turnitin

2023da04339-renduchintala Phani Teja . .

2023da04339

S2-24_DSECLZG628T - Mid

S2-24_DSECLZG628T

Birla Institute of Technology and Science Pilani WILP Division

Document Details

Submission ID

trn:oid:::1:3290269465

Submission Date

Jul 5, 2025, 1:09 AM GMT+5:30

Download Date

Jul 5, 2025, 1:11 AM GMT+5:30

File Name

2023DA04339.pdf

File Size

1.4 MB

24 Pages

3,446 Words

22,090 Characters

1% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

Filtered from the Report

▸ Bibliography

▸ Quoted Text

Match Groups

4 Not Cited or Quoted 1%
Matches with neither in-text citation nor quotation marks

0 Missing Quotations 0%
Matches that are still very similar to source material

0 Missing Citation 0%
Matches that have quotation marks, but no in-text citation

0 Cited and Quoted 0%
Matches with in-text citation present, but no quotation marks

Top Sources

1%   🌐 Internet sources

0%   📖 Publications

0%   👤 Submitted works (Student Papers)

Match Groups

4 Not Cited or Quoted 1%
Matches with neither in-text citation nor quotation marks

0 Missing Quotations 0%
Matches that are still very similar to source material

0 Missing Citation 0%
Matches that have quotation marks, but no in-text citation

0 Cited and Quoted 0%
Matches with in-text citation present, but no quotation marks

Top Sources

1% ⊕ Internet sources

0% 📖 Publications

0% 👤 Submitted works (Student Papers)

Top Sources

The sources with the highest number of matches within the submission. Overlapping sources will not be displayed.

1	Internet

5dok.net **<1%**

2	Internet

buttondown.email **<1%**

3	Internet

www.coursehero.com **<1%**

Onboarding Code Companion: A Vectorized AI Assistant for Private Repositories

DISSERTATION

Submitted in partial fulfillment of the requirements of the

Degree : MTech in Data Science and Engineering

By

Renduchintala Phani Teja

2023DA04339

Under the supervision of

Shyamkumar Jha
(Lead Software Engineer)

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE

Pilani (Rajasthan) INDIA

(June, 2025)

Dissertation Abstract

In software development, understanding extensive and complex private codebases presents significant challenges, especially when dealing with multiple interconnected business components and numerous solution files. Developers, whether new to a project or tasked with extending existing functionalities, frequently encounter difficulties in navigating and effectively understanding these extensive structures, despite the availability of documentation. Existing solutions, such as Microsoft's Copilot, have limitations in context-awareness, primarily restricted by workspace boundaries, rendering them inadequate for comprehensively handling segmented repositories composed of multiple interlinked components.

This dissertation aims to develop the "Onboarding Code Companion," a proposed advanced AI assistant designed explicitly to streamline developer onboarding and improve understanding of large-scale private repositories. The planned approach involves utilizing cutting-edge embedding techniques from OpenAI and Hugging Face to semantically encode source code and associated documentation, creating structured and context-rich vector representations. These representations will be systematically stored in vector databases such as ChromaDB or FAISS, organized clearly by repository identifiers to facilitate accurate and contextually relevant retrieval.

The backend infrastructure will leverage FastAPI within a scalable microservices architecture. Real-time webhook events, activated upon code merges or updates, will trigger targeted, incremental updates to embeddings. This strategy ensures efficient processing by focusing solely on modified or newly introduced files, thus maintaining responsiveness and minimizing unnecessary computational overhead.

To provide an intuitive user experience, a custom Visual Studio Code extension using ReactJS will be developed, offering a user-friendly, chat-based interface. Developers will be able to submit queries in natural language directly through this interface, leveraging Retrieval-Augmented Generation (RAG) to retrieve relevant embeddings and generate precise, contextually informed responses via OpenAI's GPT-4. Additionally, integration with LangChain will manage conversational context effectively, enhancing the relevance and accuracy of interactions.

The anticipated outcomes include significant improvements in developer productivity, substantial reductions in onboarding time, and precise retrieval of information from continuously evolving codebases. By combining advanced natural language processing, deep learning methodologies, and semantic retrieval techniques, the "Onboarding Code Companion" aims to contribute meaningfully to the development of intelligent assistance tools, addressing vital comprehension and navigation challenges within modern software engineering contexts.

Key Words: Generative AI, Embeddings, Information Retrieval, ChromaDB, FAISS, LangChain, GitHub Webhooks, FastAPI, ReactJS, Visual Studio Code Extension, Retrieval-Augmented Generation, Software Onboarding Assistant

List of Symbols & Abbreviations

RAG	Retrieval-Augmented Generation
GPT-4	Generative Pre-trained Transformer version 4
VS Code	Visual Studio Code
ChromaDB	An Open-Source Vector Database for Embeddings
LangChain	Framework for Building Context-Aware LLM Applications
JSON	JavaScript Object Notation
IDE	Integrated Development Environment
GitHub	Web-Based Hosting Service for Version Control using Git
LLM	Large Language Model
Embedding	Vector representation of text/code for similarity search
Webhook	HTTP callback triggered by an event (e.g., code merge)
Microservices	Architectural style where components are loosely coupled
Repository	Storage location for software code and assets
Semantic Search	Search based on meaning rather than keyword matching

List Of Tables

List of Figures

Contents

1. Chapter 1 : Introduction

1.1. Title

Onboarding Code Companion: A Vectorized AI Assistant for Private Repositories

1.2. Scope

The scope of this dissertation is to design, implement, and evaluate an advanced AI system capable of assisting developers in comprehensively understanding and navigating complex private code repositories. The project will involve developing backend services, embedding generation and retrieval mechanisms, integration with webhook systems, and creating an interactive user interface via a Visual Studio Code extension.

