ShopChat App

A Complete Guide and Solution Document

ShopChat App Document Contents

1 Pre-requisites 5

2 RAG Architecture 5

2.1 Tools and models used 5

2.2 LLM Performance Evaluation (Latency) 5

2.3 Finetune Model 6

2.4 Architecture Diagram 7

2.5 Inference UI (Prototype) 7

2.6 Feedback Loop 8

2.7 Improvements and Performance Enhancements 8

3 Initial Setups 8

3.1 Project Folder 8

3.2 Load project in VS Code 9

3.3 Create AWS user and add IAM access 9

3.4 Create AWS API keys for the user 10

3.5 Create AWS S3 bucket 11

3.6 Create Google Generative API key 11

3.7 Create OpenAI API key 12

3.8 Create Virtual environment and install dependencies 12

3.9 Update environment key and variables 13

4 Inference ShopChat App 14

4.1 Run Locally 15

4.1.1 Run Streamlit app 15

4.1.2 Inference ShopChat app 15

4.2 Run using Docker 16

4.2.1 Start Docker desktop 16

4.2.2 Build and Run Docker Image 17

4.2.3 Inference ShopChat app 18

5 Data Loading and Preprocessing 19

5.1 Data Ingestion 19

5.1.1 Ingestion Processes 19

5.1.2 Running data ingestion 19

5.2 Data Captioning [Optional Step] 20

5.2.1 Captioning Processes 21

5.2.2 Running data captioning 21

5.3 Data Embedding 22
5.3.1 Embedding Processes 22
5.3.2 Running Google Embedding 22
5.3.3 Running OpenAI Embedding 23
6 Full inferencing after data loading 25
6.1 Upload Your S3 25
6.2 Selection of LLM model 25
6.3 Display more images 26
6.4 Item not found 27
6.5 Listing Link 27
6.6 Feedback 28
6.7 Home button 28
7 CICD with GitHub Action and AWS 29
7.1 Create Git Repository 29
7.2 Create AWS ECR Repository 30
7.3 Create AWS EC2 Instance 31
7.4 Create IAM Role and add to EC2 instance 34
7.5 Install Docker into EC2 instance 37
7.6 Add Secrets to GitHub Actions 38
7.7 Add Runner 39
7.8 Add Workflow yml to GitHub repository 41
7.9 CICD Pipeline Triggers 43
7.10 Open port 8501 in EC2 45
7.11 ShopChat App inference from AWS using public URL 47
8 EDA and Data Preparation 49
9 Supporting scripts and files 50
9.1 env_variables.txt 50
9.2 exception.py 50
9.3 logging.py 51
9.4 utils.py 51
9.5 Dockerfile 52
9.6 .dockerignore 52
9.7 .gitignore 53

9.8 setup.py **54**
9.9 requirements.txt **54**
10 Observations and Issues ***55***
10.1 Finetune Embedding and LLM models **55**
10.2 Inference with OpenAI LLM model **56**
10.3 Response Thruput **56**
10.4 Bucket selection **56**
10.5 Image Caption **56**
11 Challenges ***57***
11.1 Data Handling **57**
11.2 Language and Localization **57**
11.3 Integration of AI Models **57**
11.4 Scalability and Performance **57**
11.5 User Experience Design **58**
12 Future Improvements ***58***
12.1 Model Enhancement **58**
12.2 Multi-language Support **58**
12.3 Personalization **58**
12.4 Visual Search Capabilities **58**
12.5 Analytics and Feedback Loop **58**

1 Pre-requisites

- GitHub account
- VS Code installed and setup in your local system
- Docker desktop client installed in your local system
- AWS account with access to S3, EC2 and ECR
- Google Gen AI account and API key
- OpenAI API account and API key

2 RAG Architecture

RAG based application is created from scratch using Amazon Barkley Object open-source Database.

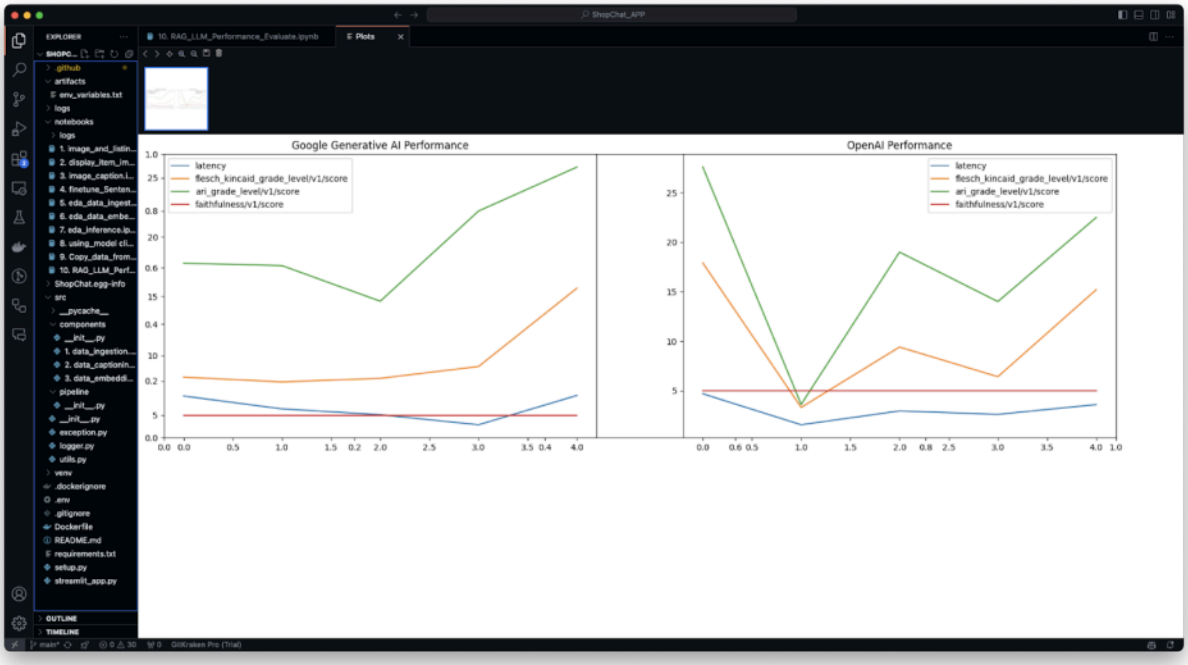
2.1 Tools and models used

Following tools and models are used in the RAG architecture

- Streamlit: is used to design inference UI
- AWS S3: storing data and vector database
- AWS ECR: repository to save application docker image
- AWS EC2: web service compute capacity to run the application
- Docker Container: containerize the application, environment
- GitHub Action: used to create CICD pipeline
- FAISS: vector store DB to store generated embedding indexes
- LangChain: framework/tools used for building applications that leverage LLM
- Embedding models used are:
  - GoogleGenerativeAIEmbeddings: For Google's embedding model
  - OpenAIEmbeddings: For OpenAI's model
  - HuggingFaceEmbeddings: For a fine-tuned model loaded from S3
- LLM models used:
  - Google Gemini-pro
  - OpenAI 3.5 turbo
  - Finetune Model

2.2 LLM Performance Evaluation (Latency)

Using mlflow evaluate function, we measured the faithfulness performance metrics. We had taken 5 simple and common questions, ran them thru the Langchain LLM chain and retriever to get the answers and stuffed documents retrieved from FIASS vector DB.
We then pass them to mlflow.evaluate() function to get the faithfulness_metric.
Source code can be found in: **10. RAG_LLM_Performance_Evaluate.ipynb** notebook



Observation: Although latency for OpenAI model is less, there are lot of variances on other metrices. Only evaluated with 5 questions. We can try with more questions and get the mean for all metrices for better quality

2.3 Finetune Model

We finetuned sentence transformer model “BAAI/bge-small-en” using synthetic questions generated from OpenAI GPT-3 model. We used 100 Json datapoint to create questions from GPT-3. These 100 questions are then split into (80%) train and (20%) validation. We finetuned sentence transformer model with test questions dataset and evaluate the performance against validation data.

Compared the Hit rate with OpenAI and non-finetuned sentence transformer “BAAI/bge-small-en” model. The compared results were not so impressive

model	epoch	steps	cos_sim-Accuracy@1	cos_sim-Accuracy@3	cos_sim-Accuracy@5	cos_sim-Accuracy@10	cos_sim-Precision@1	cos_sim-Recall@1	cos_sim-Precision@3	cos_sim-Recall@3	...	dot_score-Recall@1
bge	-1	-1	0.85	1.0	1.0	1.0	0.85	0.85	0.333333	1.0	...	0.85
fine_tuned	-1	-1	0.85	1.0	1.0	1.0	0.85	0.85	0.333333	1.0	...	0.85

Observation: Synthetic questions generated by OpenAI model were not so buyer friendly and hence the finetuned model did not perform well on inference.

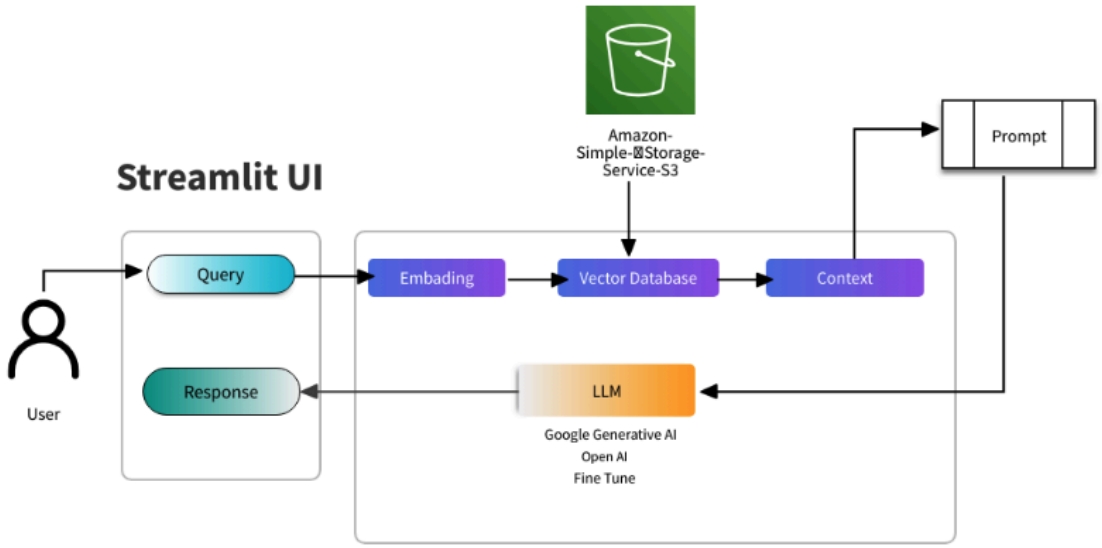
Improvements can be done on finetune:

1. Create better questions and answers from the ABO data using better prompt
2. Train for more epochs and more question-and-answer dataset

3. Question-and-answer generation model can improve performance
4. Fine Model selection can also improve on performance

Finetuning Notebook: 4. finetune_SentenceTransformers_embedding_model.ipynb

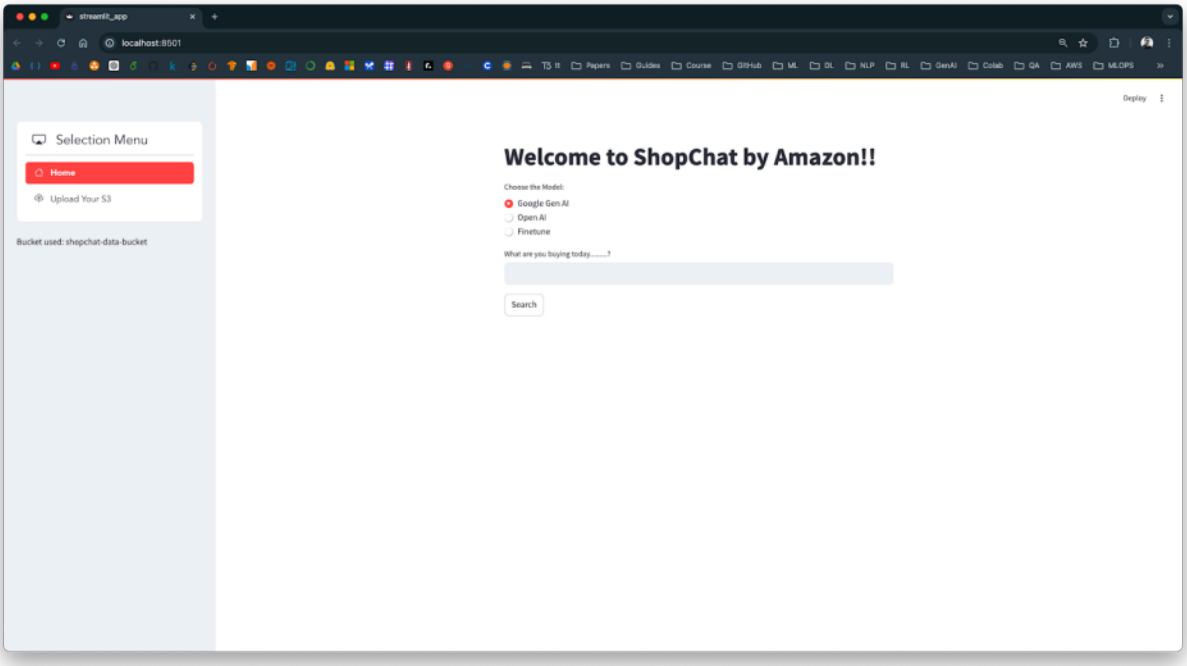
2.4 Architecture Diagram



2.5 Inference UI (Prototype)

A simple Streamlit application is created to take the user questions and provide response. There are options to select different LLM models and data source to run the application.

- User Input: Query received through Streamlit UI.
- Model Selection: Choose between Google Gen AI, OpenAI, or Finetuned models.
- Response Generation: Using selected LLM to process the query and generate recommendations.
- Image Retrieval: Display relevant images from ABO S3 using Python Pillow library
- Displaying Results: Showing recommendations and images in the Streamlit UI.



2.6 Feedback Loop

Feedback taking mechanism is created within the Streamlit app where user can like and dislike a product, share the feedback comments/suggestion. This is enabled for every item that LLM returns in respond.
We have not done anything with the feedback received from the UI, there is a scope to improve the quality LLM response by implementing feedback received from the user

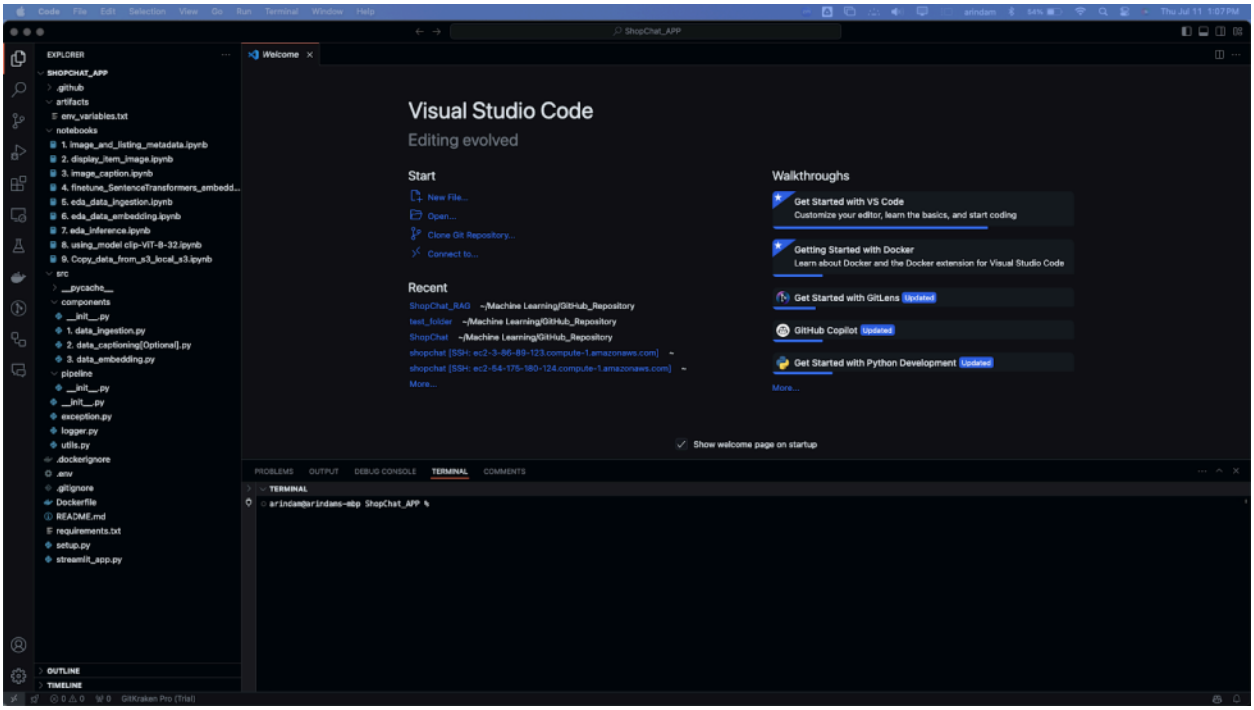
2.7 Improvements and Performance Enhancements

- Optimize Vector store and Indexing: Ensure efficient indexing and retrieval.
- Caching: Implement caching mechanisms for frequently accessed data.
- Batch Processing: Handle bulk queries to improve throughput.
- Parallel Processing: Utilize parallel processing for downloading.
- Fine tuning: Both Embedding and LLM models can give better results.
- Feedback mechanism: To retrieve right results.

