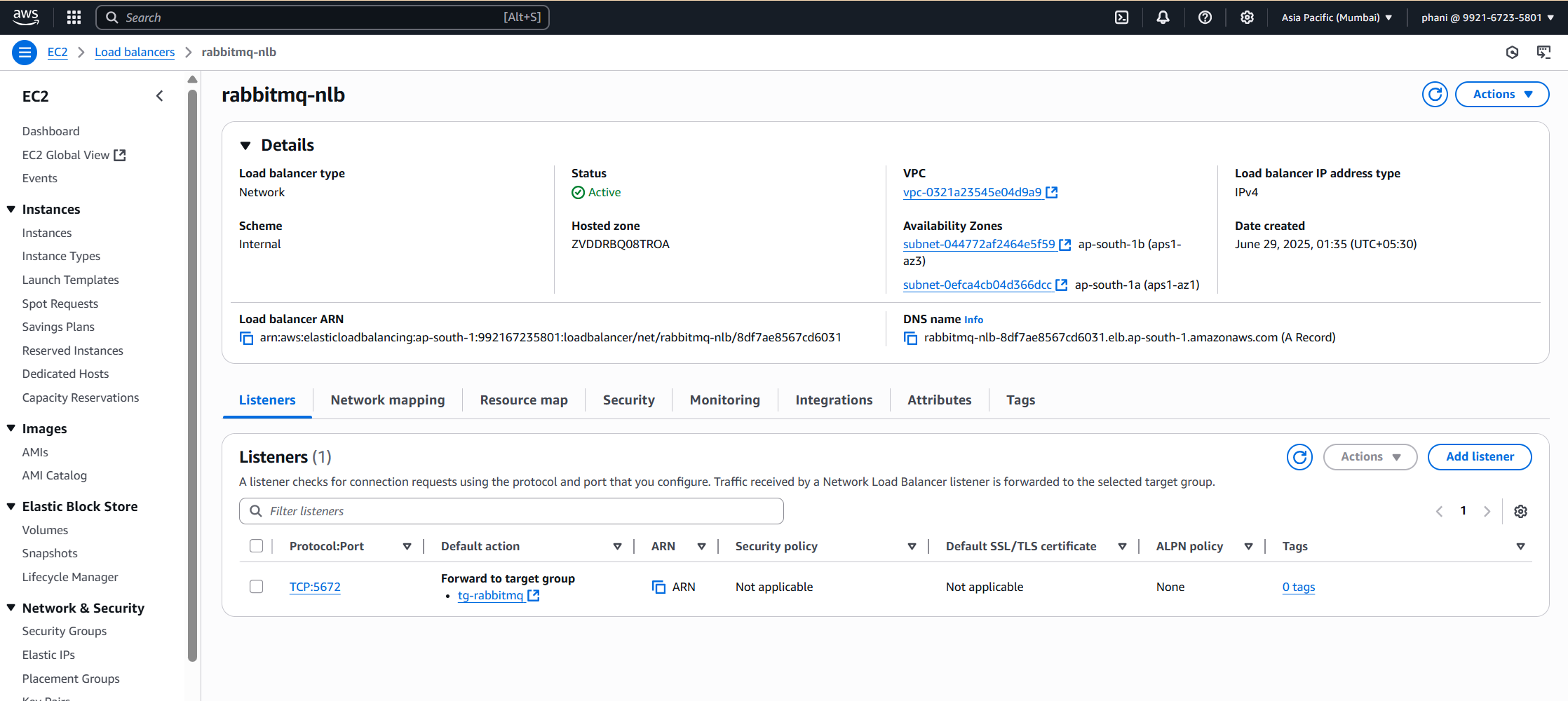
1. AWS ALB 🡺 Application Load Balancer
   1. Created an Application Load Balancer for deployment into the Internet.
   2. From the Application Load Balancer, we are getting the DNS Name which is the public URL which can be accessible throughout the Internet.