1.3. Objectives mentioned in the abstract

- 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.

1.4. Objectives met till mid semester

a) **Development of React JS Web Application :** Implemented a React Application as frontend for the server process, which is required for the user to configure the repository path on which the Embeddings needs to be created.

b) **Integration with GitHub OAuth Framework:** Integrated GitHub OAuth Framework for Authorization of Front-End Application

c) **API Development:** Developed APIs to communicate between Front-End and Back-End

d) **Fetch List of Files from GitHub Repository:** Created a process to fetch the list of files from GitHub Repository

e) **Publisher Process for Generation of Text Embeddings:** Created a Publisher process to publish the downloaded files into the RabbitMQ Queue

f) **Consumer Process for Generation of Text Embeddings and Storing the Embeddings into the Vector Database:** Created a Consumer process to consume the files from RabbitMQ Queue in order to perform the Embeddings for those files. Generated the Text Embeddings for the files published from RabbitMQ queue and stored the generated text embeddings into the ChromaDB.

g) **AWS Deployment:** Deployed the entire application into the AWS Cloud which is required for the future scope of the dissertation.

2. Chapter 2 : Development of React JS Web Application

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.

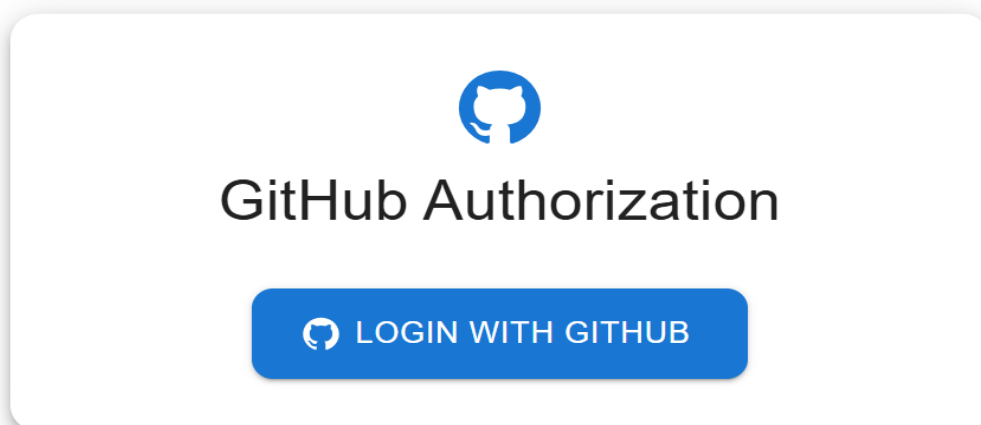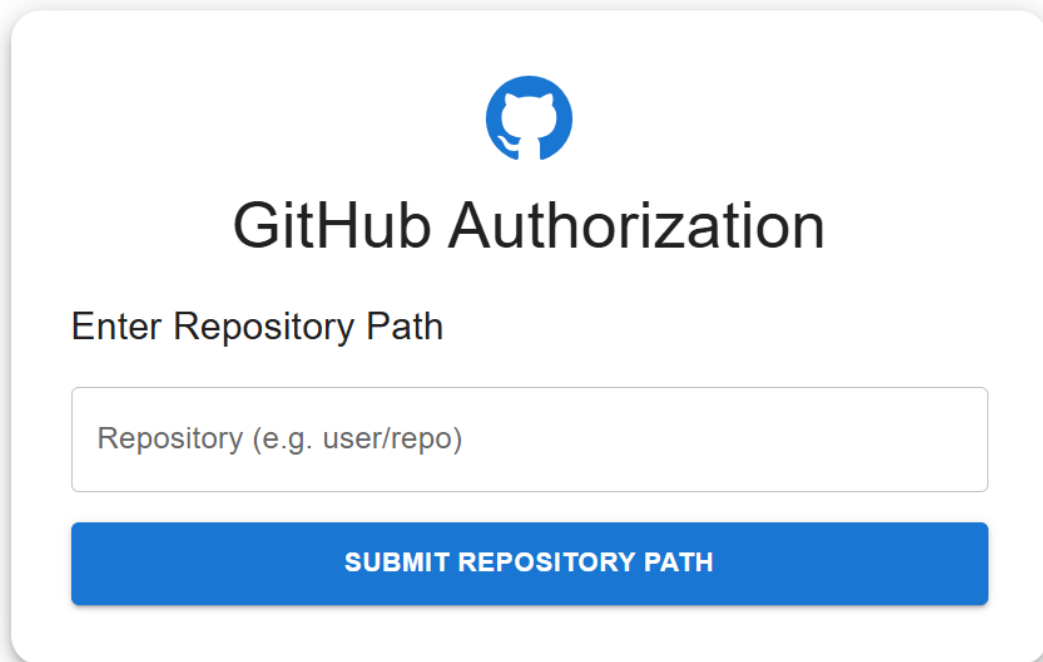Figure 1: React Application Login Page

2.) **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.