3 Initial Setups

3.1 Load project in VS Code

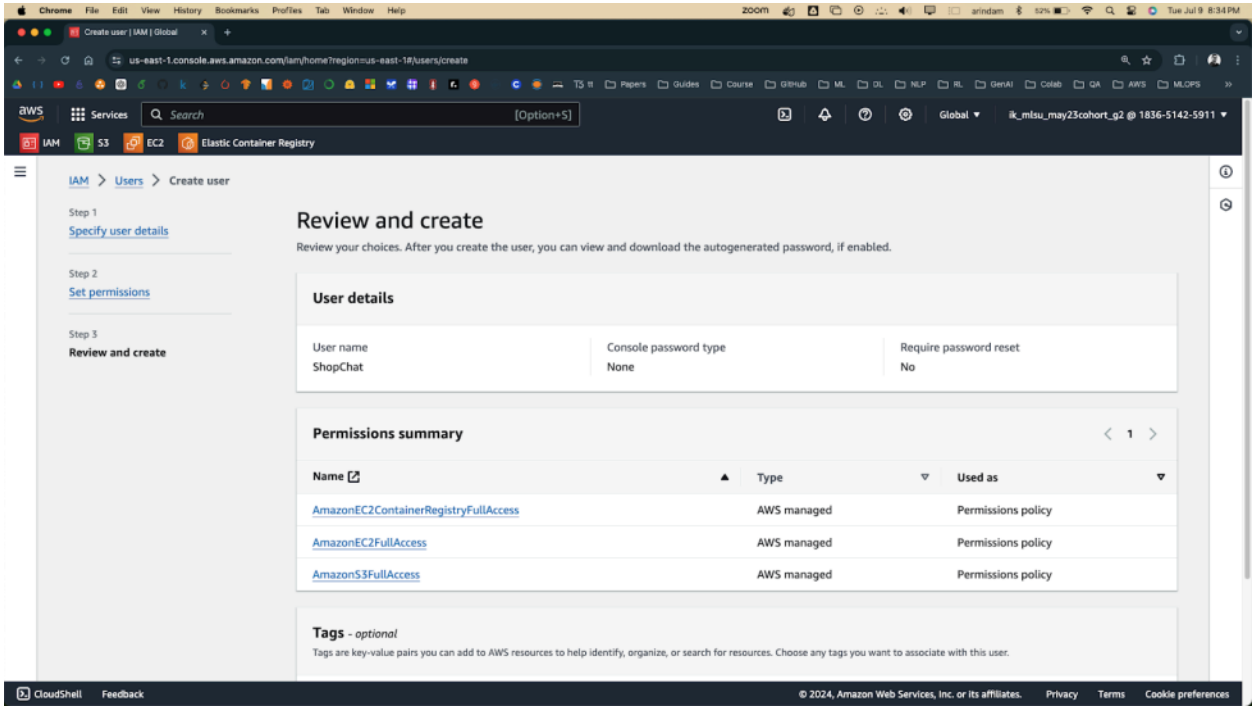
Get the project from Github: https://github.com/arindam-29/ShopChat_APP



3.2 Create AWS user and add IAM access

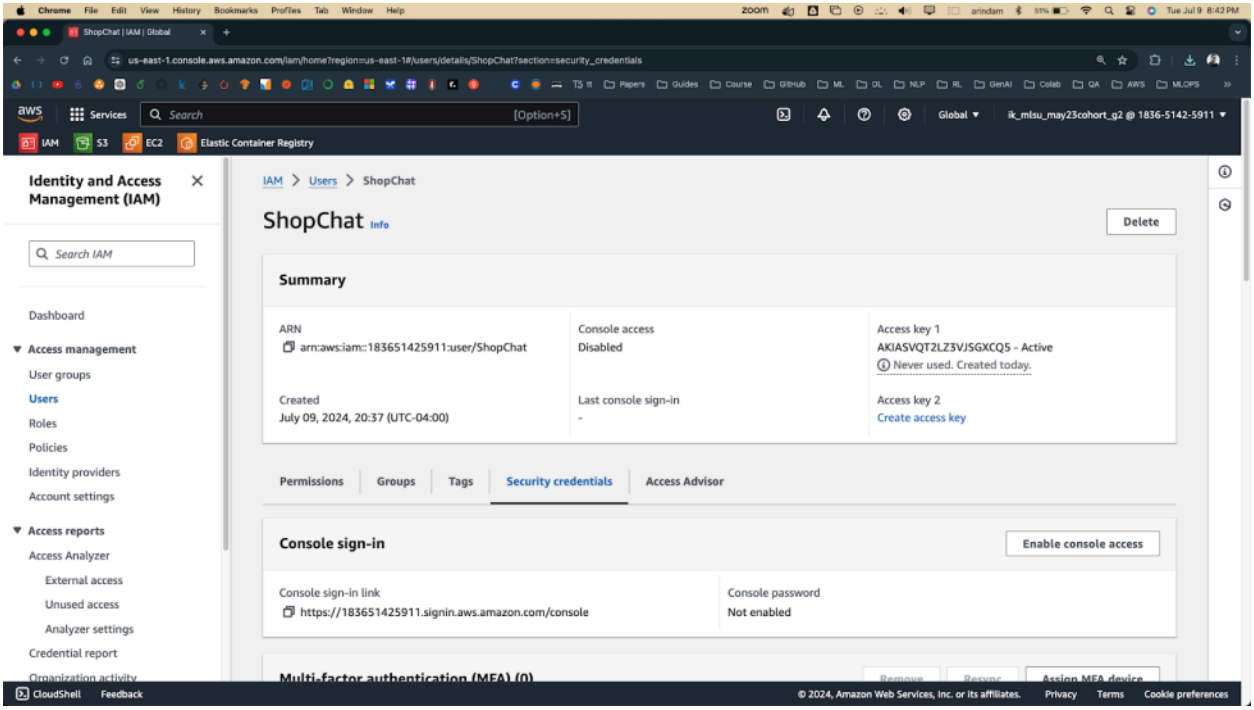
Create a new user in IAM and attach the following policies directly:

- AmazonEC2ContainerRegistryFullAccess
- AmazonEC2FullAccess
- AmazonS3FullAccess



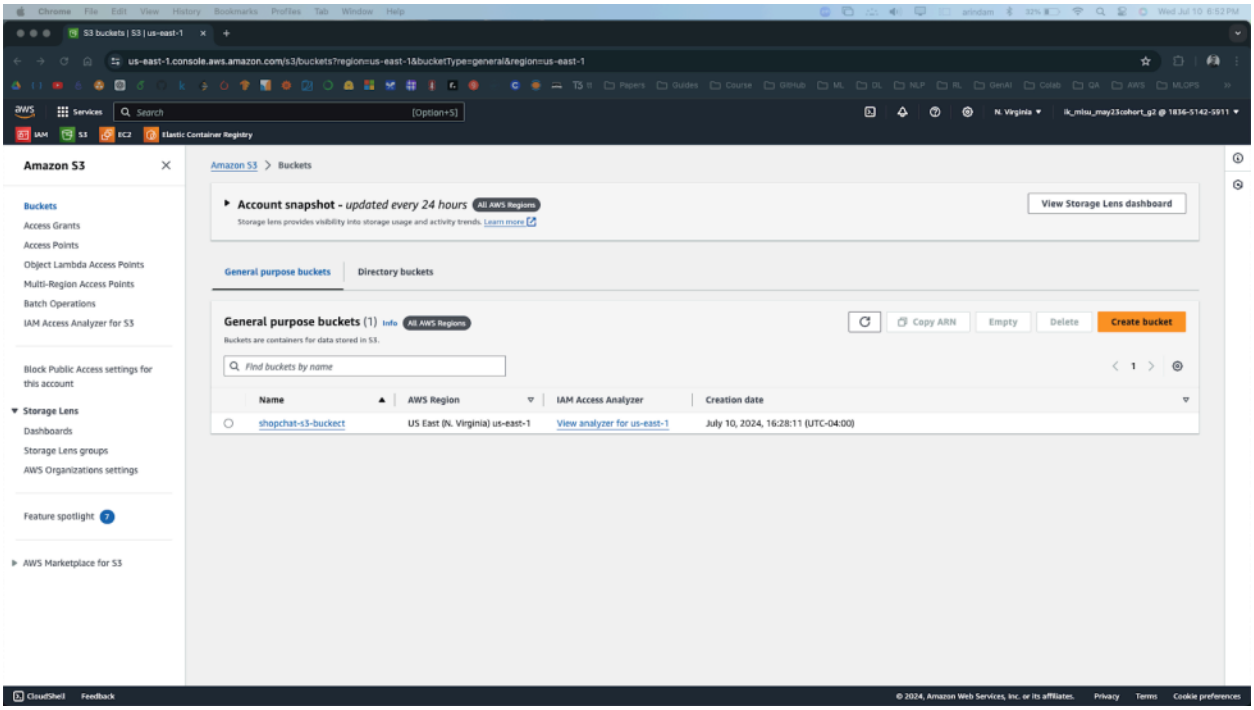
3.3 Create AWS API keys for the user

Once the new user is created, click the username to go to the IAM access page. Under security credential create access keys (you can select use case as Other in next page) and download the access key



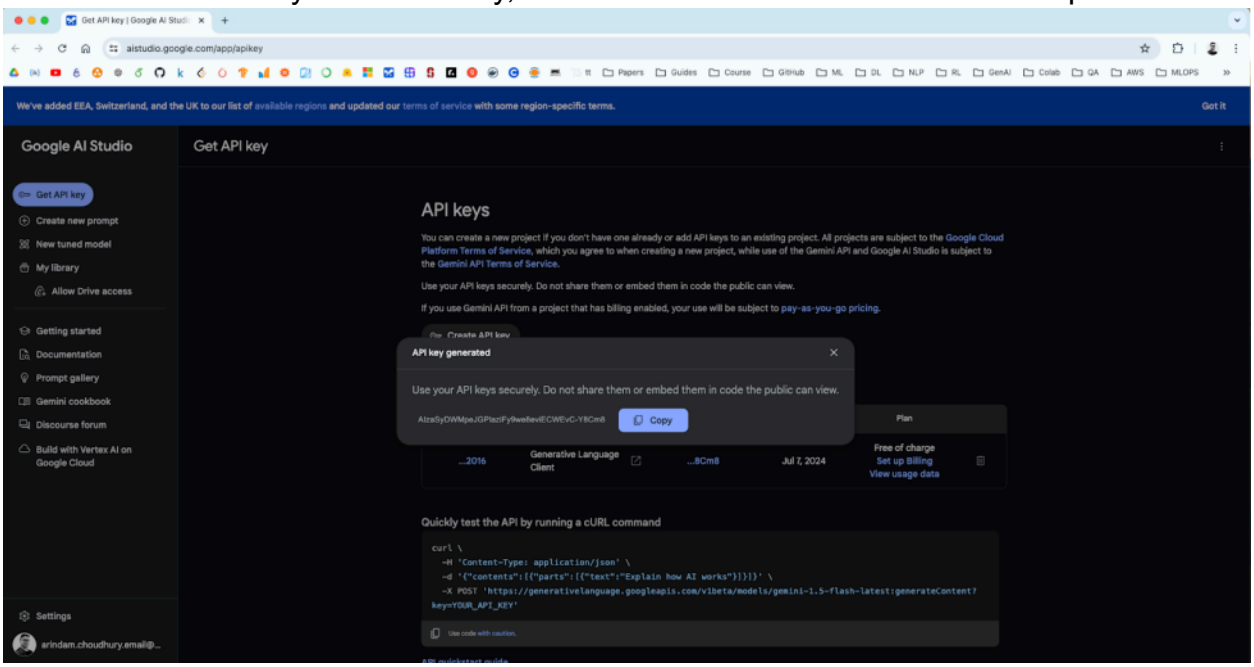
3.4 Create AWS S3 bucket

Create a new S3 bucket to save all data that will get created during execution



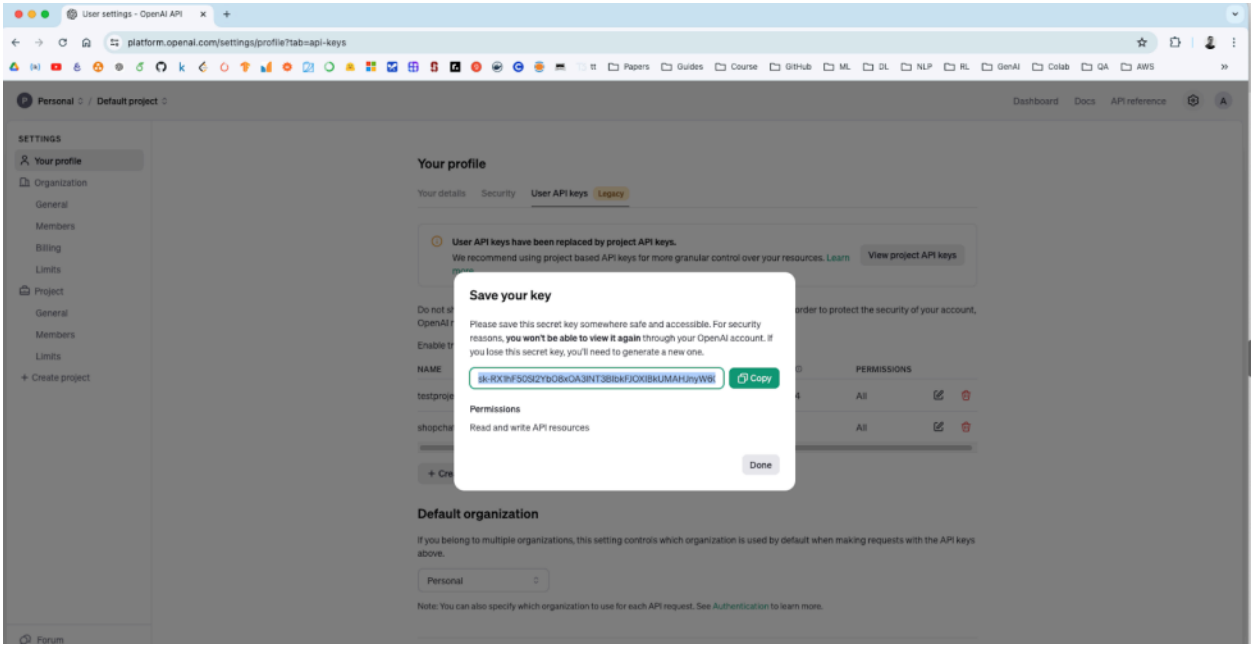
3.5 Create Google Generative API key

- Go to the link: https://aistudio.google.com/app/apikey
- login / create account with google generative AI
- Create an API key. **Save** the key, we will add them in .env file in next step



3.6 Create OpenAI API key

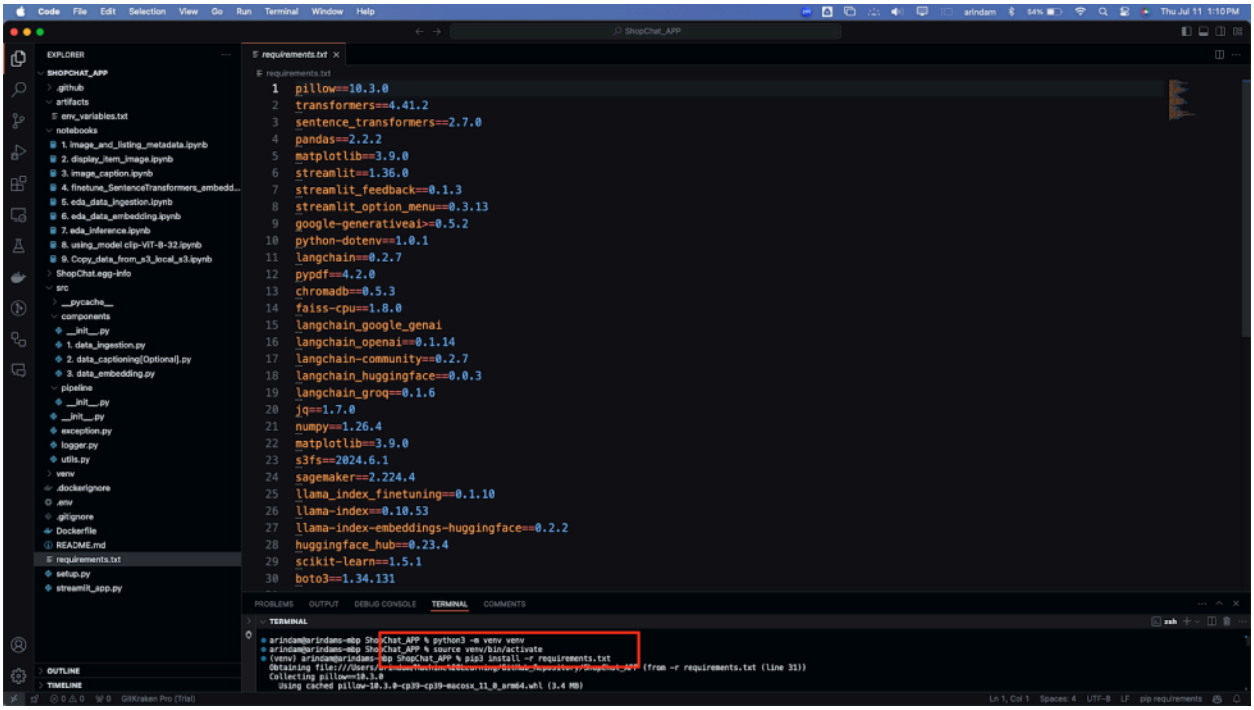
- Go to the link: https://platform.openai.com/settings/profile?tab=api-keys
- login / create account with OpenAI
- Create an API key. Save the key, we will add it in .env file in next step



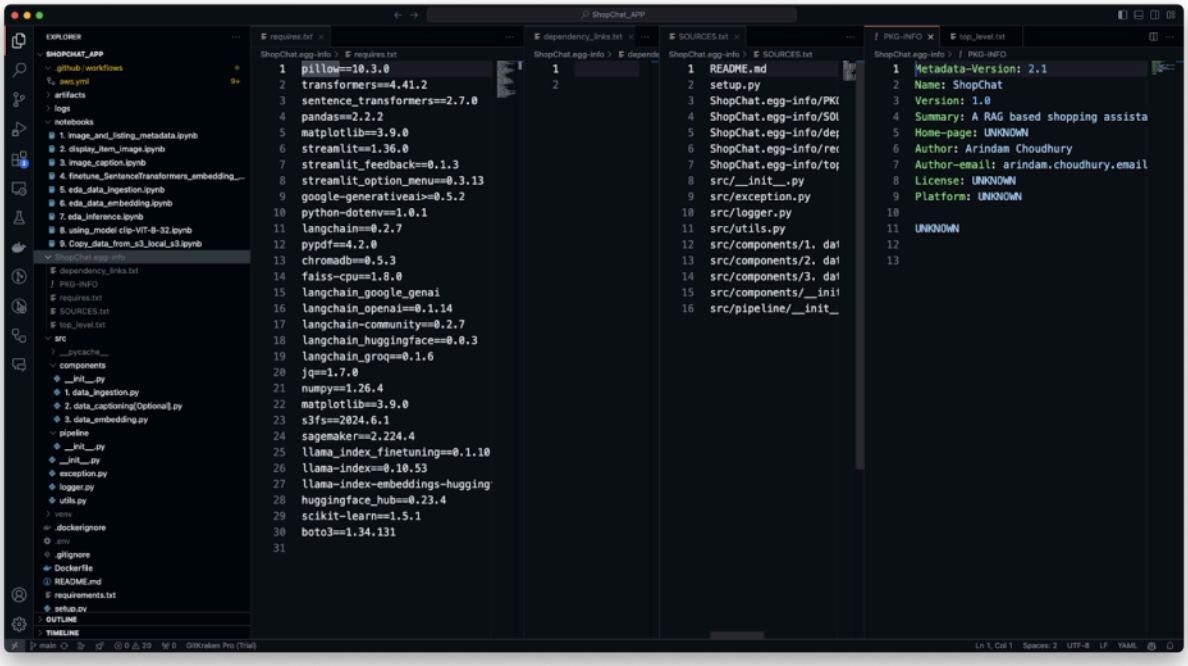
3.7 Create Virtual environment and install dependencies

Setup the virtual environment for running this project. We need **Python 3.9** for this. If your machine has python 3.9 then you can directly create virtual environment or may use conda or other tools to create the environment with python 3.9.
We will use python venv in this project since my machine has Python 3.9 and install dependencies (requirements.txt) using following commands

a. python3 -m venv **venv**
b. source **venv**/bin/activate
c. pip3 install -r requirements.txt

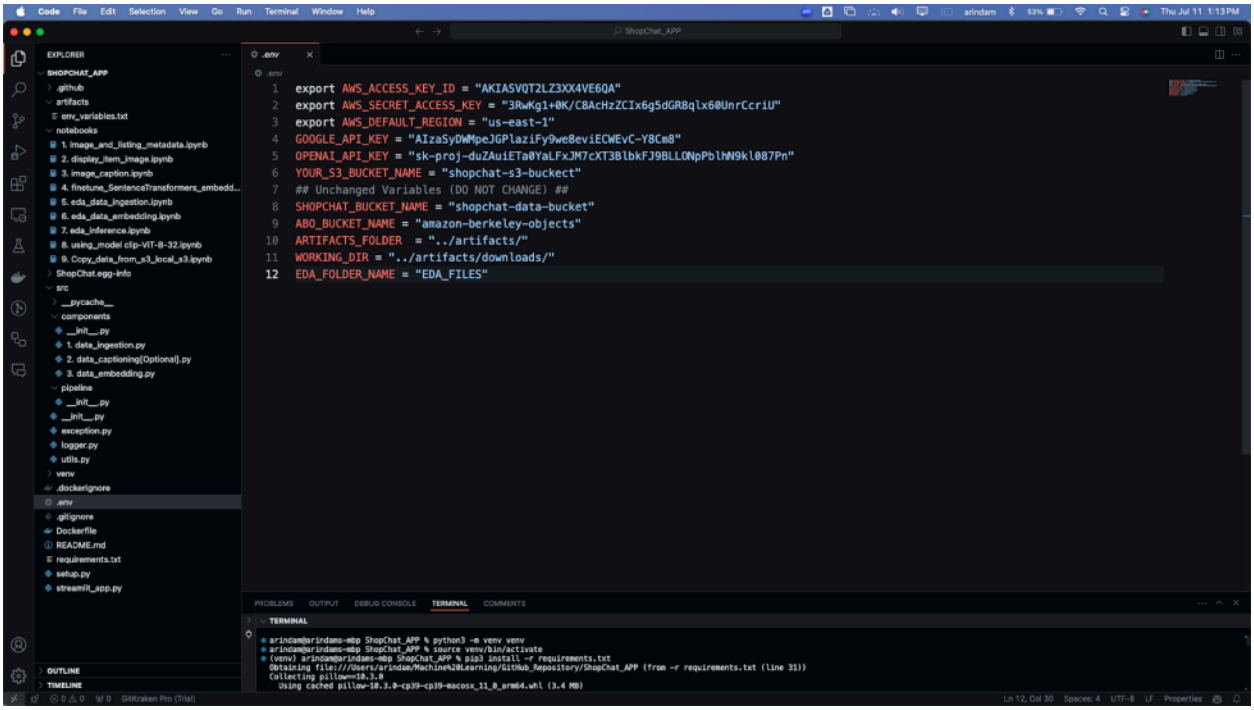


Once the requirements get installed in virtual environment, setup tool creates the Info folder about the project