   3. The Application Load Balancer with name “dissertation-alb” has been created.
   4. Created a Target Group “tg-frontend” and “tg-backend” for both the frontend and backend since they will be exposing to the Public Internet.
   5. Created a Listener at the Port 80 to the Application Load Balancer and attached the tg-frontend target group
   6. Created a Rule (Path based) to the Listener Port 80 to the Application Load Balancer and attached tg-backend target group



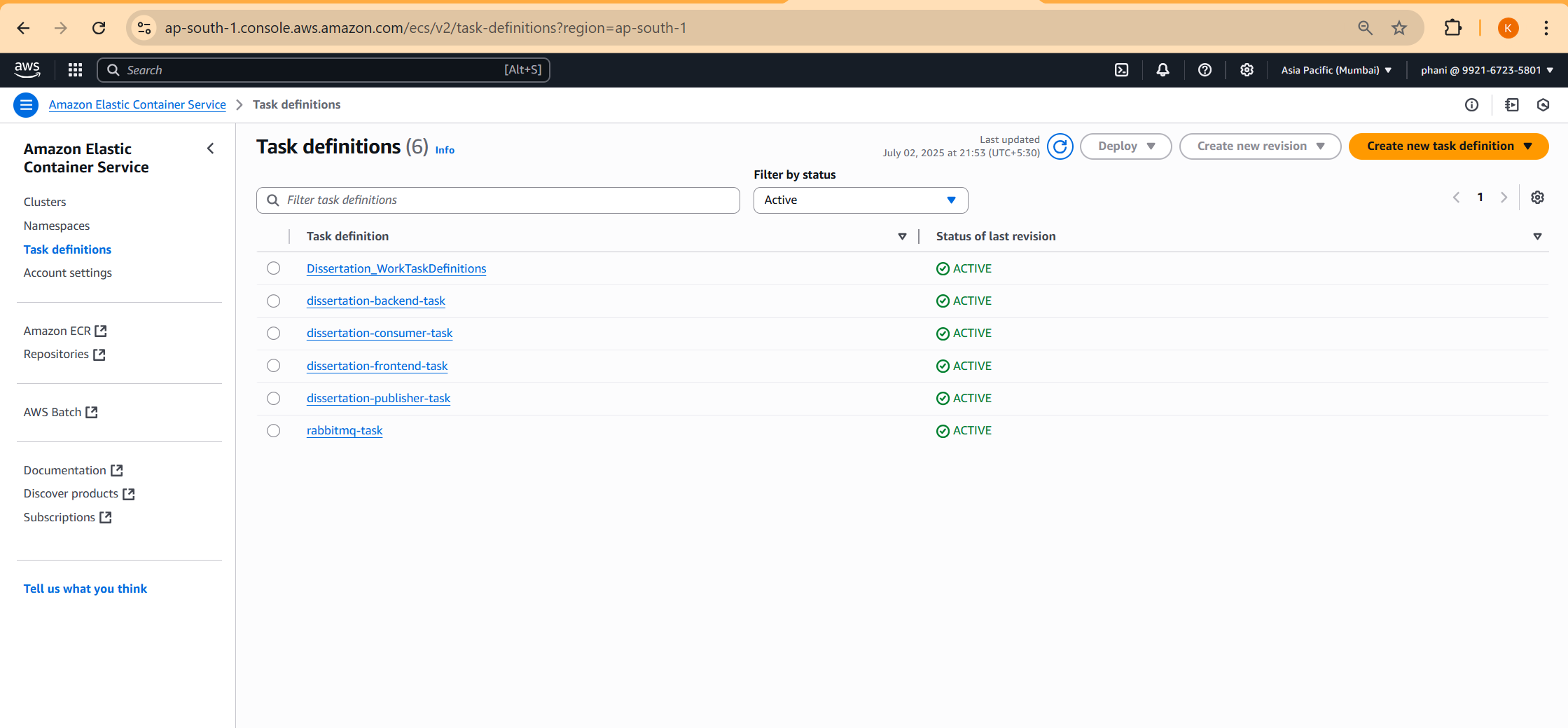
1. AWS EFS 🡺 Elastic File System
   1. Created a shared directory called “shared-repo-dir” which is required for Publisher and Consumer Process.



1. AWS NLB 🡺 Network Load Balancer
   1. Created a Network Load Balancer for the RabbitMQ service, since the RabbitMQ service is Internal inside VPC and not exposed to the outside public internet.