Figure 2: React Application Repository Configuration Page

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

3. Chapter 3 : Integration with GitHub OAuth Framework

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

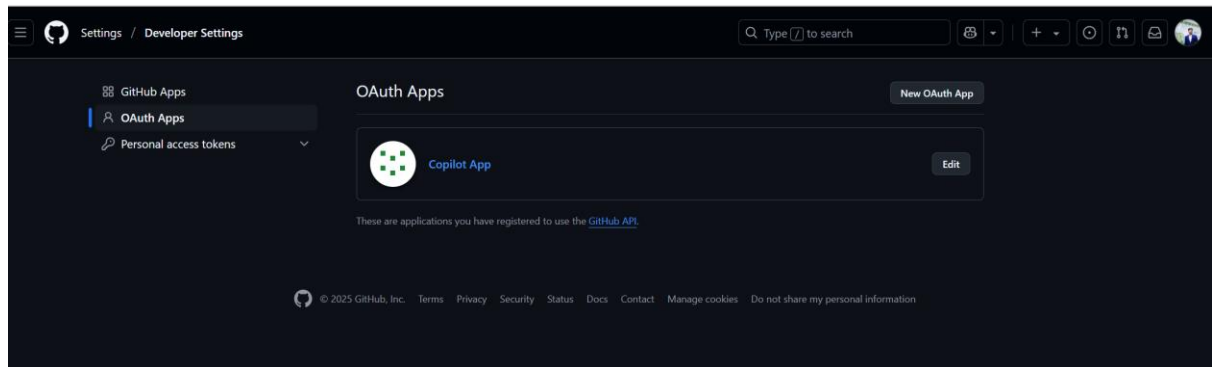1.) I have created a GitHub OAuth Application called "Copilot App"

Figure 3: GitHub OAuth Application

2.) In this Copilot App, there are 2 types of fields.
    a.   Auto Generated Fields by the OAuth Application
    b.   User Configured fields for the OAuth Application
3.) The below are the fields that are Auto Generated Fields by the OAuth Application.

Auto Generated Field Name
Client ID
Client Secret ID

Table 1:Auto Generated Fields by OAuth Application

4.) The below are the fields that needs to be Configured by the User for the OAuth Application.

User Configurable Field Name	Essential
Home Page URL	From which endpoint it receives the request for Authorization
Authorization Callback URL	Once the credentials have been authenticated and authorized, the endpoint it redirects to further continue with the application.

Table 2: User Configurable fields for the OAuth Application

Figure 4: OAuth Application configuration page

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

4. Chapter 4 : API Development

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

As part of Implementation, I have created 4 main APIs

1.) /dissertation/login

    a. This API is mainly used to receive the Request from React Application.
    b. Post receiving the Request, it will redirect to the GitHub OAuth App for Authorization.

2.) /dissertation/oauth/callback

    a. This API is used to communicate between GitHub OAuth Application, which I have created in earlier chapter and backend server post its authorization.
    b. 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.
    c. For example, reading the information from the GitHub Repository, Creation of Web-Hooks on the Repository.
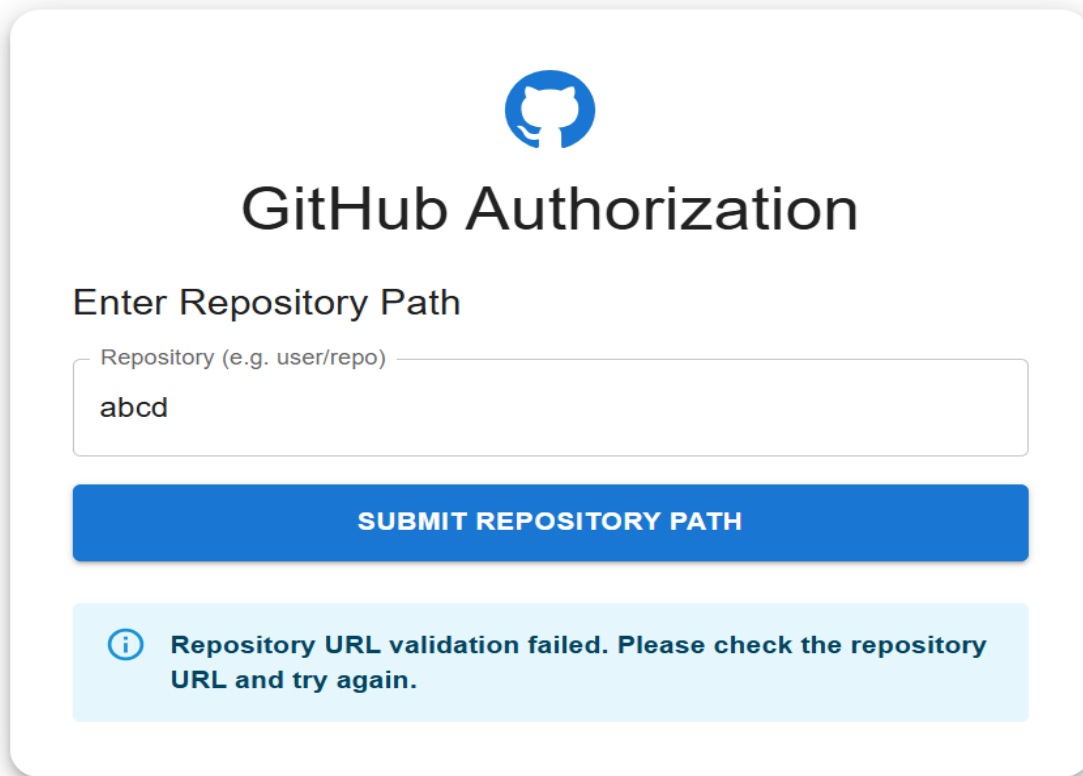
3.) /dissertation/set_repo