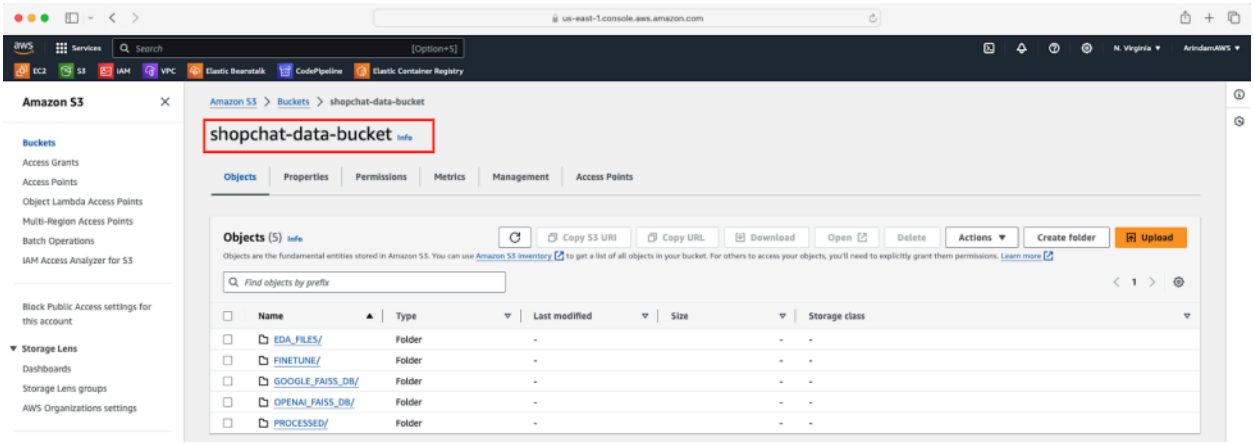
3.8 Update environment key and variables

- Go to the file **.env** and add all the keys saved from previous steps
- Add your S3 bucket name in YOUR_S3_BUCKECT_NAME variable
- Leave all other variables unchanged as described



4 Inference ShopChat App

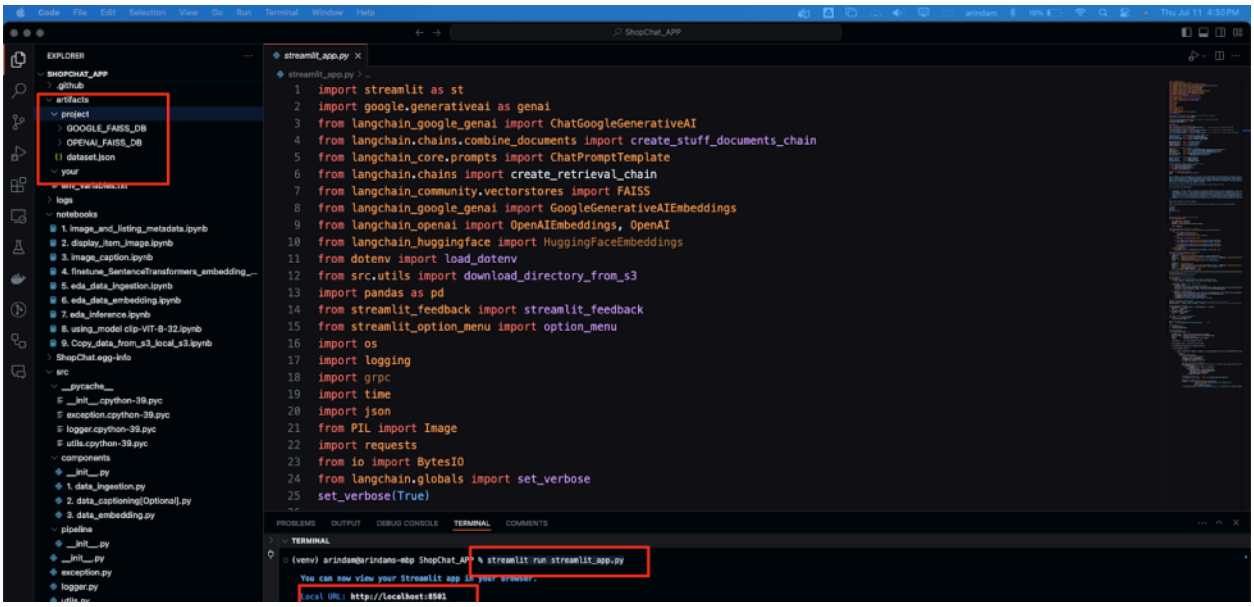
We will discuss three approaches here to interact with ShopChat app. To run the app you **may not** need the data loading scrips to run first. Required data is already present in another public S3 bucket and can be access by any AWS user using any AWS secret key and access key. However, in the app you cannot upload data from your S3 bucket until you run the data loading process described later in this document. App uses following bucket during inference [listed as SHOPCHAT_BUCKET_NAME = "shopchat-data-bucket" in .env file]
Note: This bucket will be used by default inside the app



4.1 Run Locally

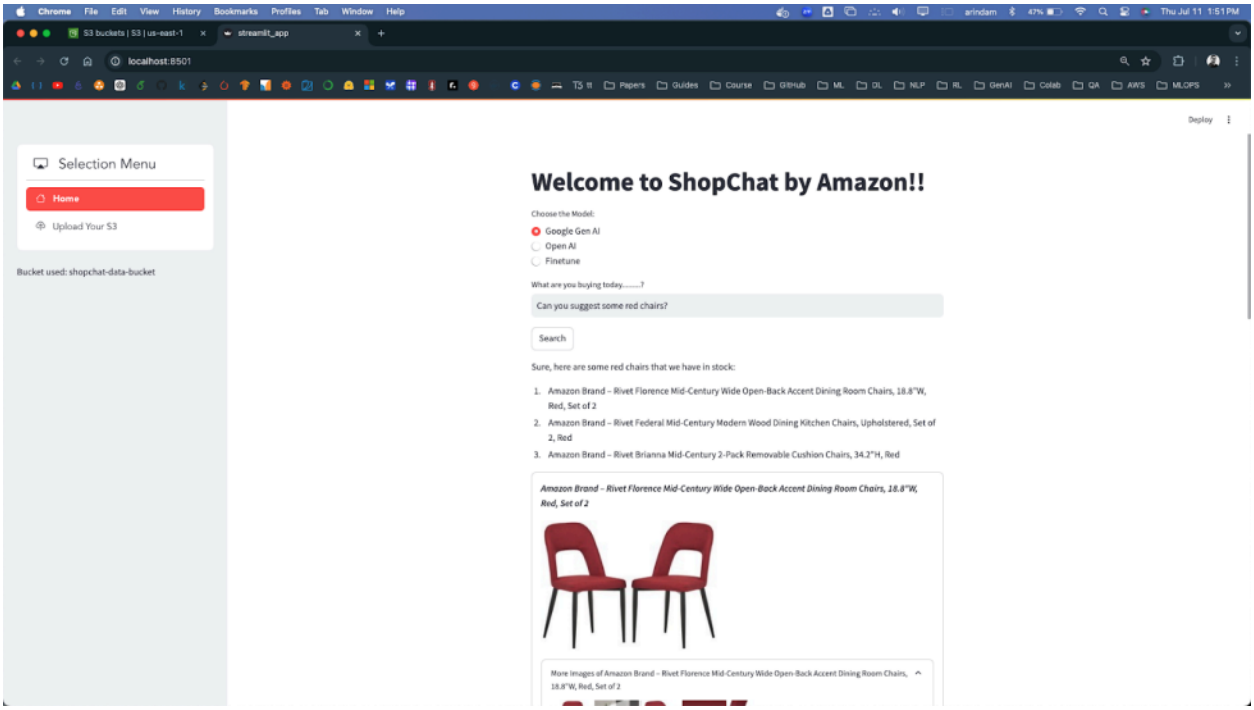
4.1.1 Run Streamlit app

- Open a new terminal and type: **streamlit run streamlit_app.py**
- A new folder in artifacts/project will be created and necessary FAISS databases and Json dataset will be copied from project S3 bucket as mentioned above

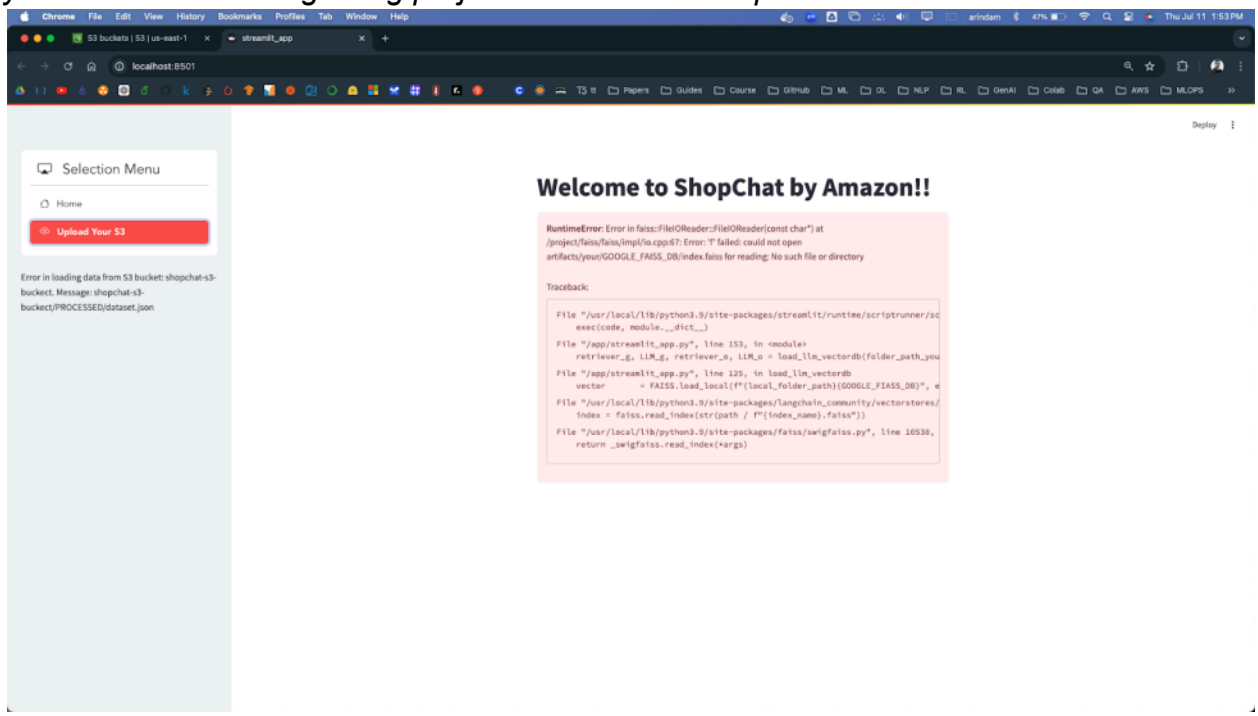


4.1.2 Inference ShopChat app

Go to the Local URL: http://localhost:8501 to interact with the app, ask questions and select different LLM models for your response



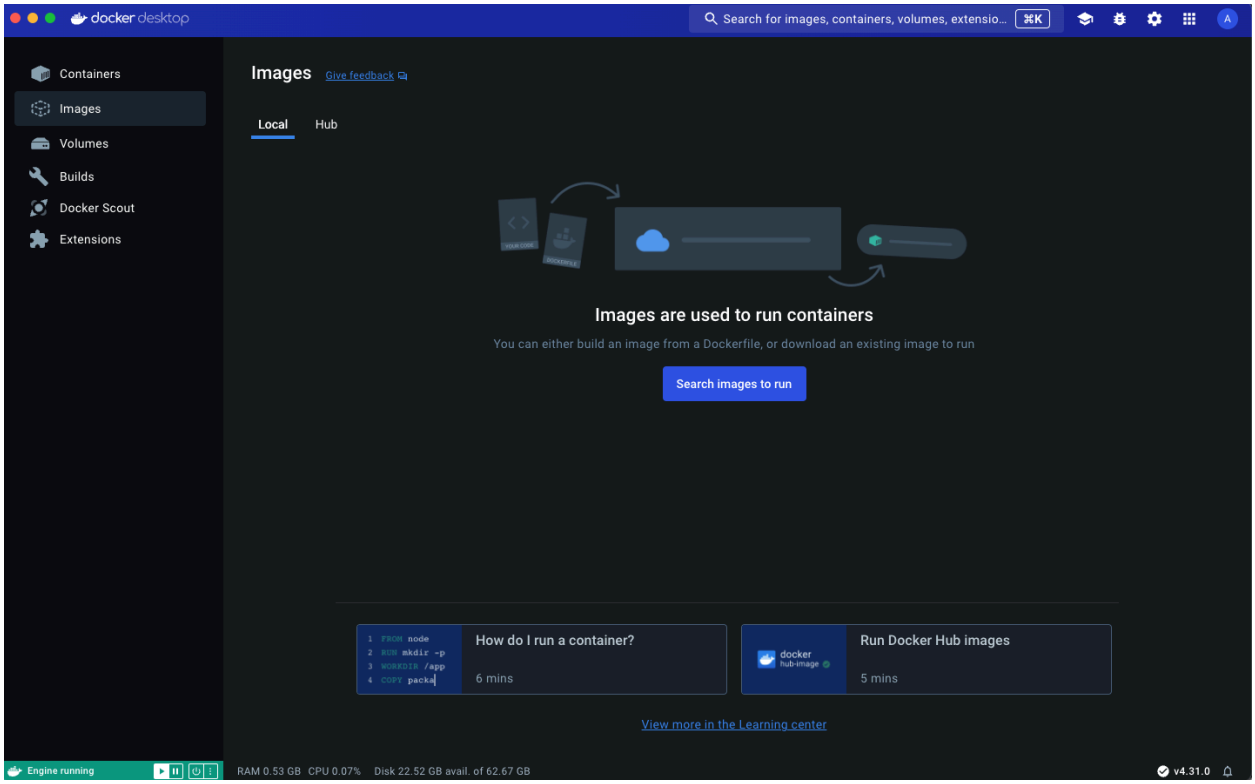
Note: If you click on Upload Your S3; this will give error as we have not created S3 yet nut default running using project's S3 bucket "shopchat-data-bucket".



4.2 Run using Docker

4.2.1 Start Docker desktop

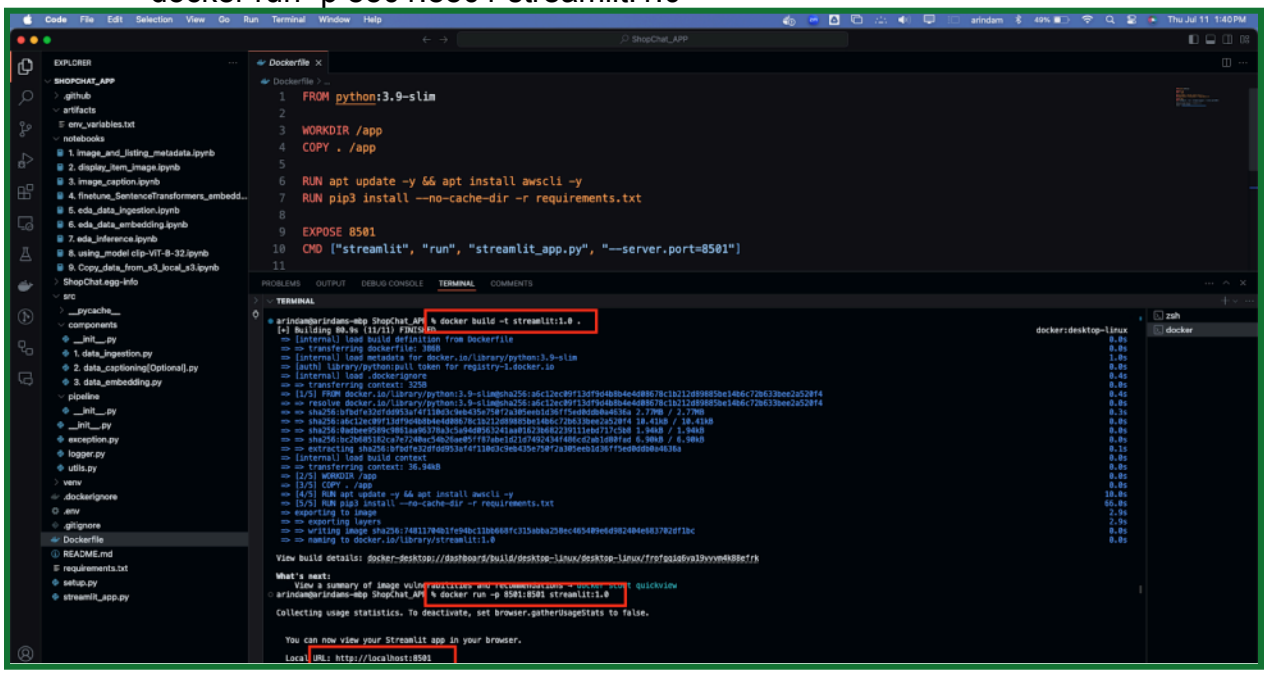
Start docker desktop client in your local system



4.2.2 Build and Run Docker Image

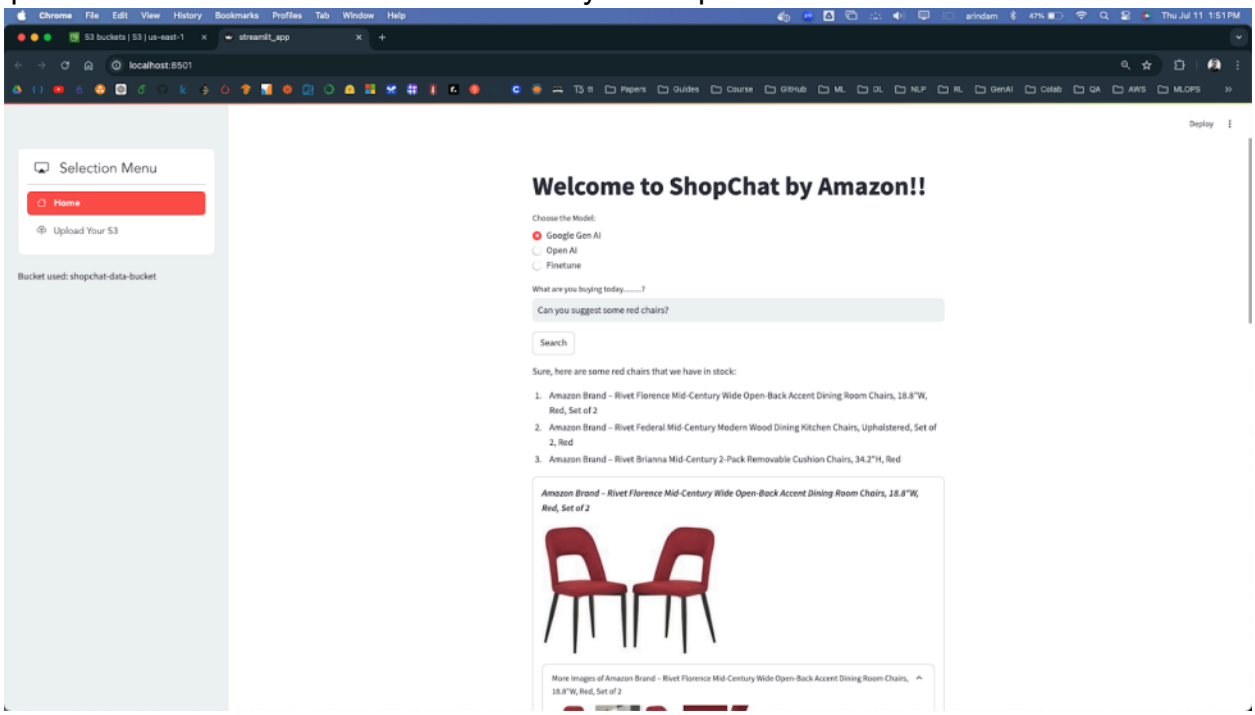
Goto VS code and in a new terminal issue the following commands to build and run the docker image:

- docker build streamlit:1.0 .
- docker run -p 8501:8501 streamlit:1.0

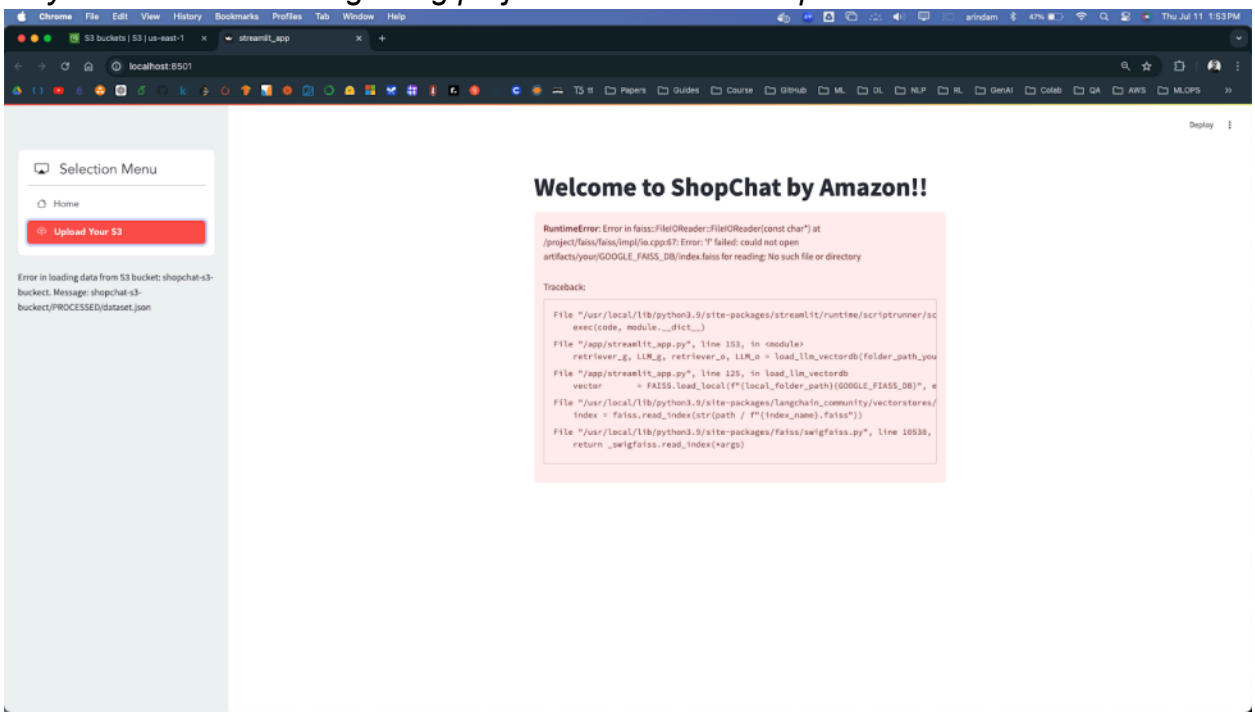


4.2.3 Inference ShopChat app

Go to the Local URL: http://localhost:8501 to interact with the app to ask questions and select different models for your response



Note: If you click on Upload Your S3; this will give error as we have not created S3 yet nut default running using project's S3 bucket "shopchat-data-bucket".