   2. The DNS name of Network Load Balancer will be used for backend, publisher, consumer process.

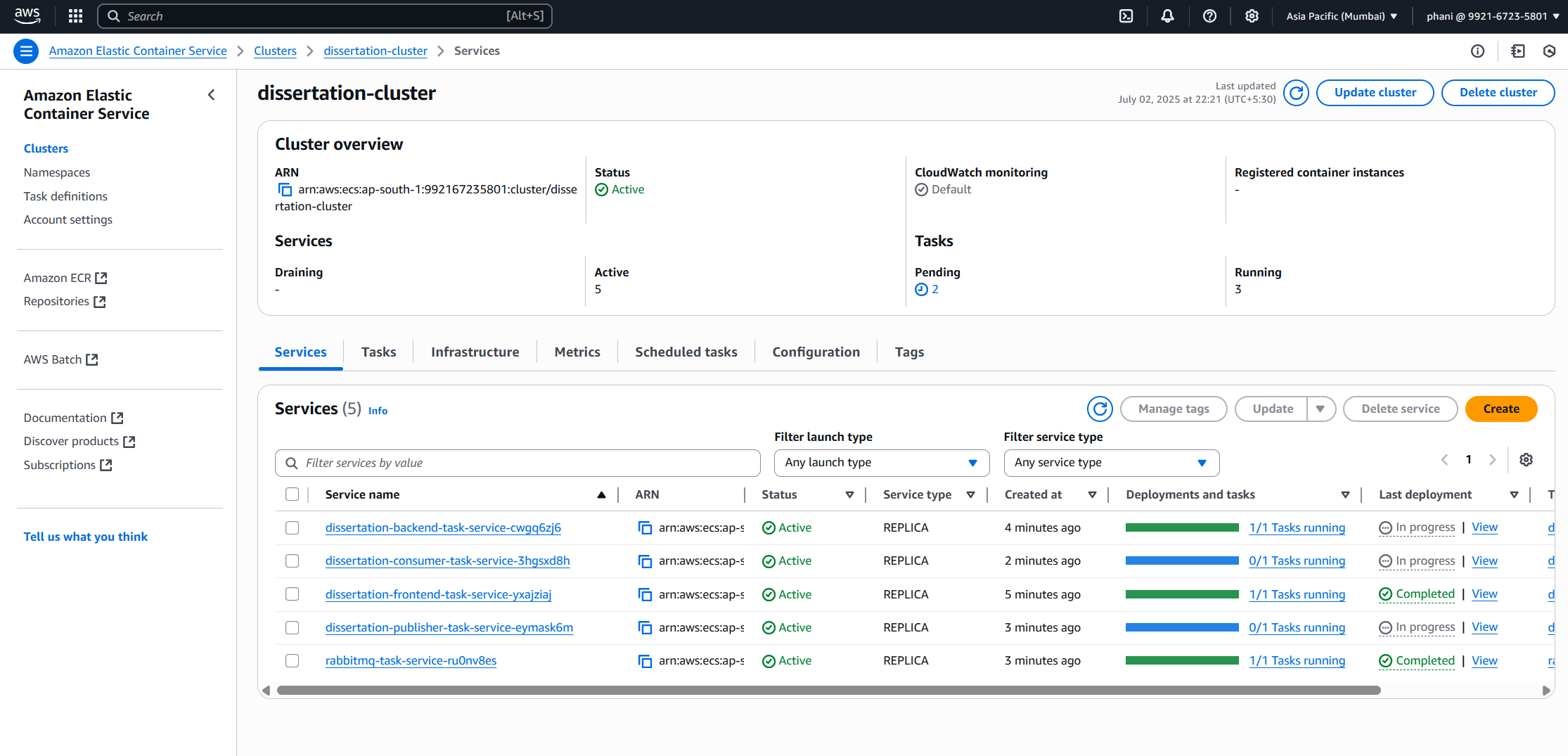


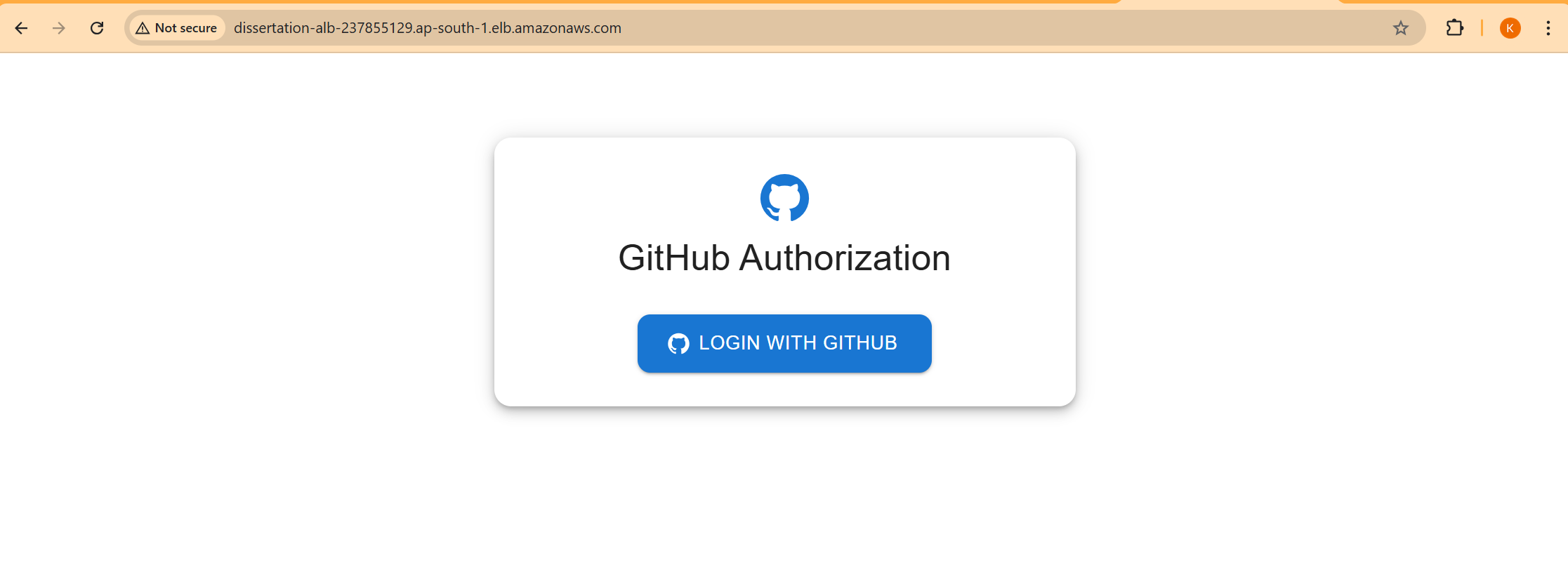
1. AWS ECS 🡺 Elastic Container Service with Fargate as Launch Type.
   1. In order to launch ECS containers, there are prerequisite things needs to be defined initially.
   2. Creation of ECS Cluster, where the ECS containers will be running.
   3. Once the ECS Cluster has been created. Next we have to create the Task Definitions for each repository.
   4. Created the Task Definitions for all the repositories.
   5. In the Task Definition file, we need to specify any Environment Variables used inside the Docker file and any EFS Volume used.