    a. This API is used to set the GitHub Repository name in the React Application Text-Box.

4.) /dissertation/repo/configuration

    a. This API will be used once the GitHub Repository is given by the user.
    b. In this API, the Request will contain certain parameters as part of Request Body.
        i. GitHub OAuth Token ➔ This OAuth Token will be used for validating permissions on the GitHub Private Repository Path.
        ii. Private GitHub Repository Path ➔ The Repository on which the vectorized needs to be done.
    c. As part of this API, the sequence of execution as follows.
        i. 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

ii. 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.

Figure 5: React JS Web Application Repository Configuration with Invalid Credentials

iii. 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.
iv. Once the user has valid permission to read the files from the
repository. Then the files will be read from the Repository.

5. Chapter 5 : Fetch List of Files from GitHub Repository

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.

6. Chapter 6 : Publisher Process for Generation of Text Embeddings

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

I have created a separate Python Process called **Embeddings_Publisher.py**

In this process, we will download the files from GitHub Repository into the local working directory.

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

7. Chapter 7 : Consumer Process for Generation of Text Embeddings and Storing the Embeddings into the Vector Database

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.

8. Chapter 8 : AWS Deployment

In this chapter, the main focus deploying the Application into AWS

The main server-side implementation contains 5 processes.

1.) React Application

    a. Front End React JS Web Application
    b. The Application will be running on Nginx Server
    c. A separate Dockerfile is created with name "Dockerfile.frontend"

2.) FastAPI Application

    a. Back End API Development
    b. The Application will be running on Uvicorn Server
    c. A separate Dockerfile is created with name "Dockerfile.backend"

3.) RabbitMQ Process

    a. RabbitMQ is used for Asynchronous Communication between Process
    b. Pulled the official RabbitMQ Image with Tag "3:management" from the Docker Hub

4.) Embeddings_Publisher.py

    a. This is a Python Application which is used to download files from the GitHub Repository and publish the files into the RabbitMQ Queue.
    b. A separate Dockerfile is created with name "Dockerfile.publisher"

5.) Embeddings_Consumer.py

    a. 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.
    b. A separate Dockerfile is created with name "Dockerfile.consumer"

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.

Sequence of steps post the creation of Dockerfiles for each processes mentioned above

1.) Built 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)

a. I have used this service in order to store the Docker Images which are created earlier.
b. I have created 5 Repositories, where each repository stores each process Docker Image.
c. Docker Image created from Dockerfile.frontend will be stored under "my-frontend" ECR Repository
d. Docker Image created from Dockerfile.backend will be stored under "my-backend" ECR Repository.
e. Docker Image created from RabbitMQ Image will be stored under rabbitmq ECR Repository.
f. Docker Image created from Dockerfile.publisher will be stored under "my-publisher" ECR Repository
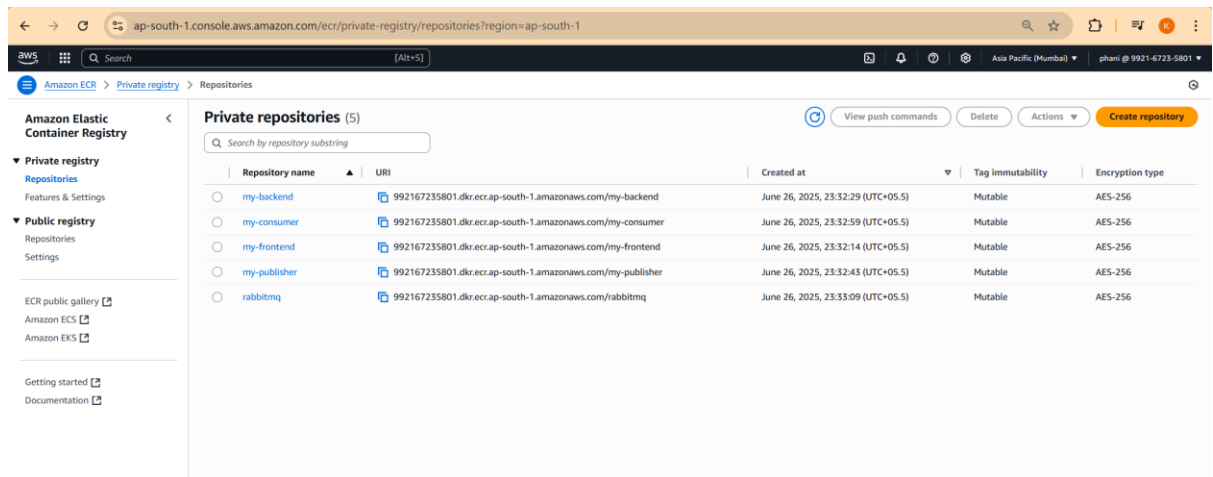g. Docker Image created from Dockerfile.consumer will be stored under "my-consumer" ECR Repository.