5 Data Loading and Preprocessing

All created data in this process will be loaded to your S3 bucket created in earlier step [YOUR_S3_BUCKET_NAME = "**shopchat-s3-buckect**" as listed in .env file]

5.1 Data Ingestion

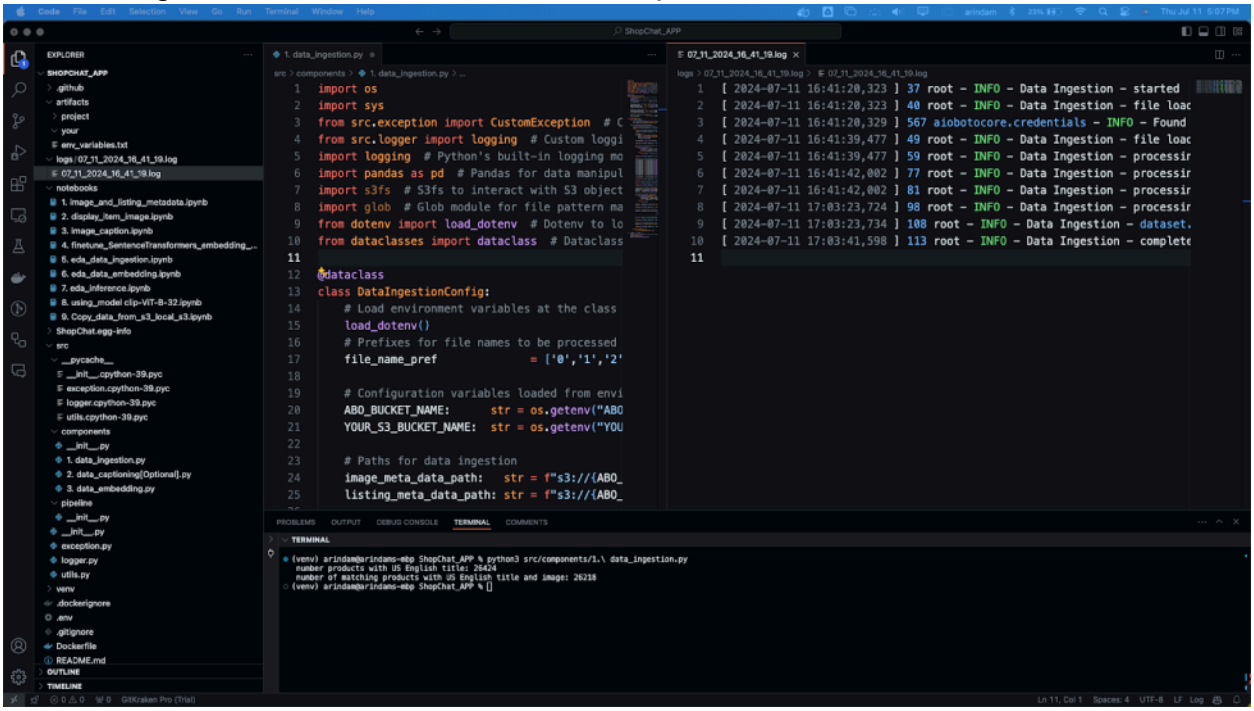
Script Name: “1.data_ingestion.py”
This script reads the data from ABO S3 bucket (amazon-berkeley-objects) https://amazon-berkeley-objects.s3.amazonaws.com/index.html) process the data and create a Json file (dataset.json) and save it to your S3 bucket.

5.1.1 Ingestion Processes

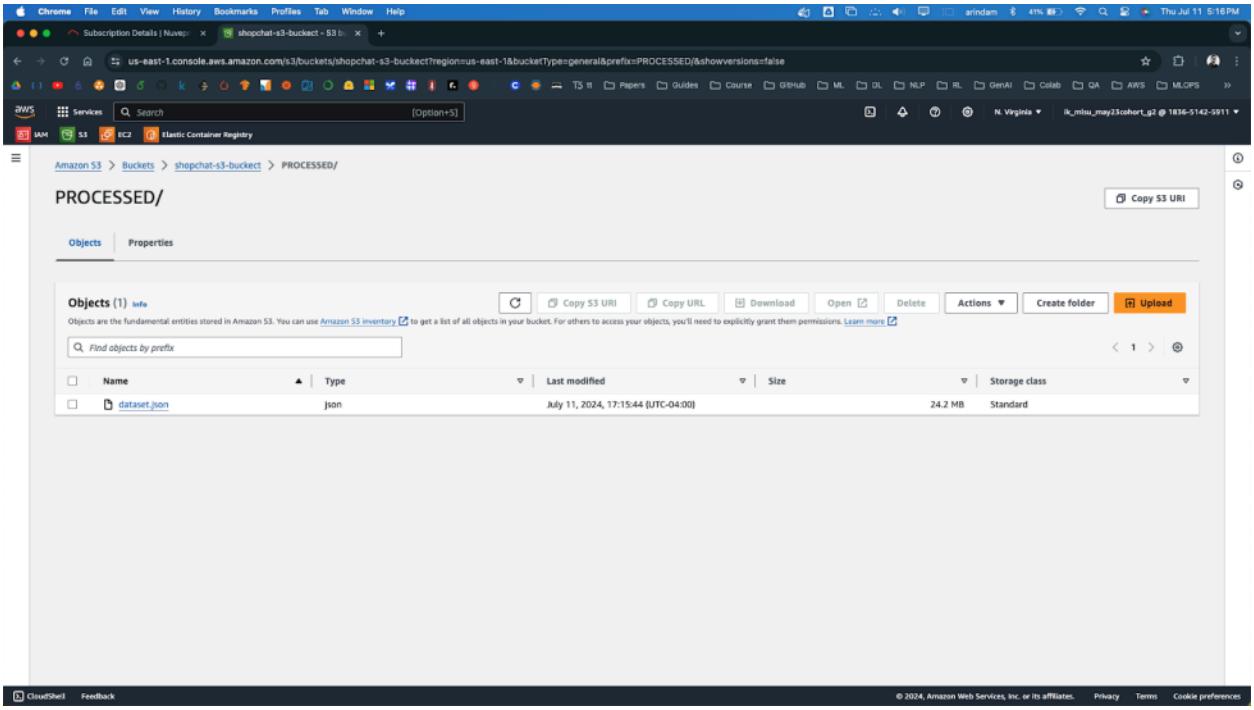
- Loads JSON files of product listings from ABO S3 bucket
- For each listings, extracts US_English metadata fields from the product listings.
- Loads image metadata from ABO S3 bucket
- Merges product listings metadata with image metadata to get the image paths for product image and additional image paths of the product
- Drops unnecessary columns and saves the processed dataset as Json file in project S3 bucket.

5.1.2 Running data ingestion

View the log file and S3 bucket. *Note: This script runs about 20-25 munities*



Entry in S3 in PROCESSED folder



5.2 Data Captioning [Optional Step]

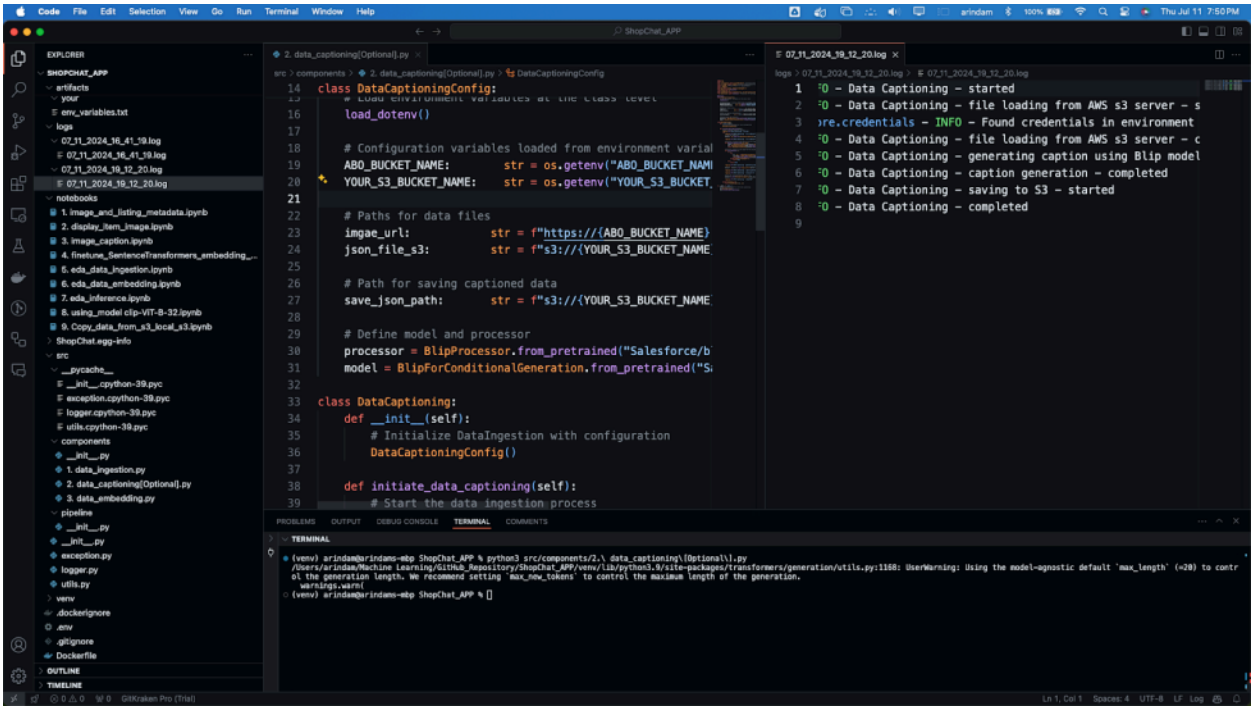
Script Name: “2. data_captioning[Optional].py”
This script reads the Json dataset created in previous step from S3. For every item image captioning is done using SoTA Salesforce BLIP model.
Captioning each image takes time and is not a **Must Have feature** for embedding hence this script is made optional but vector search does perform well if added in text embedding

5.2.1 Captioning Processes

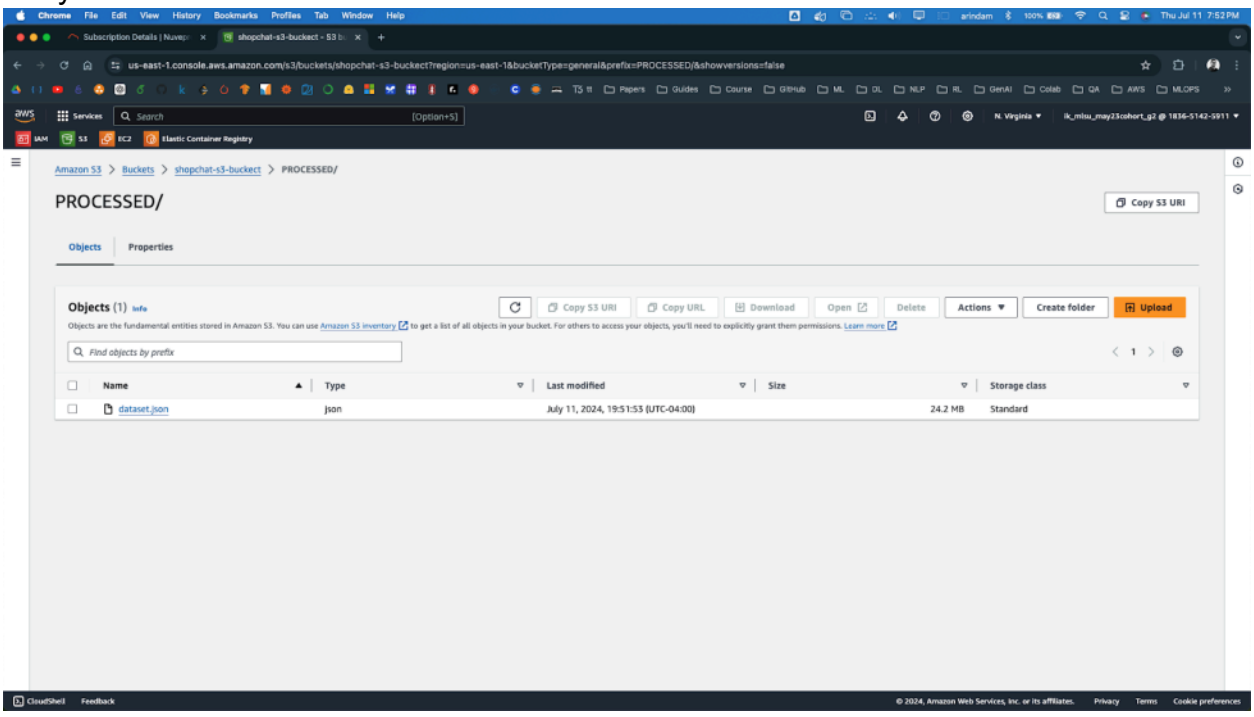
- Loads dataset JSON file created in ingestion process
- For each product, gets the image path
- Loads the images from ABO S3
- Sends the image to BLIP model for captioning
- Adds the caption back to the Json dataset.
- Saves back the Json file in project S3 bucket

5.2.2 Running data captioning

The caption is added back to the same Json dataset in your S3 bucket.
Note: This scrip runs for long time (more than 3 hours depending on the system config) and hence it is an optional script for this process. If used, then gives better results during inference thru LLM.



Entry in S3 in PROCESSED folder



5.3 Data Embedding

Script Name: “3. data_embedding.py”
This script reads the Json dataset created in previous step from S3. Uses Langchain JSONLoader and Google embedding and(or) OpenAI embedding and save to FAISS vector database.

5.3.1 Embedding Processes

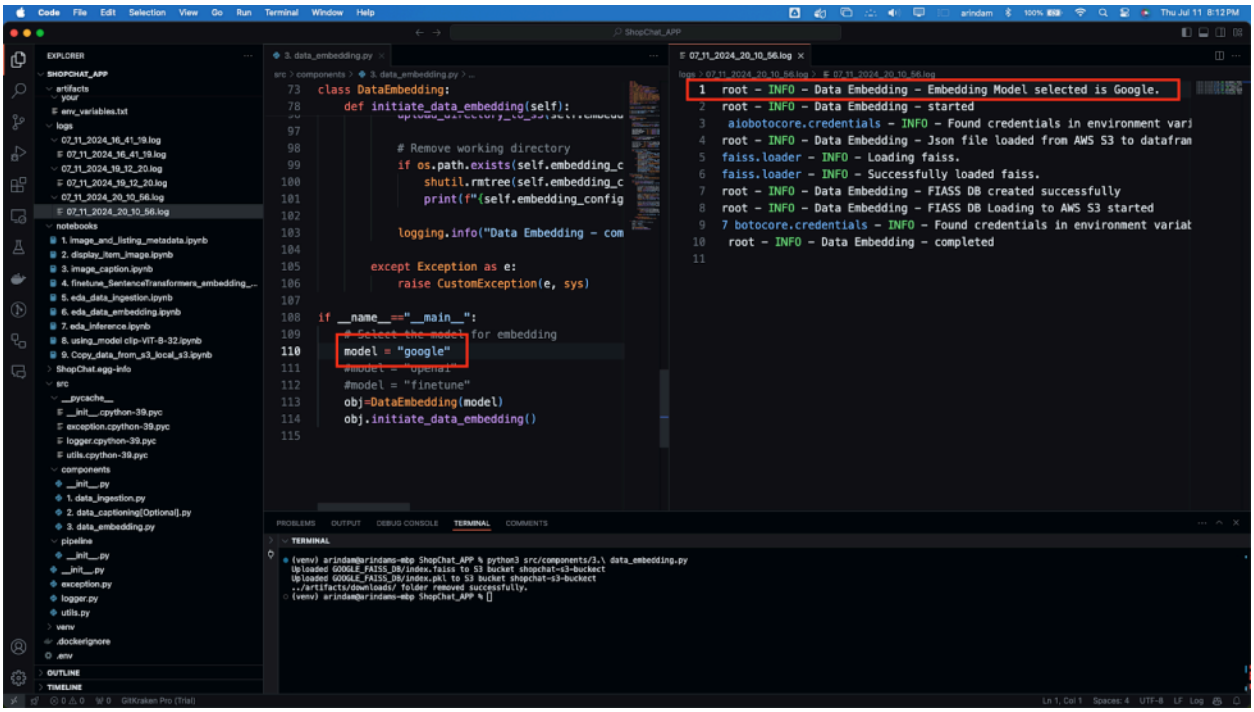
- Loads dataset.json file from project S3 bucket
- Converts the Json list of objects into LangChain Documents objects using JSONLoader, which is compatible with the embedding generation.
- Create Vector Store: Utilizes FAISS, a highly efficient library for approximate nearest neighbor search in the vector space, to create a vector store from the documents using the specified embedding model.
- Saves the vector database to S3

Models used for Embedding are:

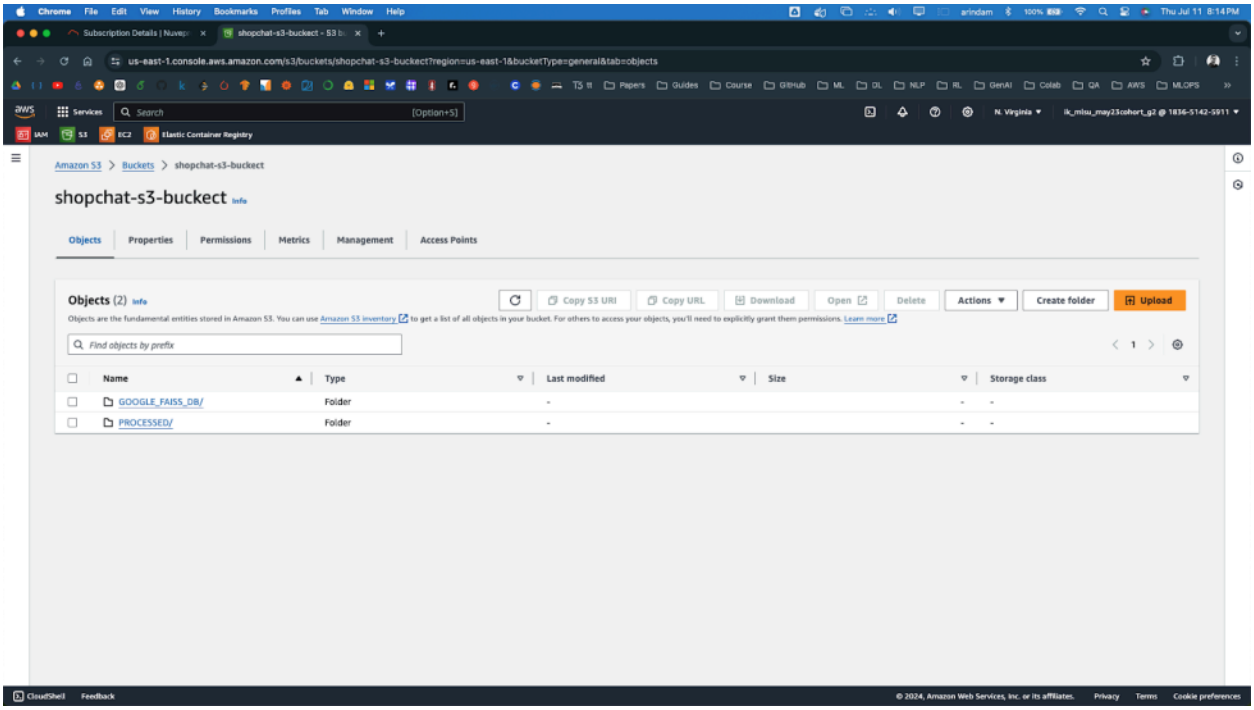
- GoogleGenerativeAIEmbeddings: For Google's embedding model.
- OpenAIEmbeddings: For OpenAI's model.
- ~~HuggingFaceEmbeddings: For a fine-tuned model loaded from S3~~

5.3.2 Running Google Embedding

We will be using GoogleGenerativeAIEmbedding model from Langchain google genai. Script is set to use "google" from the selection menu. *Note: This script runs about 20-25 munities*

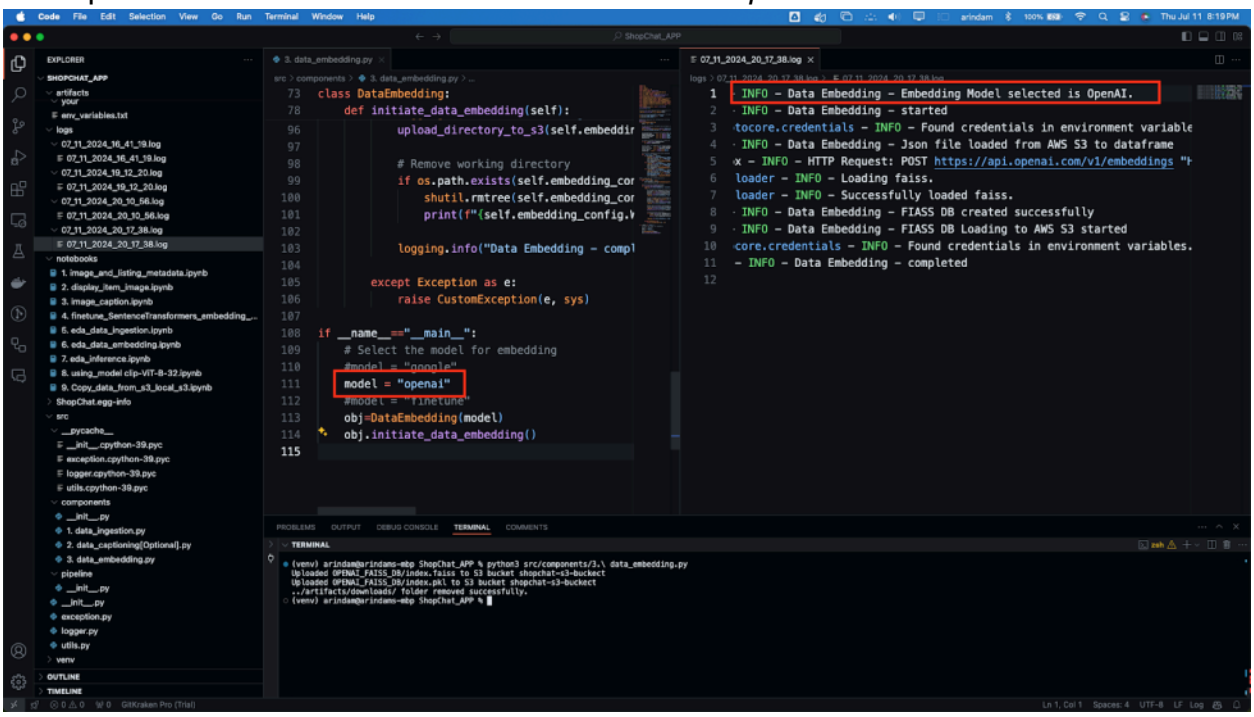


Entry in S3 GOOGLE_FAISS_DB folder

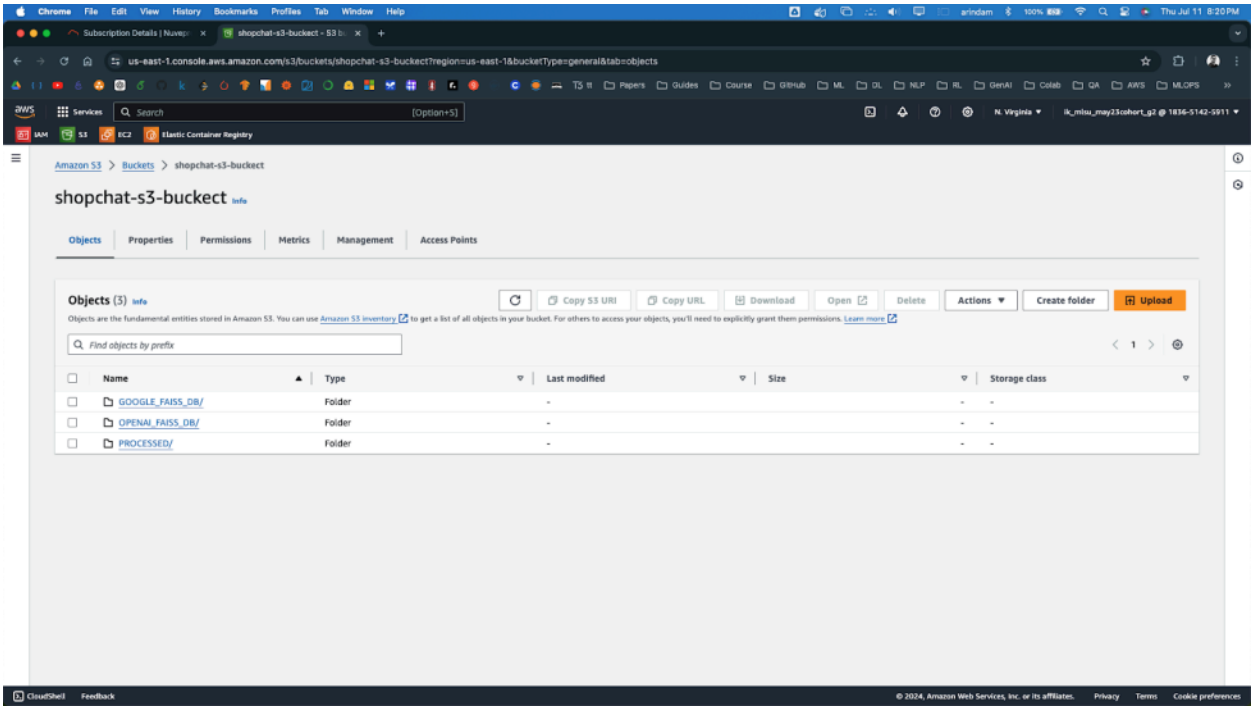


5.3.3 Running OpenAI Embedding

We will be using OpenAIEmbedding model from Langchain_openai. Script is set to use “openai” from the selection menu. *Note: This script runs about 20-25 munities*



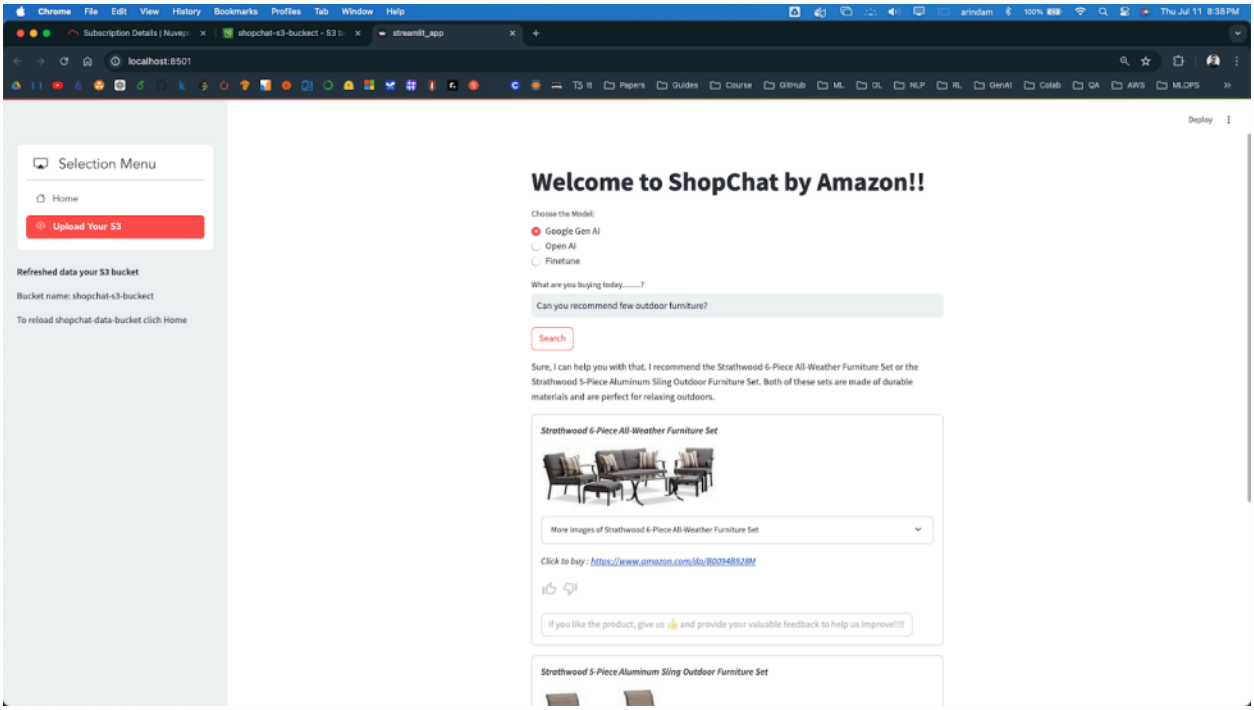
Entry in S3 OPENAI_FAISS_DB folder



6 Full inferencing after data loading

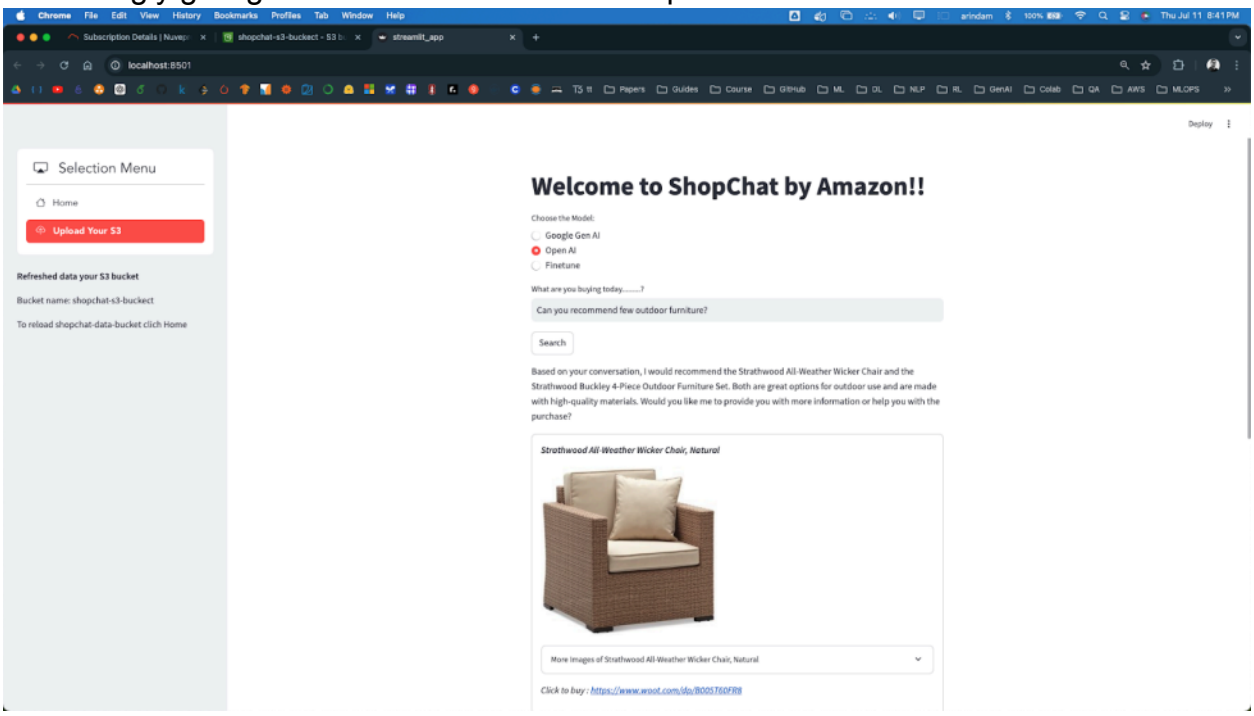
6.1 Upload Your S3

Now we can rerun the ShopChat app using any of the methods described earlier it must not give any error while uploading your S3 bucket. Now using your S3 data to give the results.



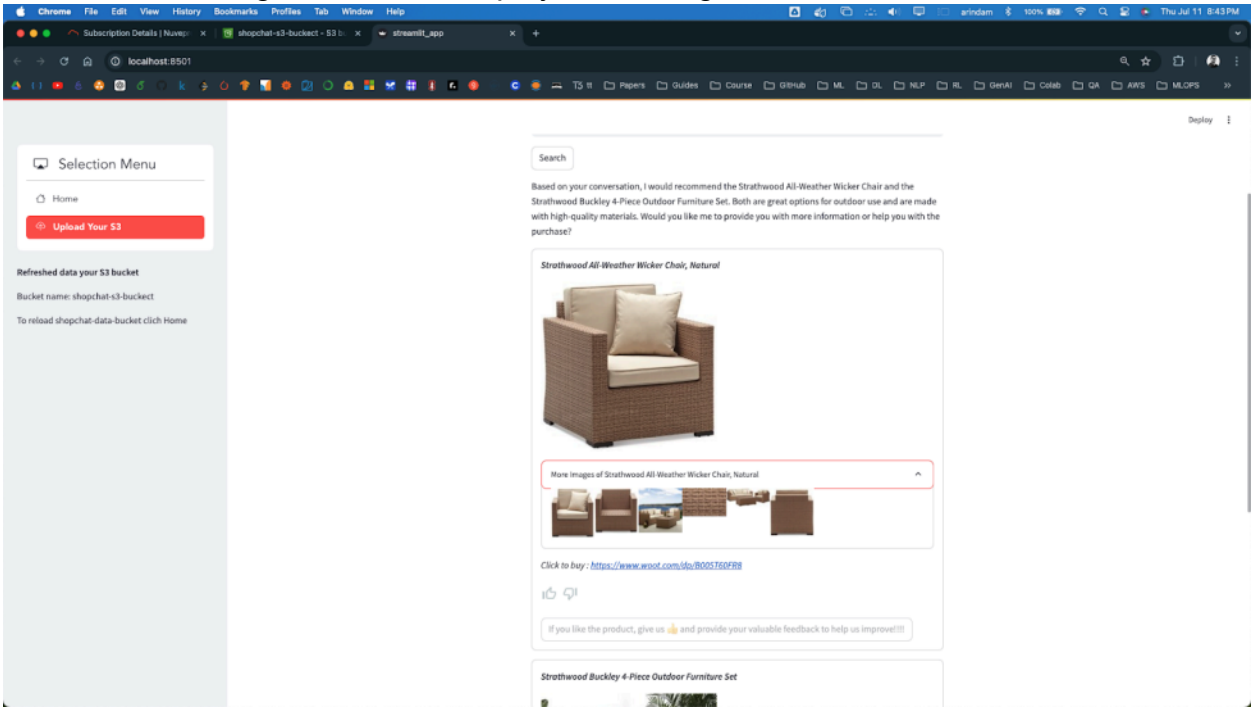
6.2 Selection of LLM model

Changing LLM is easy just select any of the listed models to use for inference.
Interestingly giving different result when same question is asked ☺

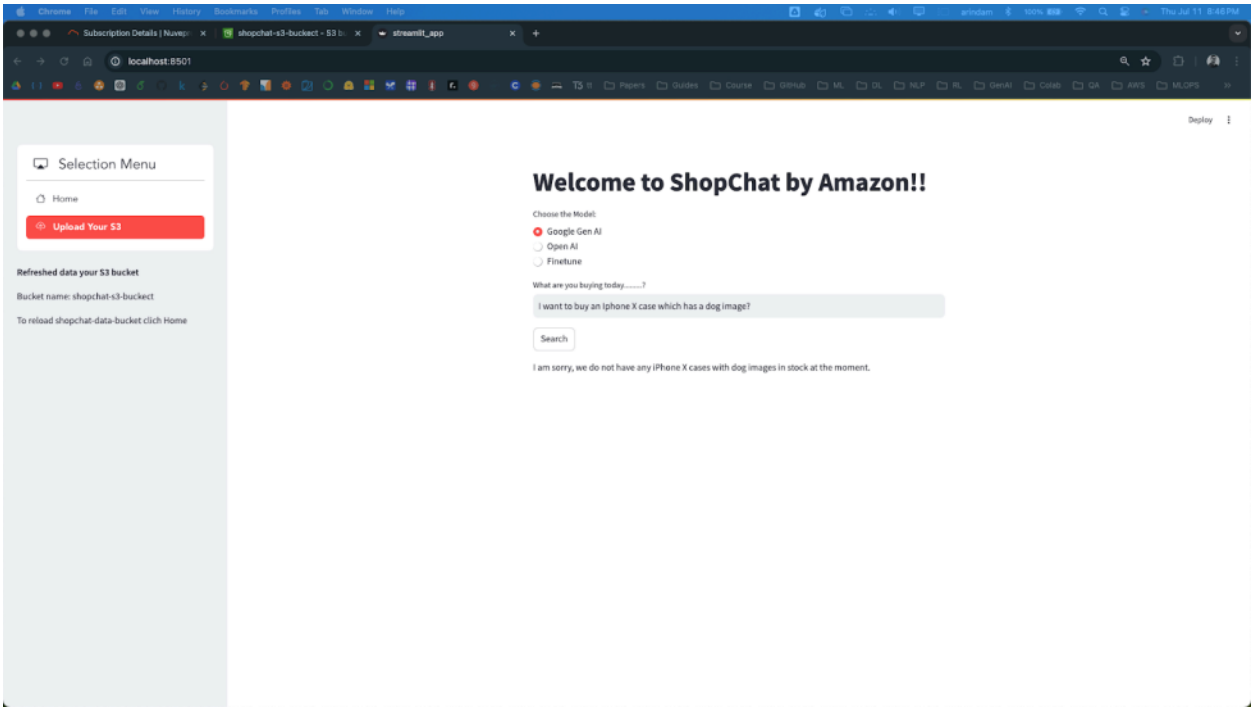


6.3 Display more images

Click on more images and it will display more images of the item if available.