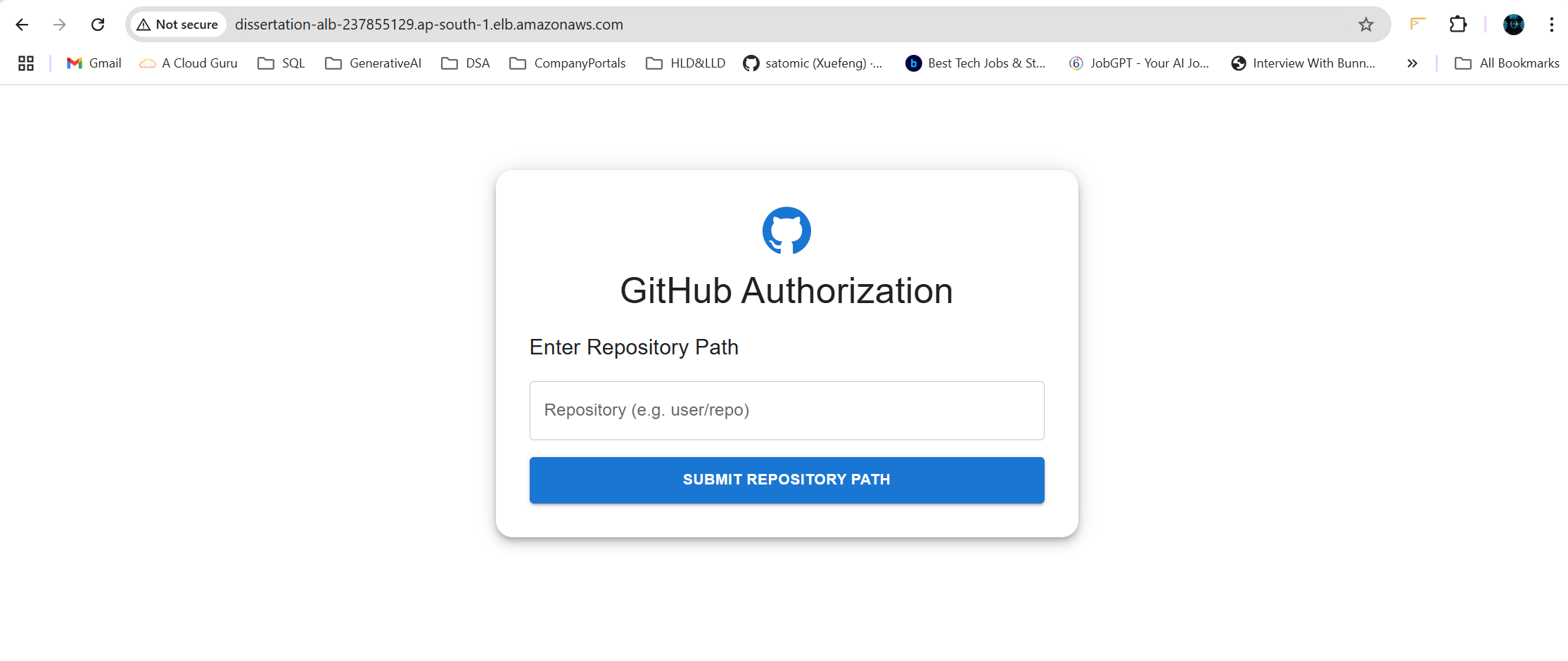


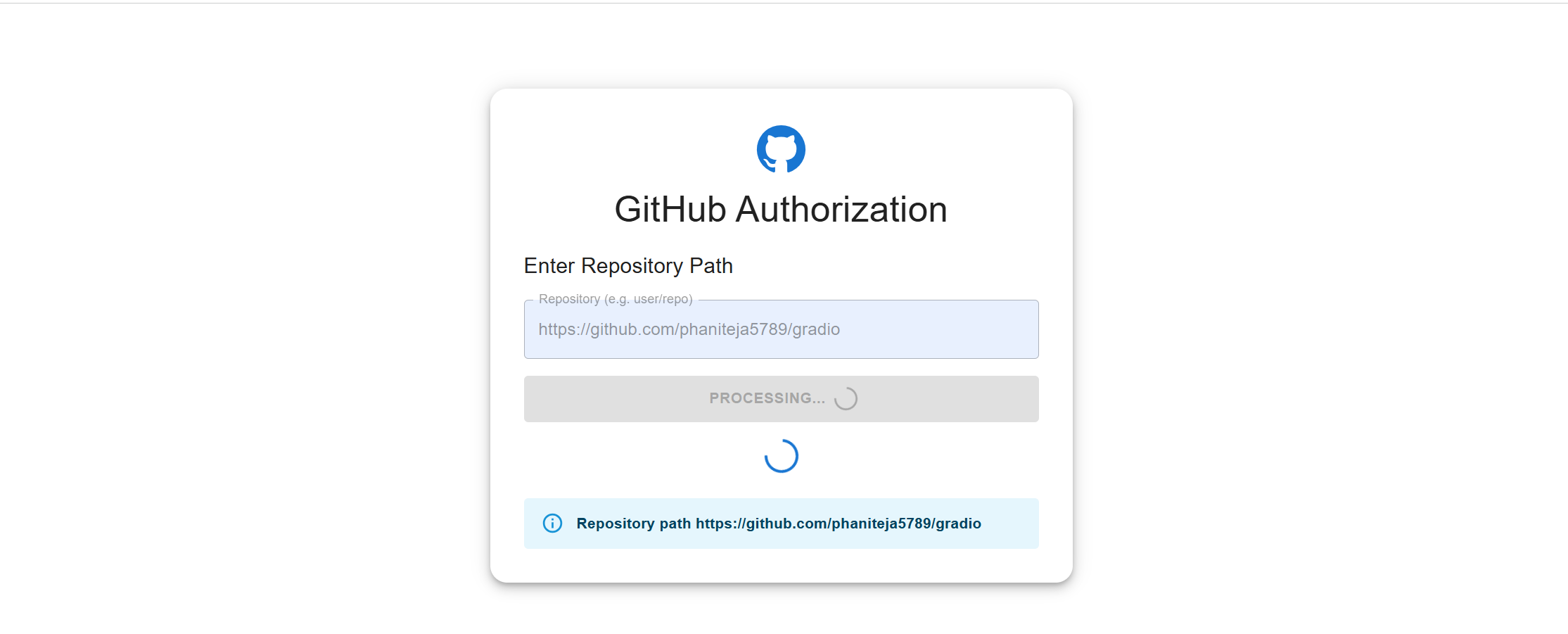
* 1. Created Task Definitions for each ECR Repository.
  2. Total 5 Images are created as part of Server-Side process and each Docker Image has been placed under different ECR Repository.
     1. Front-End Process
        1. Docker Image = my-frontend:latest
        2. ECR Repository = my-frontend
        3. ECS Task Definition = frontend-dissertation-task (which contains frontend Docker Image)
     2. Back-End Process
        1. Docker Image = my-backend:latest