Figure 6: AWS ECR Repositories Page

2.) AWS ALB (Application Load Balancer)

a. Created an Application Load Balancer for deployment into the Internet.

b. From the Application Load Balancer, we are getting the DNS Name which is the public URL which can be accessible throughout the Internet.

c. The Application Load Balancer with name "dissertation-alb" has been created.

d. Created a Target Group "tg-frontend" and "tg-backend" for both the frontend and backend since they will be exposing to the Public Internet.

e. Created a Listener at the Port 80 to the Application Load Balancer and attached the tg-frontend target group

f. Created a Rule (Path based) to the Listener Port 80 to the Application Load Balancer and attached tg-backend target group
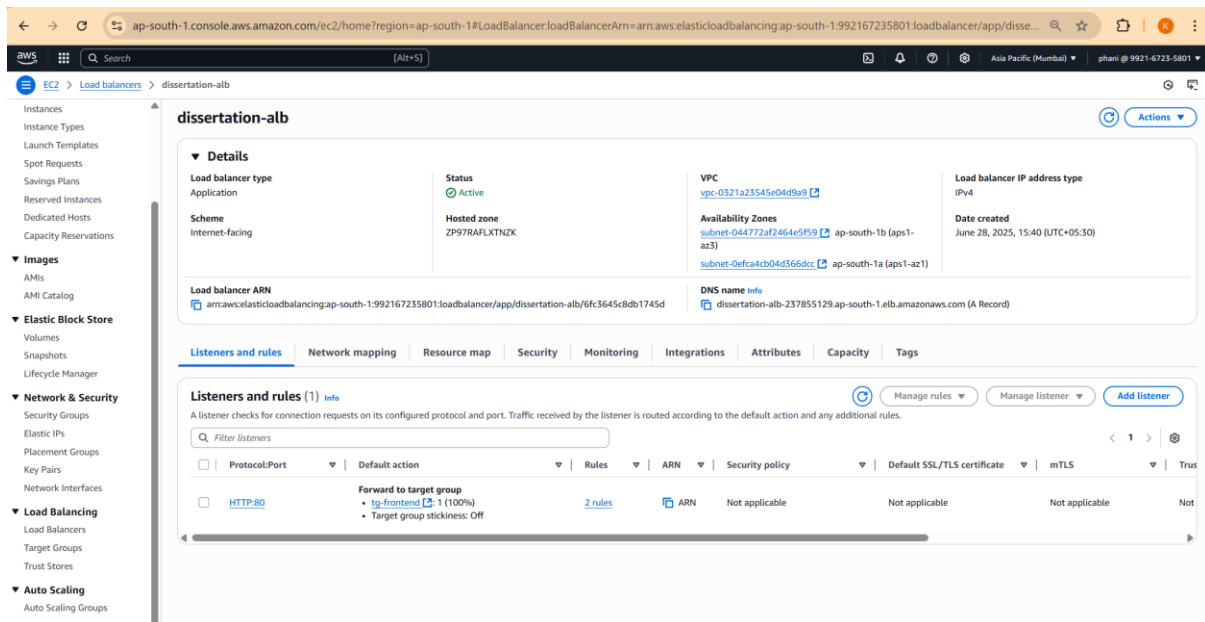
Figure 7:AWS ALB Configuration Page

3.) AWS EFS (Elastic File System)

    a. Created a shared directory called "shared-repo-dir" which is required for Publisher and Consumer Process.
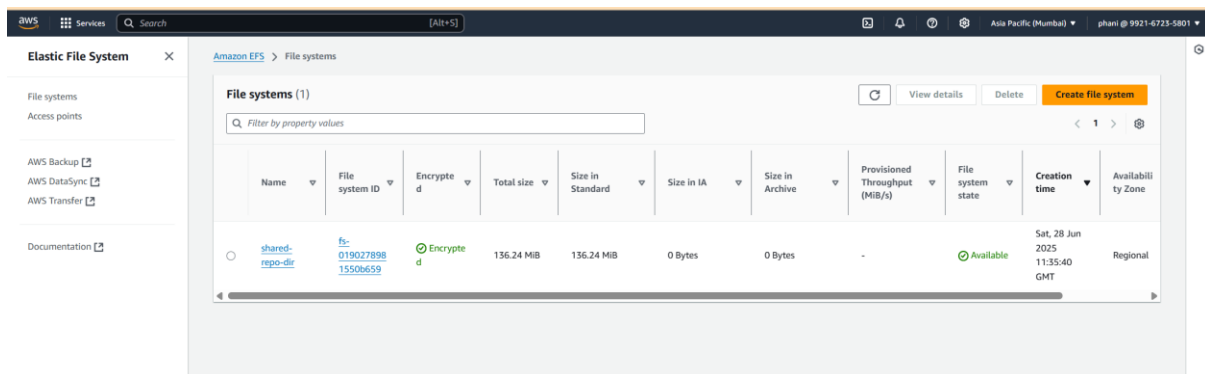