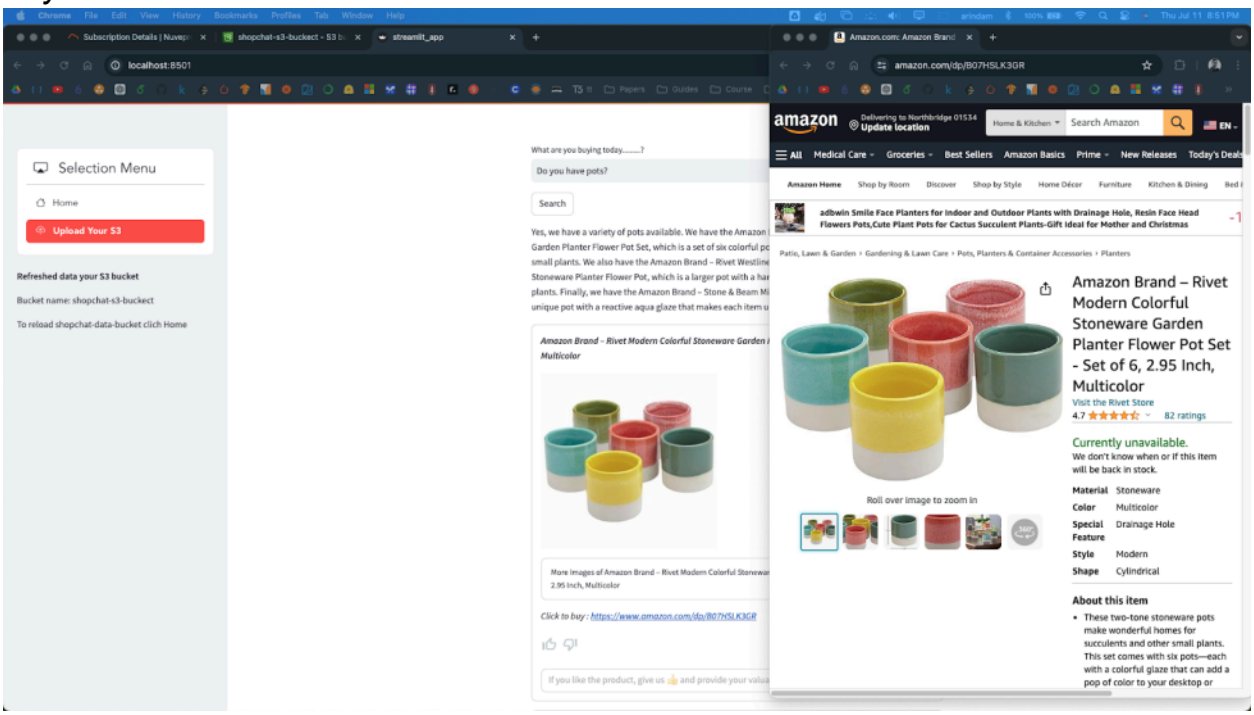


6.4 Item not found



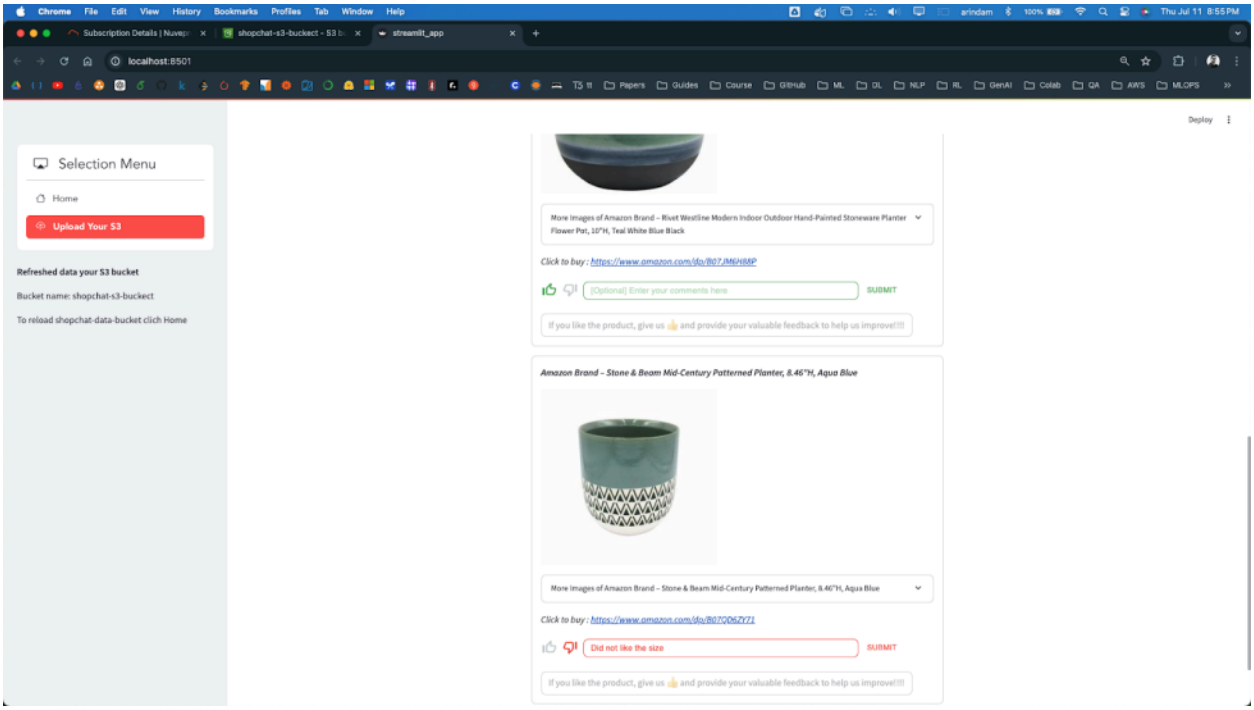
6.5 Listing Link

On clicking, link took me to the Amazon.com website for the product listing. *Note – may not work for all items as warned at ABO site itself.*



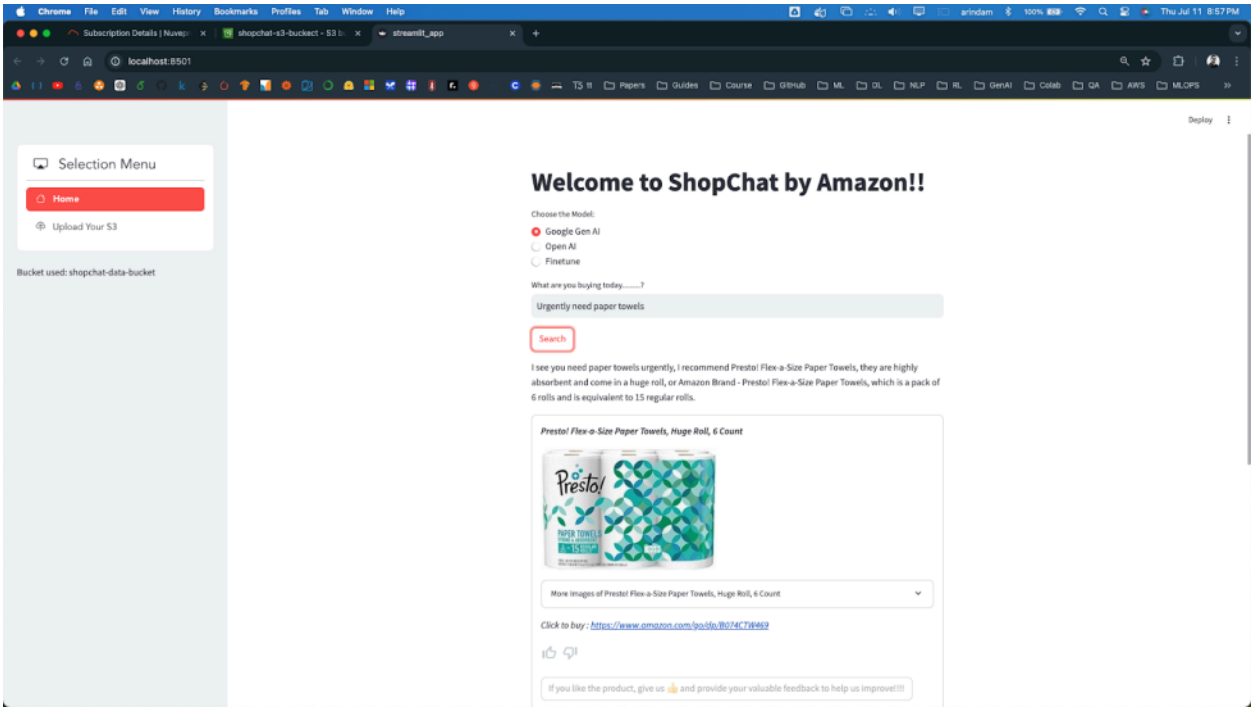
6.6 Feedback

You can like or dislike the product listing and provide your feedback



6.7 Home button

You can click Home button on the left side to refresh the data with default S3 bucket



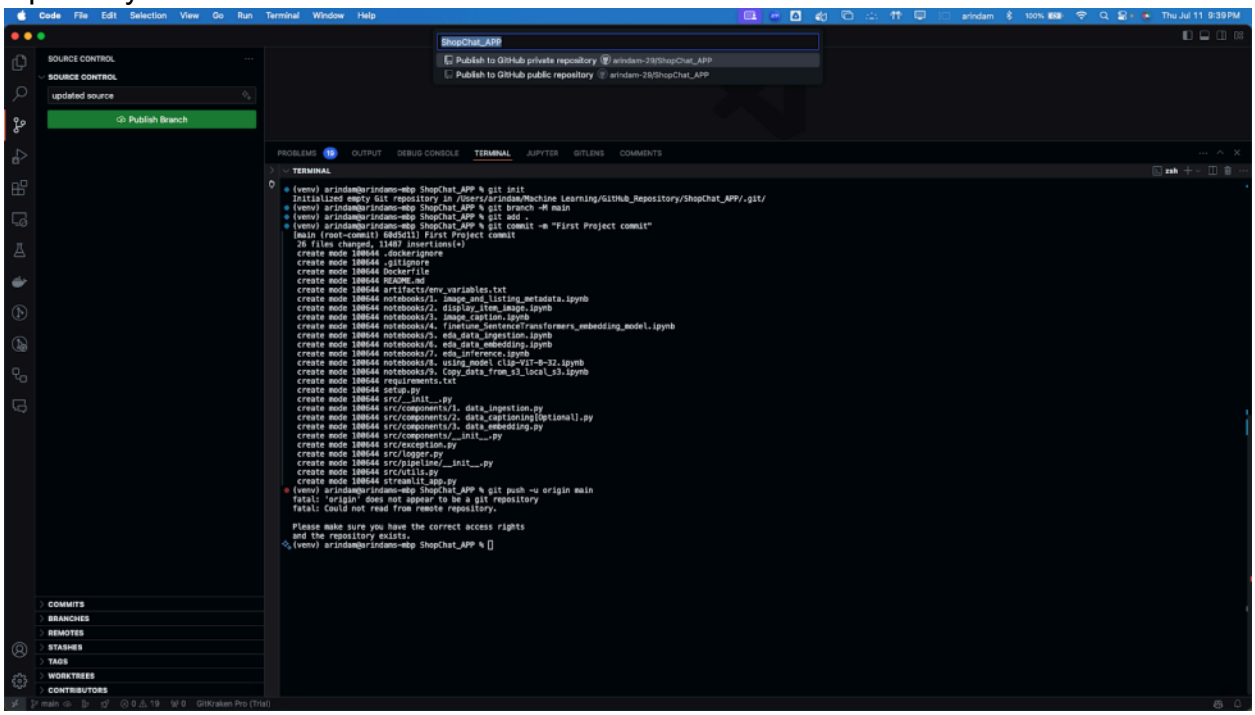
7 CICD with GitHub Action and AWS

7.1 Create Git Repository

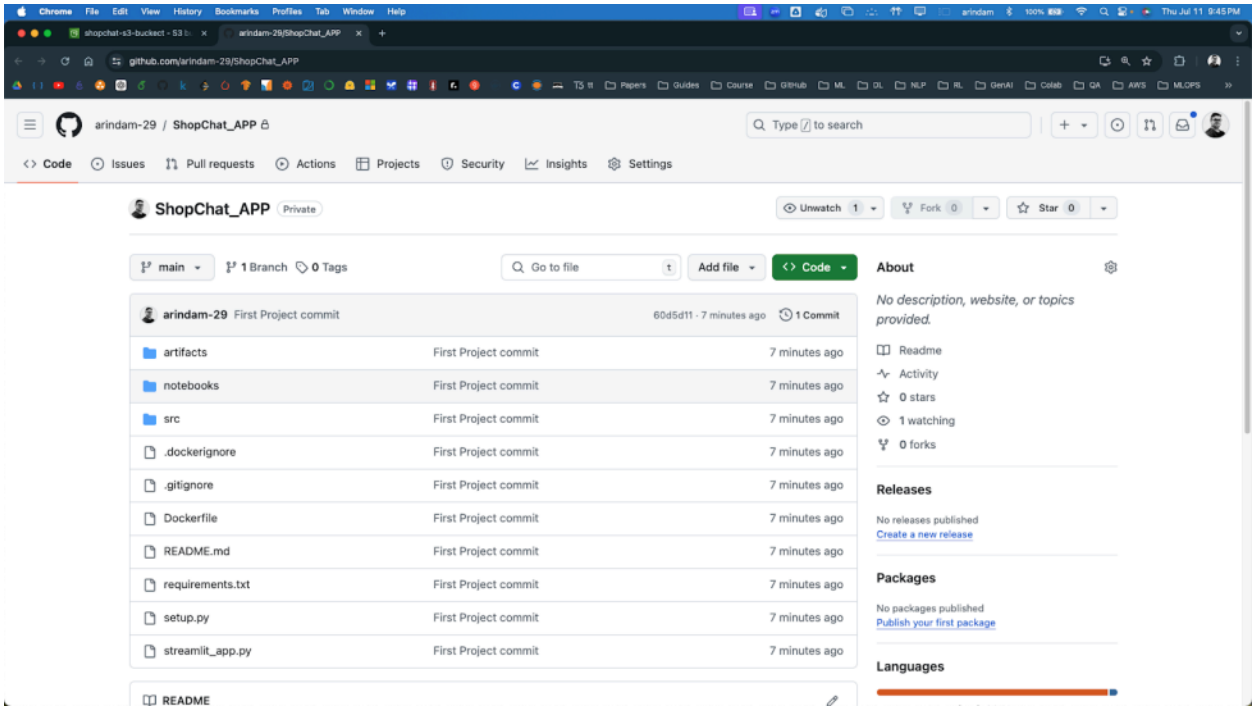
Run Following commands on VSCode terminal within this folder, this will create a GitHub repository (ShopChat_App)

- git init [initialize the git in this folder]
- git branch -M main [changing the branch to main from master]
- git add . [add all files]
- git status [check the file status]
- git commit -m “First git setup for the project” [first commit to the project]
- git push -u origin main [push the changes to github]

Go to the VS Code and click on git icon and make sure to select private repository.

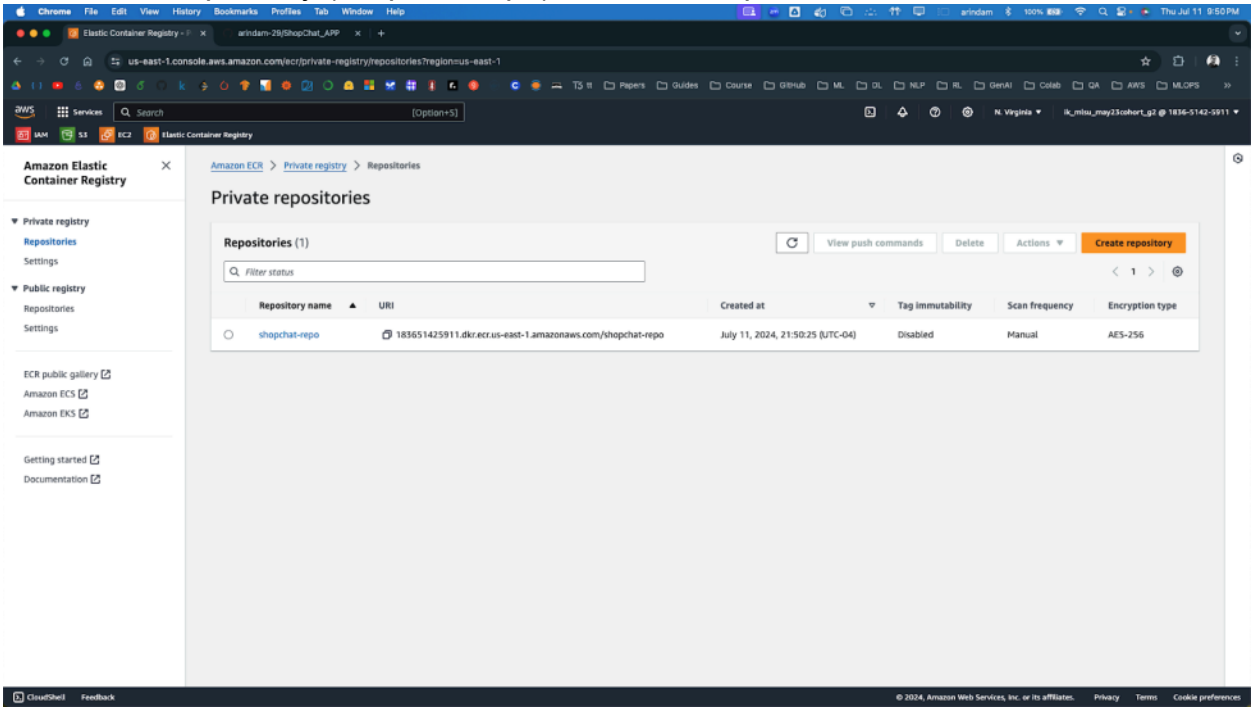


Check in GitHub to make sure repository is created



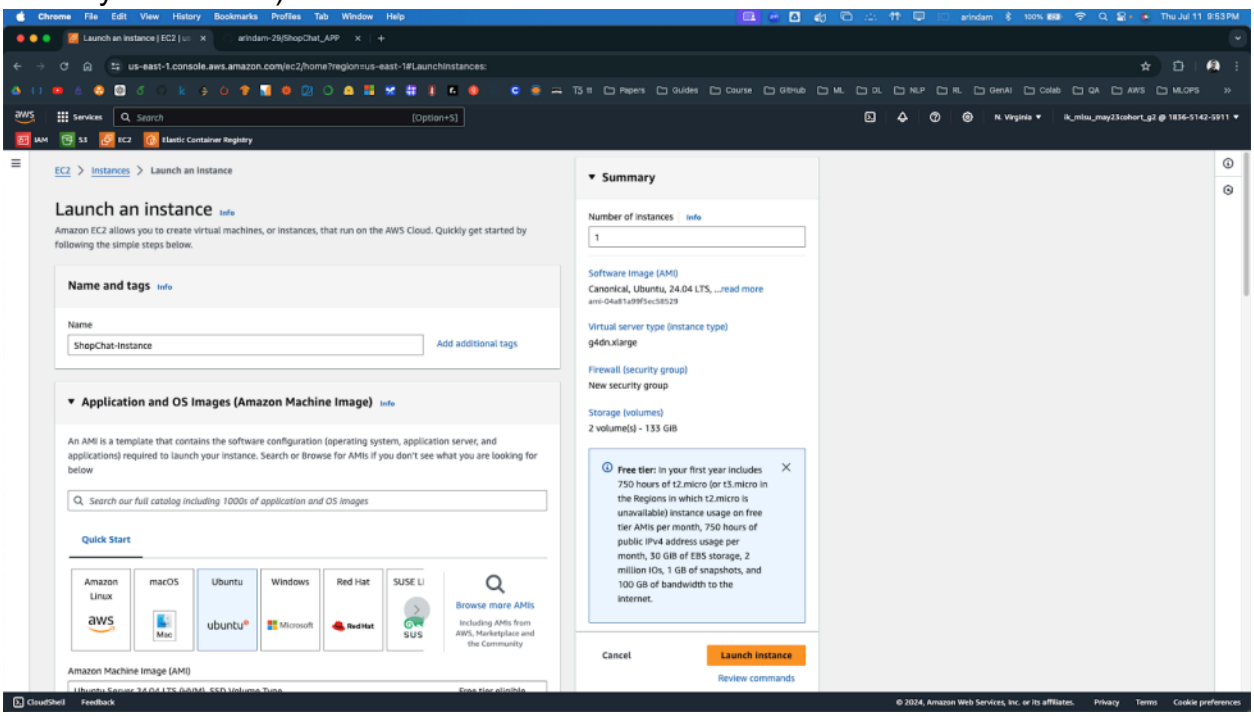
7.2 Create AWS ECR Repository

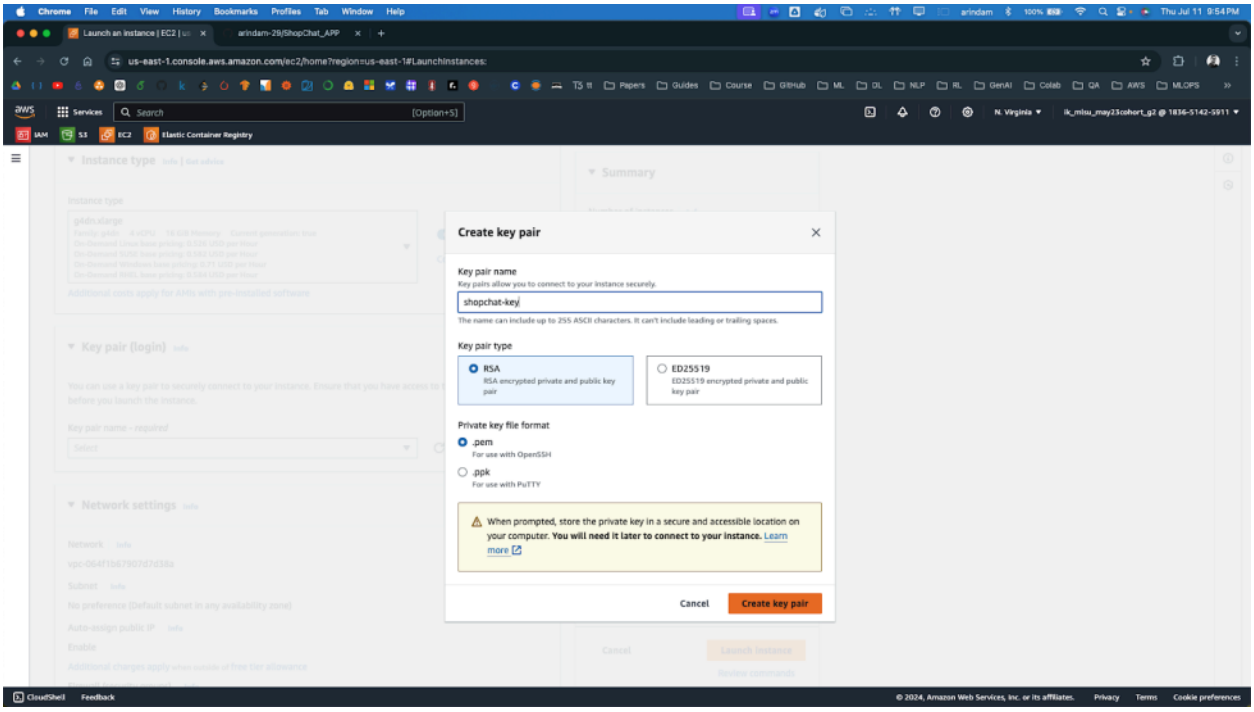
Log into your AWS account and search for ECR, select “private repository”, give a name to the repository (shopchat-repo) and rest all options are left default.