        2. ECR Repository = my-backend
        3. ECS Task Definition = dissertation-backend-task (which contains backend Docker Image)
     3. RabbitMQ Process
        1. Docker Image = RabbitMq:3-management
        2. ECR Repository = rabbitmq
        3. ECS Task Definition = rabbitmq-task
     4. Publisher Process
        1. Docker Image = my-publisher
        2. ECR Repository = my-publisher
        3. ECS Task Definition = dissertation-publisher-task
     5. Consumer Process
        1. Docker Image = my-consumer
        2. ECR Repository = my-consumer
        3. ECS Task Definition = dissertation-consumer-task

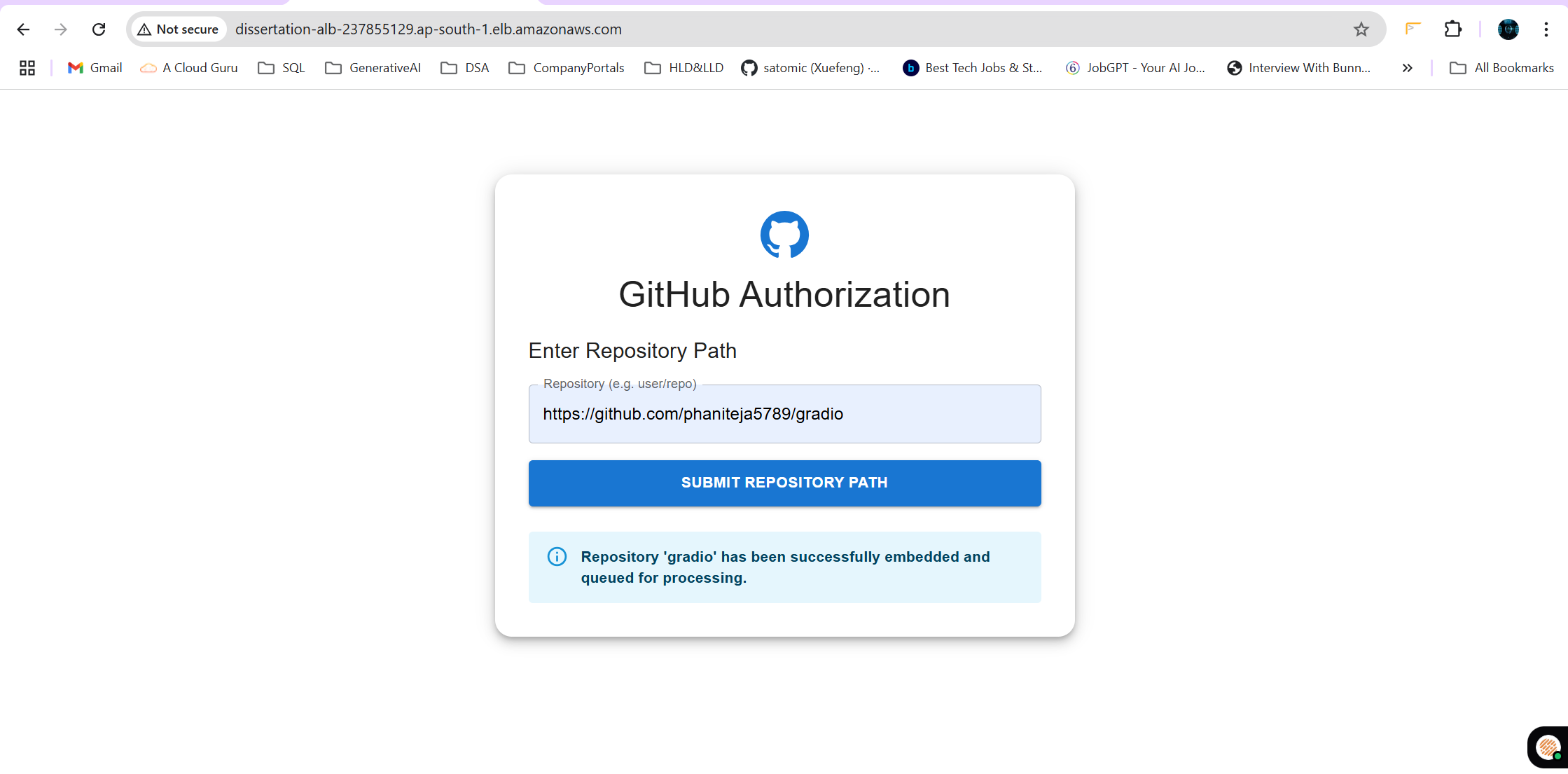
1. Once the ECS Task Definitions has been created. Next step is to create the ECS Services, where I have integrated the ALB and Target Groups to the Task Defintions.



1. Now the Application has been deployed into the internet.
2. The Application has been deployed in the ap-south-1 region of AWS with the DNS Name as “dissertation-alb-237855129.ap-south-1.elb.amazonaws.com”
3. 







Once the embeddings has been processed, the resultant embeddings will be stored inside the ChromaDB.