Figure 8:AWS EFS Configuration Page

4.) AWS NLB (Network Load Balancer)

    a. 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.

    b. The DNS name of Network Load Balancer will be used for backend, publisher, consumer process.
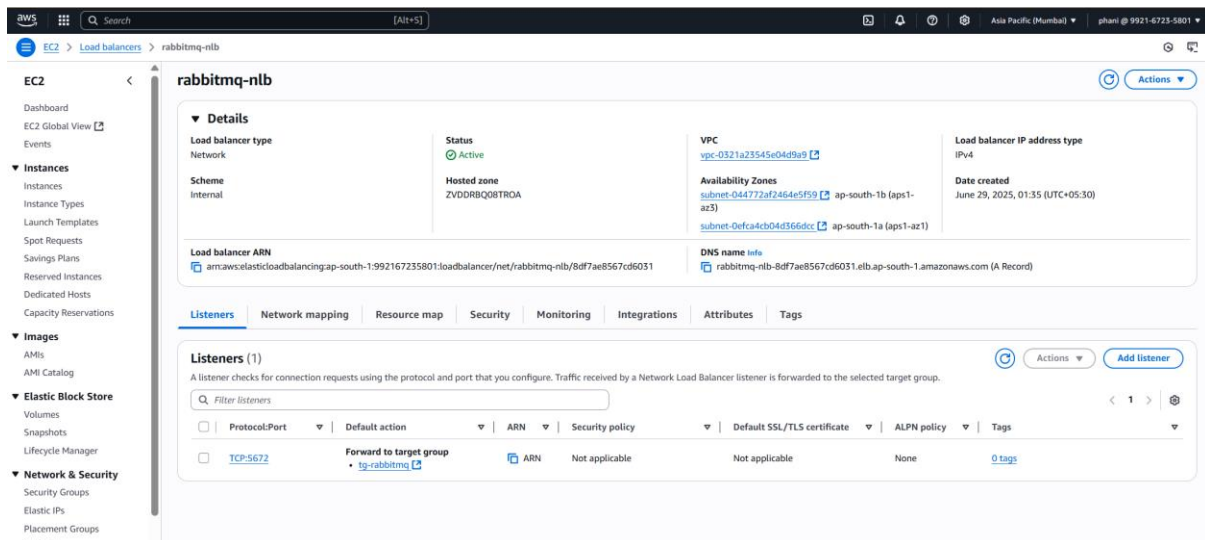
Figure 9: AWS NLB Configuration Page

5.) AWS ECS (Elastic Container Service with Fargate as Launch Type)

a. In order to launch ECS containers, there are prerequisite things needs to be defined initially.

b. Creation of ECS Cluster, where the ECS containers will be running.

c. Once the ECS Cluster has been created. Next we have to create the Task Definitions for each repository.

d. Created the Task Definitions for all the repositories.

e. In the Task Definition file, we need to specify any Environment Variables used inside the Docker file and any EFS Volume used.
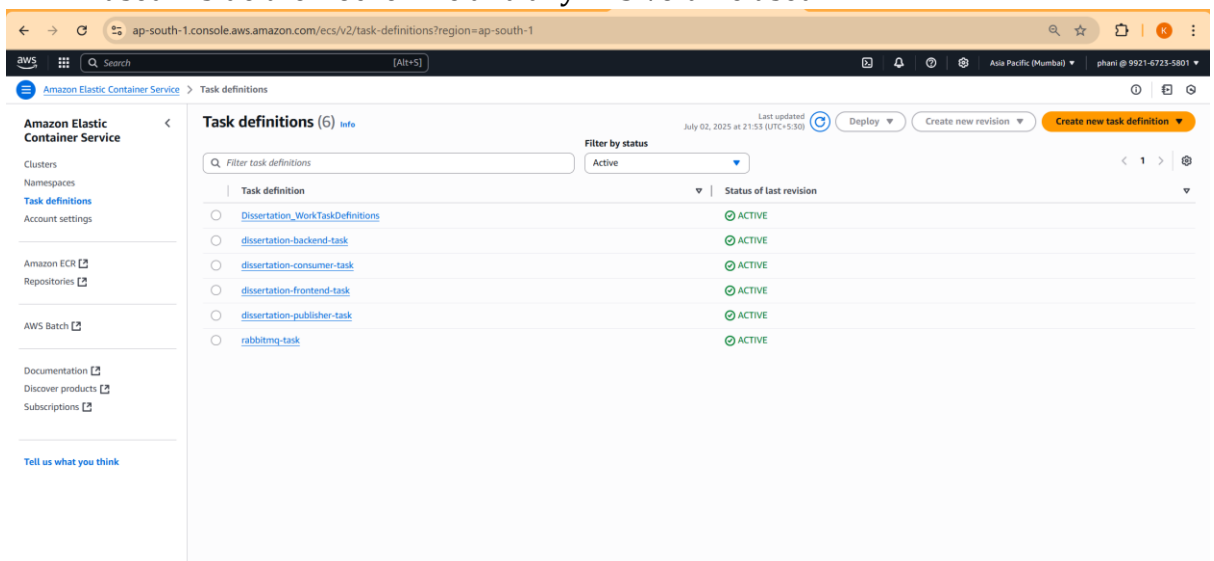