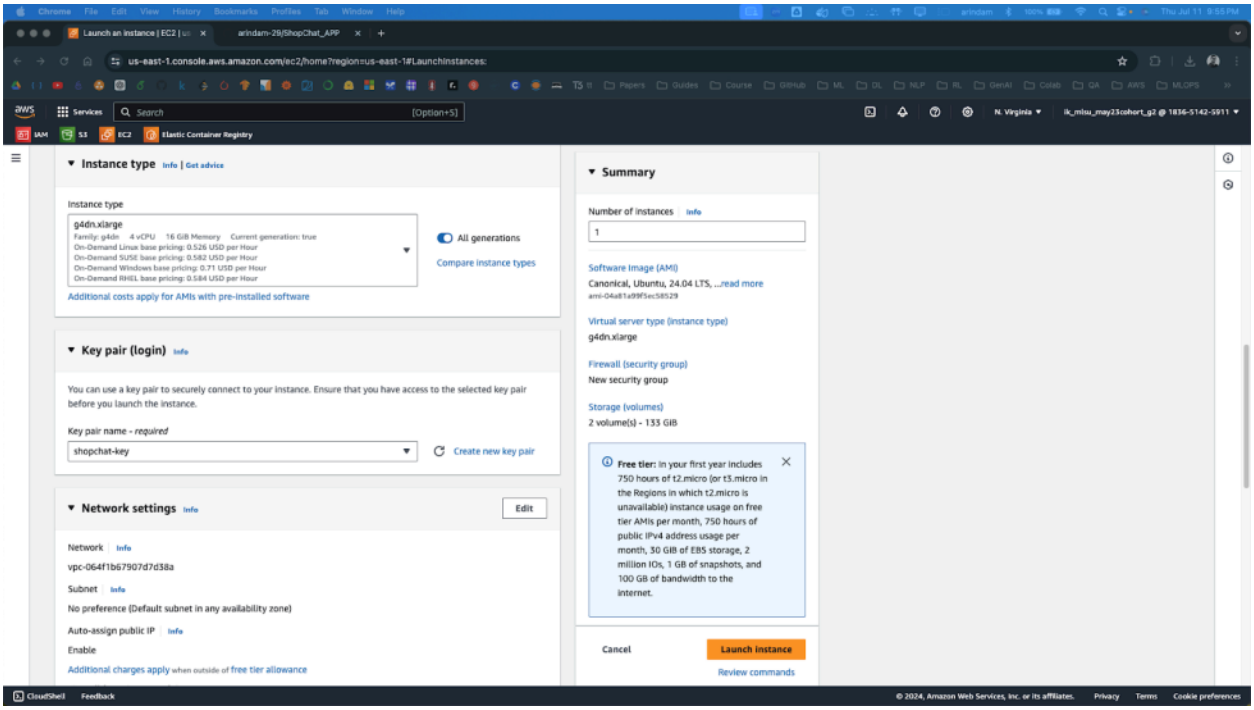
7.3 Create AWS EC2 Instance

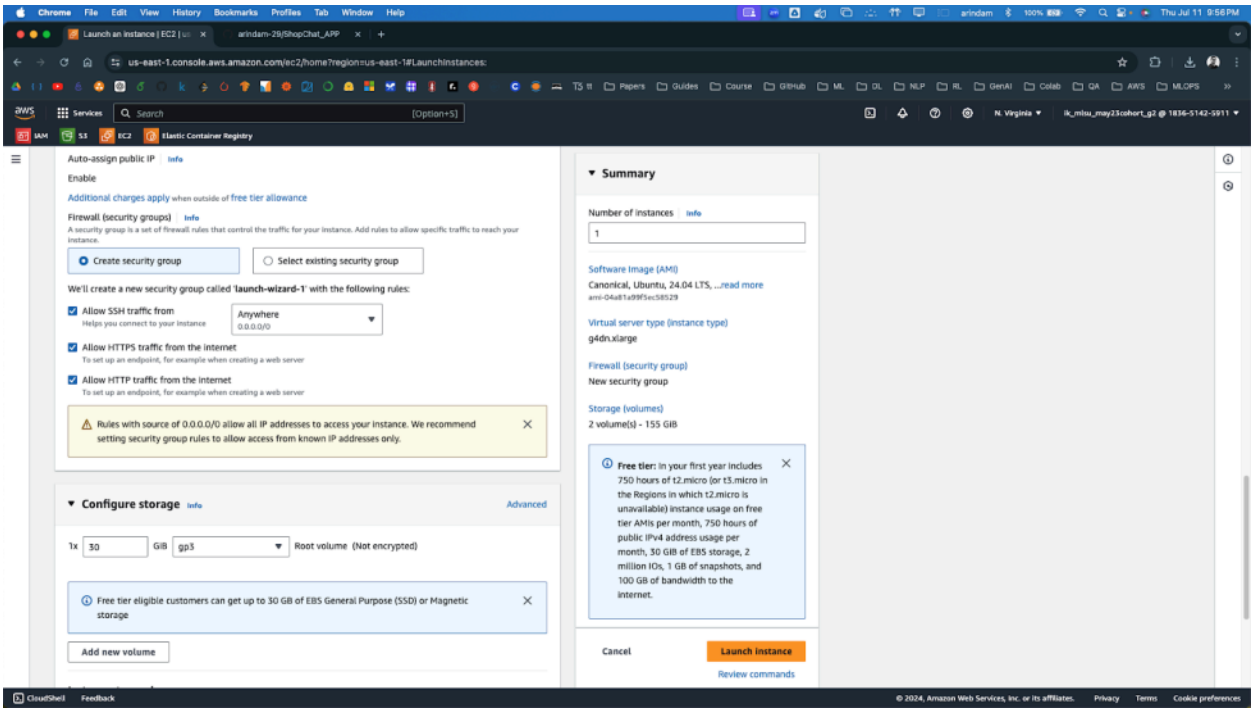
Go to AWS account, search for EC2 and launch a new EC2 instance with following suggested configuration (can be different as well, but make sure at least 16gb of memory is selected)



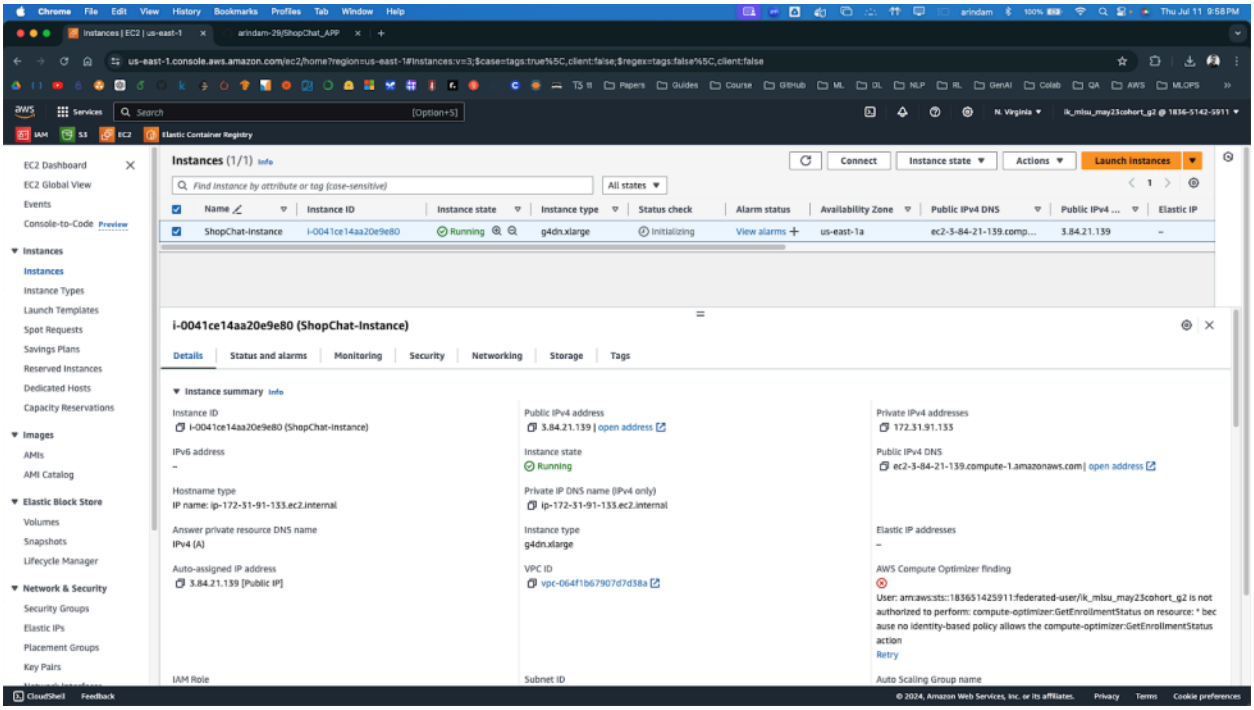


Note: Download the new key pair, which will be needed later to log into the instance

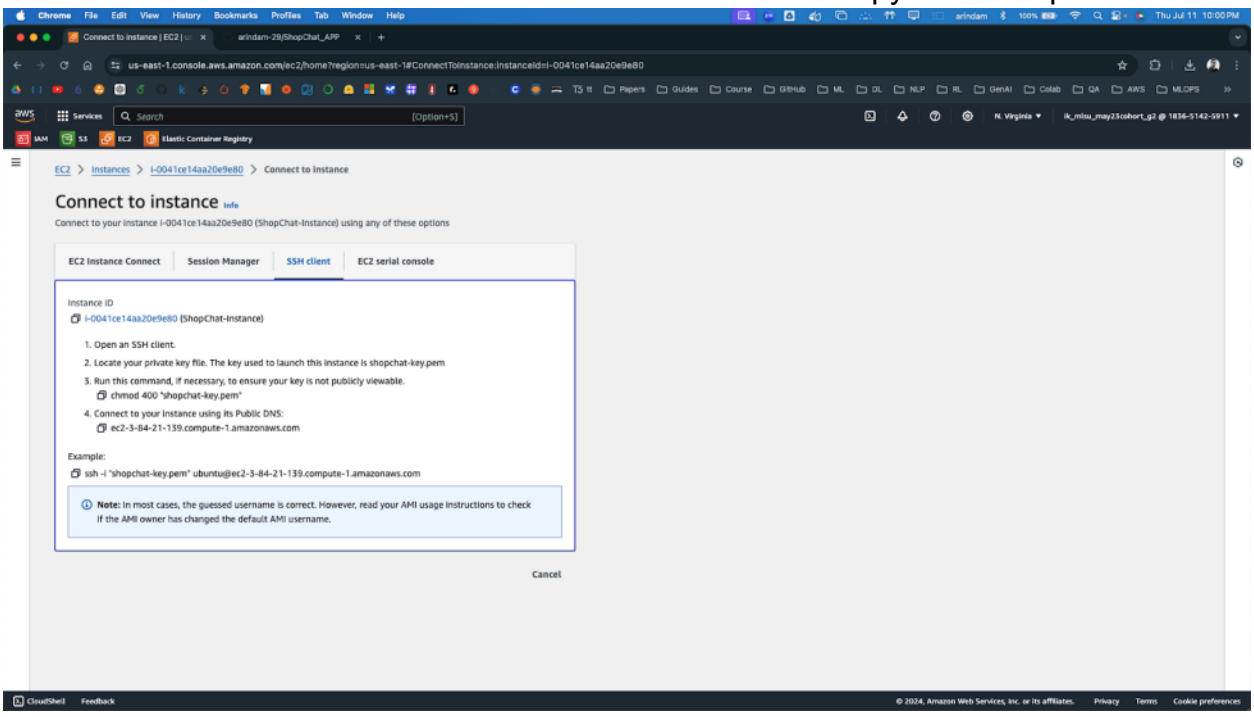




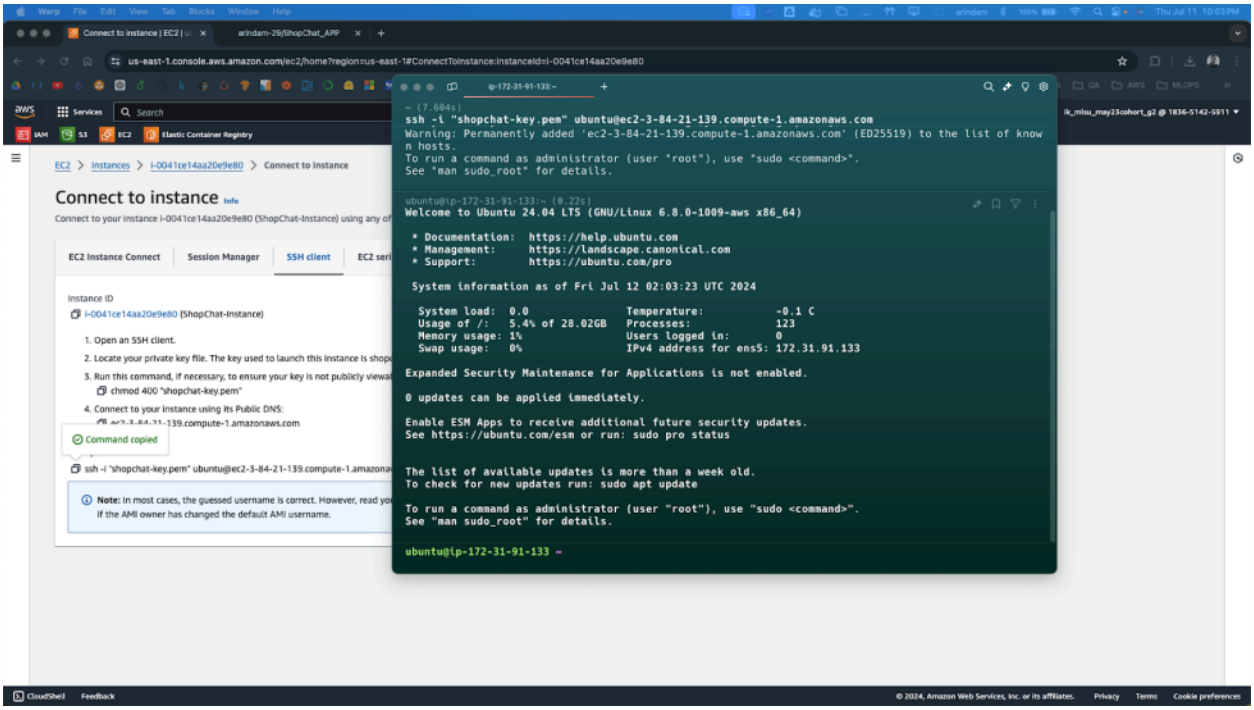
Instance is created



Go to Connect and SSH client. Follow the instructions and Copy the Example.