Figure 10: ECS Task Definitions Page

f. Created Task Definitions for each ECR Repository.
g. Total 5 Images are created as part of Server-Side process and each Docker Image has been placed under different ECR Repository.
  i. 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)
  ii. 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)
  iii. RabbitMQ Process
    1. Docker Image = RabbitMq:3-management
    2. ECR Repository = rabbitmq
    3. ECS Task Definition = rabbitmq-task
  iv. Publisher Process
    1. Docker Image = my-publisher
    2. ECR Repository = my-publisher
    3. ECS Task Definition = dissertation-publisher-task
  v. Consumer Process
    1. Docker Image = my-consumer
    2. ECR Repository = my-consumer
    3. ECS Task Definition = dissertation-consumer-task

h. 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 Definitions.
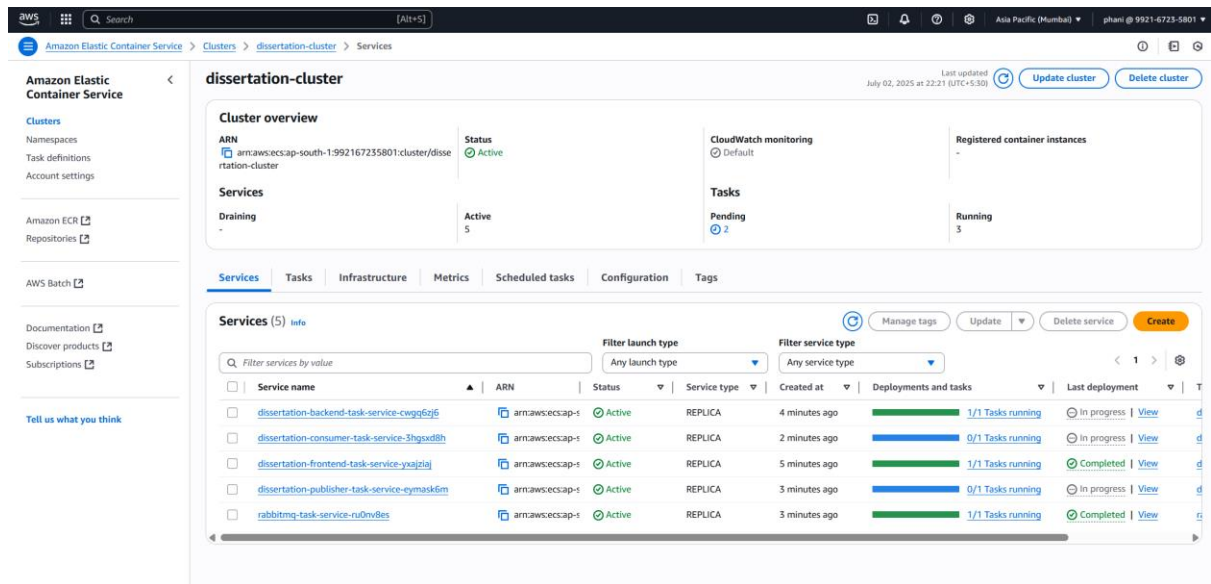
Figure 11: AWS ECS Services Configuration Page

i. Now the Application has been deployed into the internet.
j. 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"
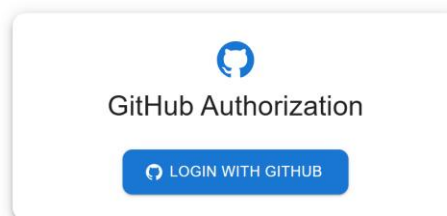

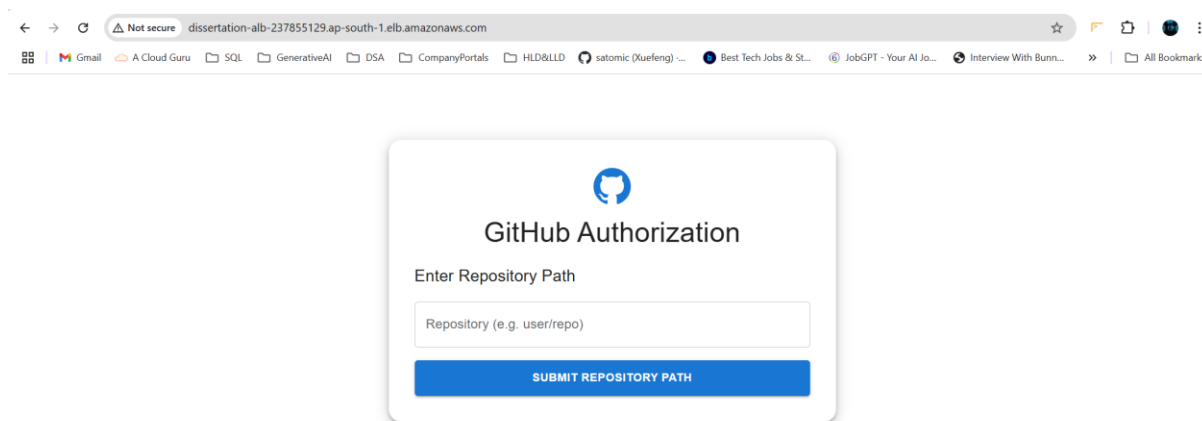Figure 12: React Application Login Page in AWS Environment

Figure 13: React Application Repository Configuration Page in AWS Environment
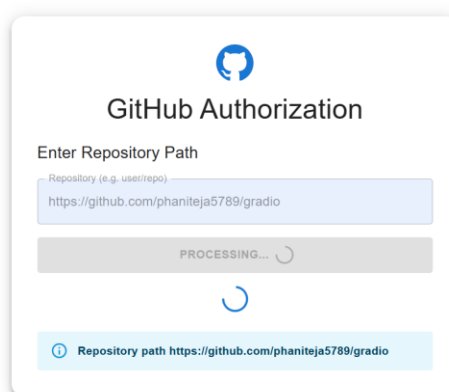

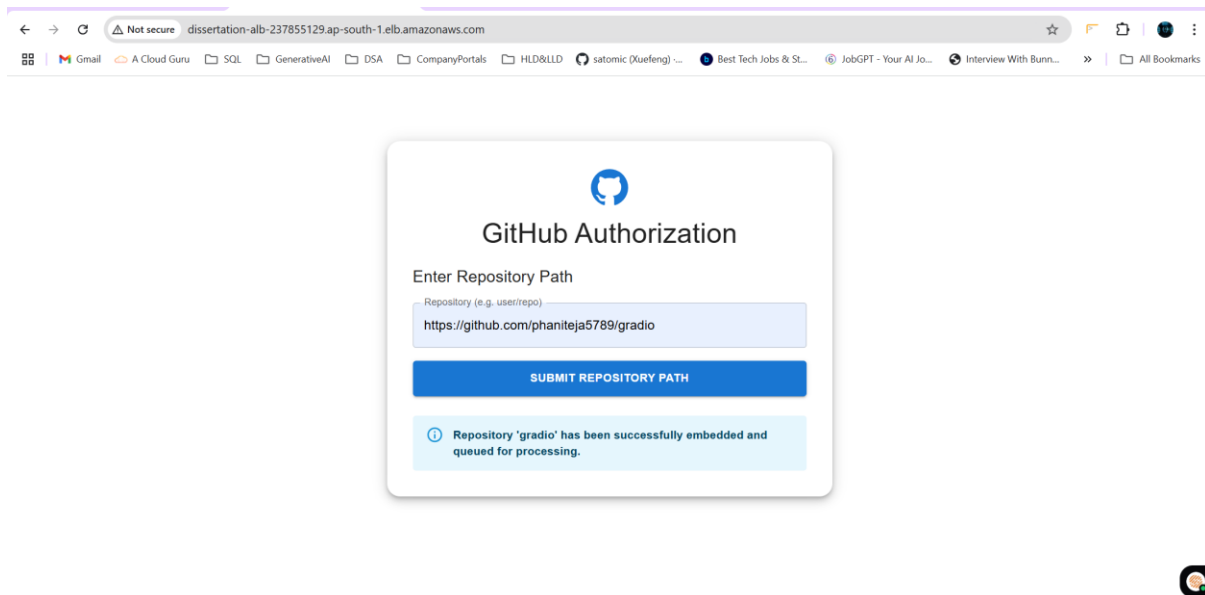Figure 14: Fetching the files from the Configured Repository

Figure 15: Queued for Processing

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

9. Directions for future work after mid semester

As till mid-semester the deployment into the AWS has been completed. The below are the next steps with respect to the pending chapters.

9.1.   Chapter 9 : Webhook Configuration & Integration

We need to create the GitHub Webhook for the configured repository and needs to the listen to the list of changes that gets merged to the main branch of the Repository.

9.2.   Chapter 10 : Creation of Retrieval APIs.

Need to create the API which will be responsible for retrieval of relevant content from the ChromaDB Collection where the embeddings were stored

9.3.   Chapter 11 : Creation of Visual Studio Code Extension

Need to create the visual studio code extension, where user can query the question as per their request.

9.4.   Chapter 12 : Integration of Retrieval APIs with Visual Studio Code Extension.

Need to Integrate the Visual Studio Code Extension along with the Retrieval APIs where the RAG will be performed and generates the answer. And the generated response will be given back to the user which displays in the Visual Studio Code

9.5.   Chapter 13 : Testing all workflows

Now this completes the dissertation work, needs to verify all the workflows and the error messages and responses generated by the LLM.

10. References

The following are referred journals from the preliminary literature review.

• Vaswani, Ashish, et al. "Attention Is All You Need." Advances in Neural Information Processing Systems (2017).

• Semantic Web Technologies: Trends and Research in Ontology-based Systems. Journal of Semantic Web and Information Systems.

• LangChain Documentation. Available at: https://docs.langchain.com/

• ChromaDB and FAISS Documentation. Available at: https://docs.trychroma.com/, https://faiss.ai/

• ReactJS and Visual Studio Code Extension Guidelines. Available at: https://code.visualstudio.com/api

• OpenAI and Hugging Face Embeddings Technical Specifications and Documentation.