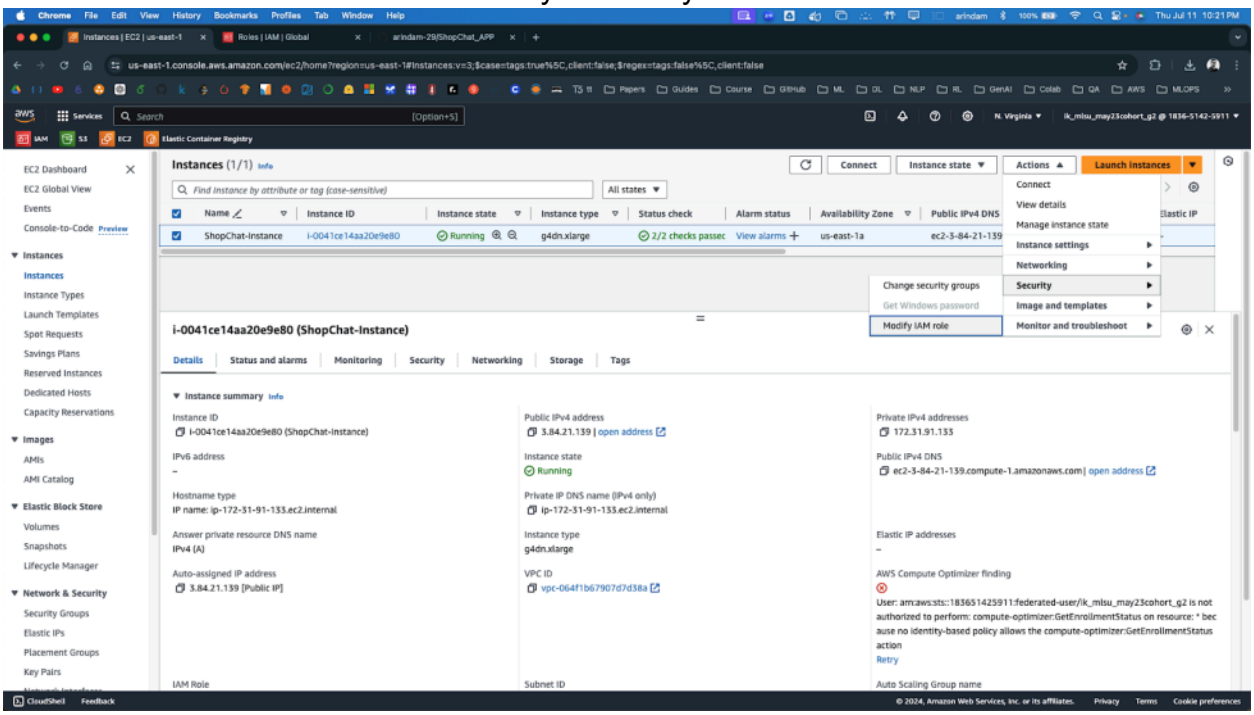


Logged in to the instance using ssh client

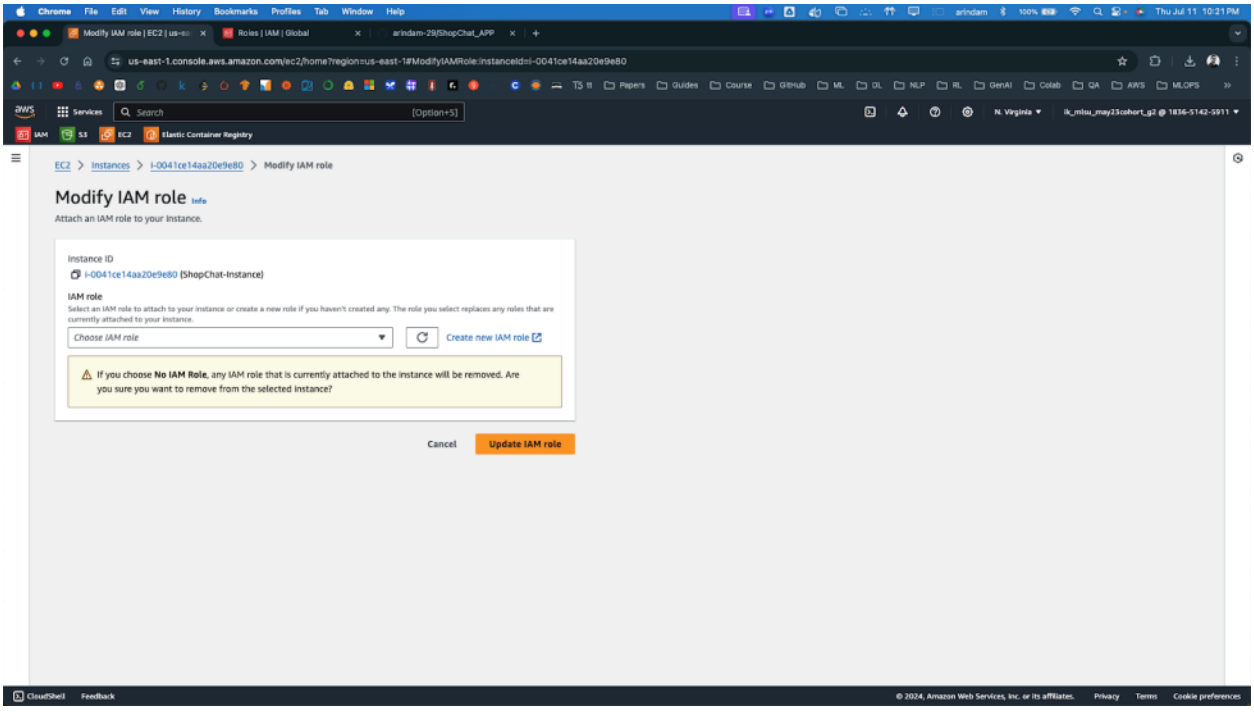


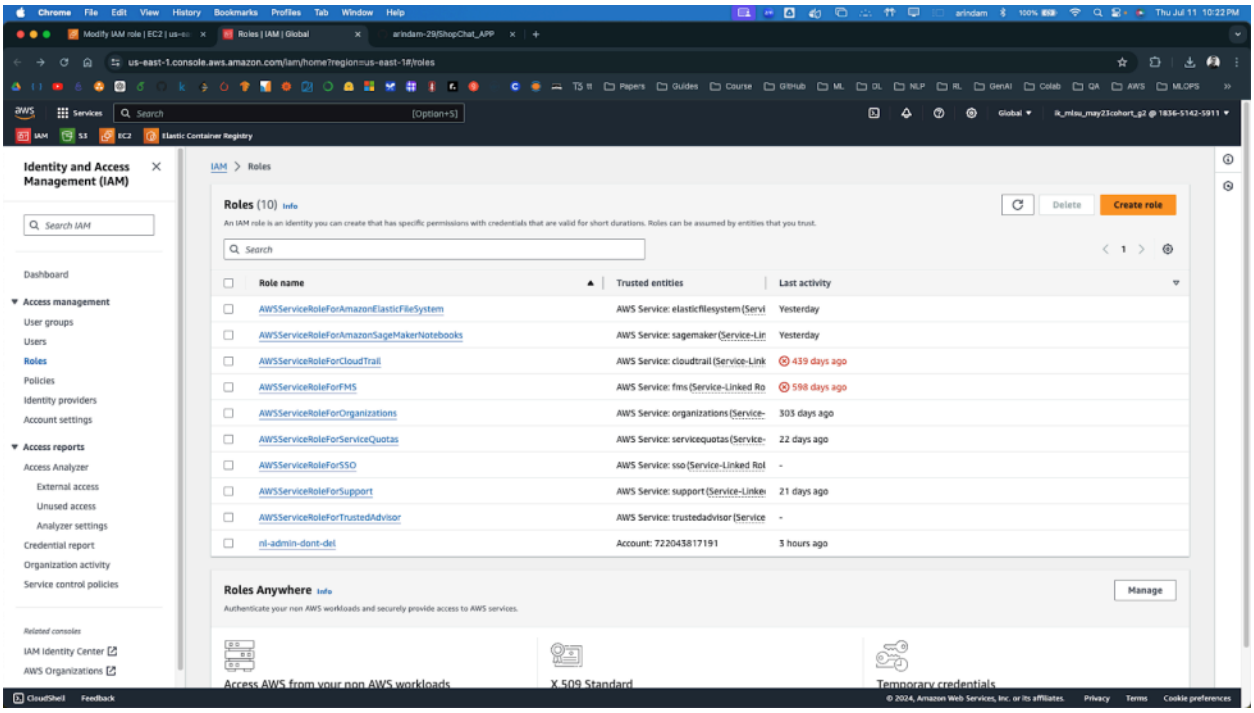
7.4 Create IAM Role and add to EC2 instance

Now we need to give necessary access to S3 and ECR to the new EC2 instance so that it can access both the resources. Go to the main page of EC2. Select the instance and click on Action -> Security -> Modify IAM role

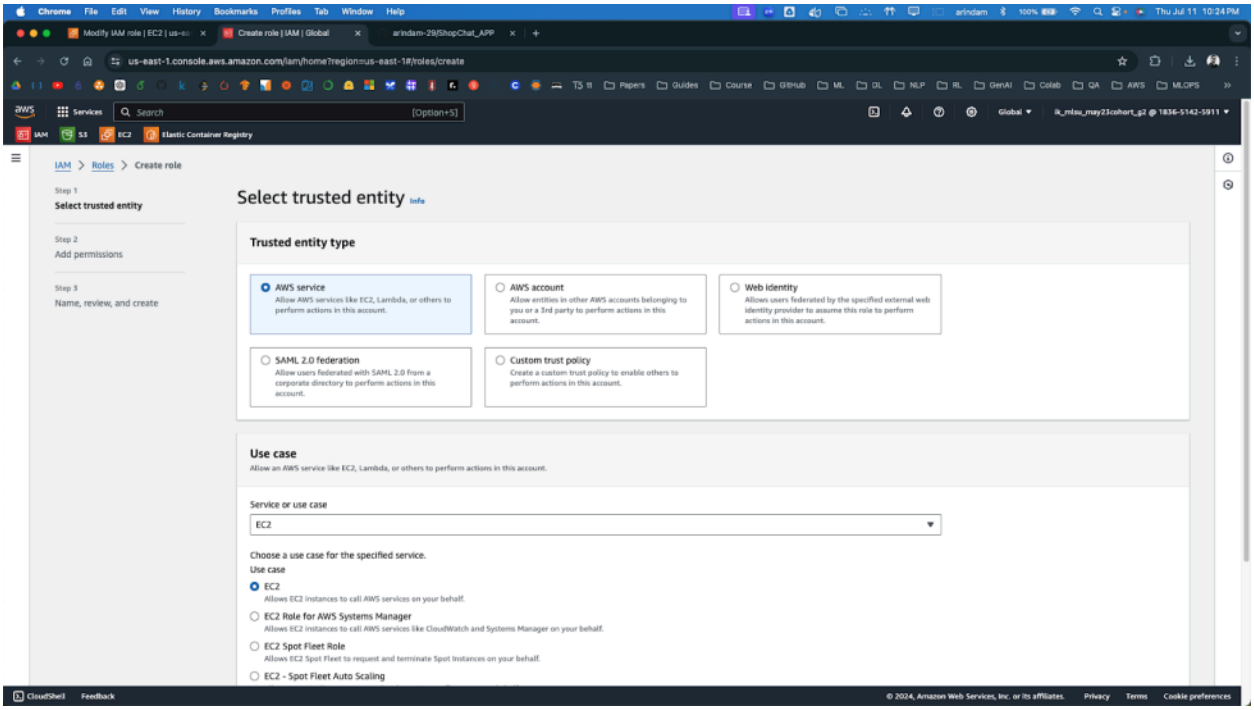


From here select “Create new IAM role”





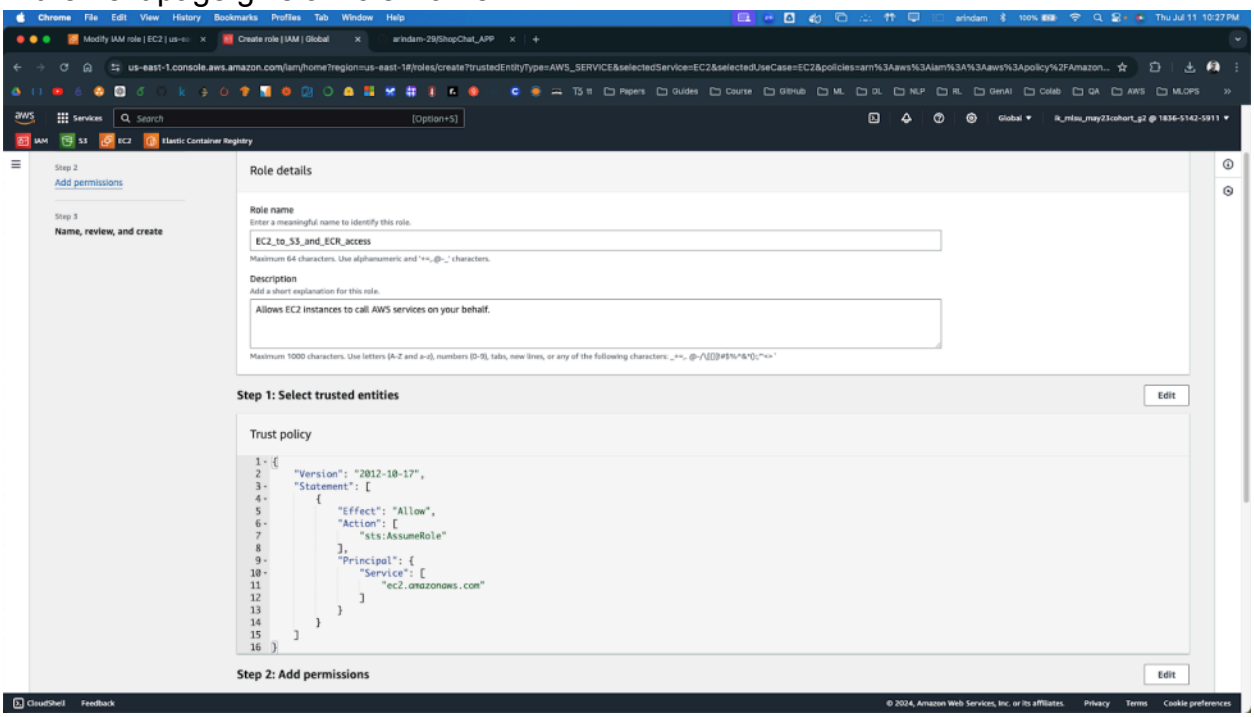
Click on Create role and select Use Case as EC2 and click next



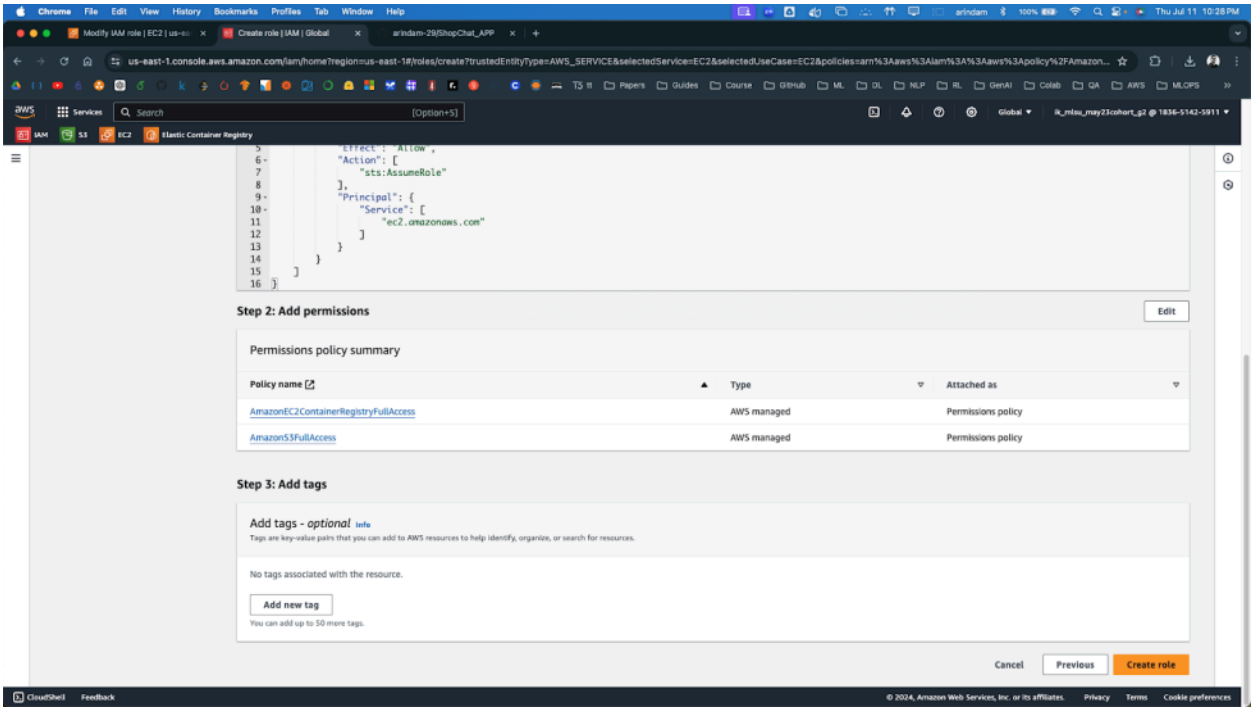
Select policy permission

- AmazonEC2ContainerRegistryFullAccess
- AmazonS3FullAccess

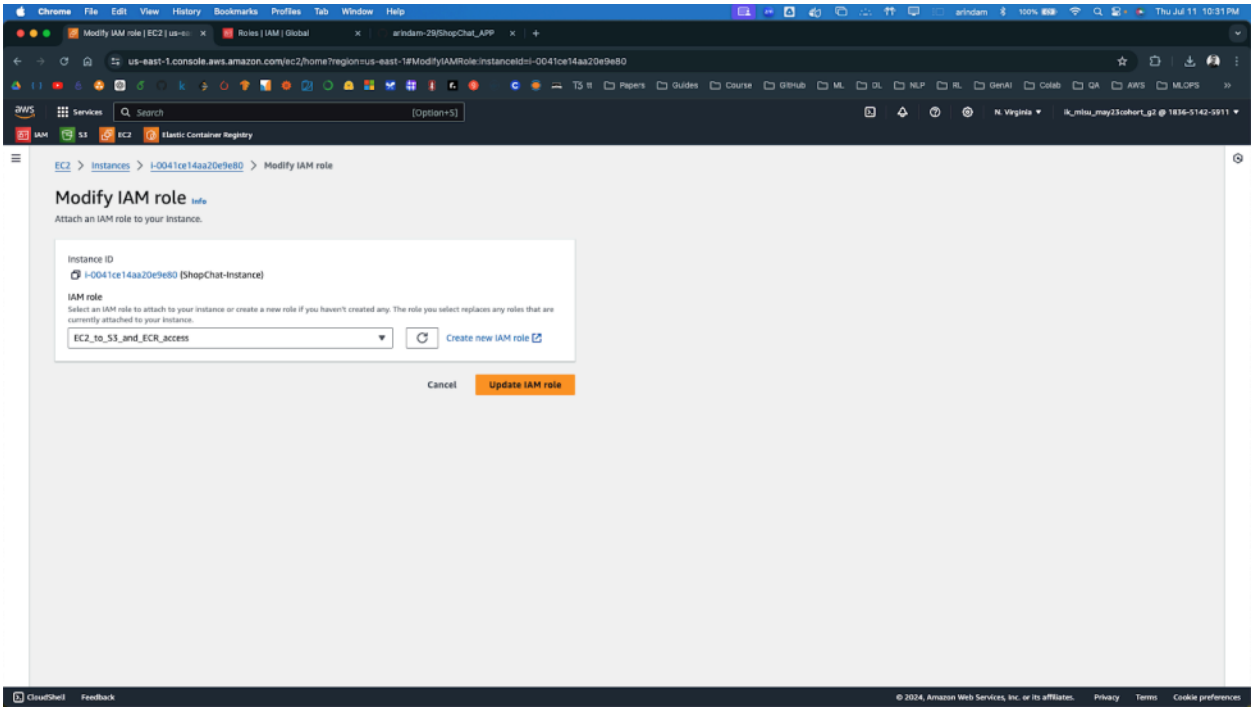
In the next page give a Role Name



Click on create role



Go back to Modify IMA role page, refresh the IAM role and select the one you have created and click Update IAM role



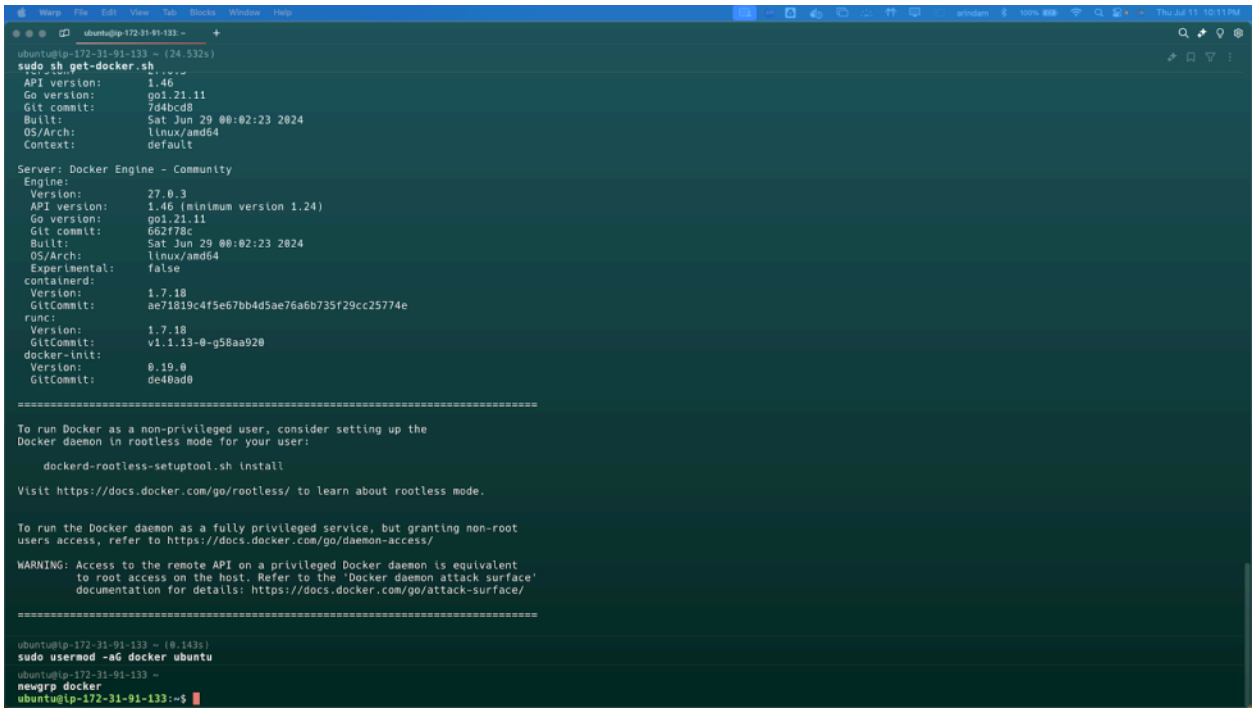
IAM role is now added to EC2 instance

7.5 Install Docker into EC2 instance

Run the following commands one by one in EC2 ssh client to install and run docker:

- sudo apt-get update -y
- sudo apt-get upgrade
- curl -fssl https://get.docker.com -o get-docker.sh
- sudo sh get-docker.sh
- sudo usermod -aG docker ubuntu
- newgrp docker

Docker is now installed in EC2



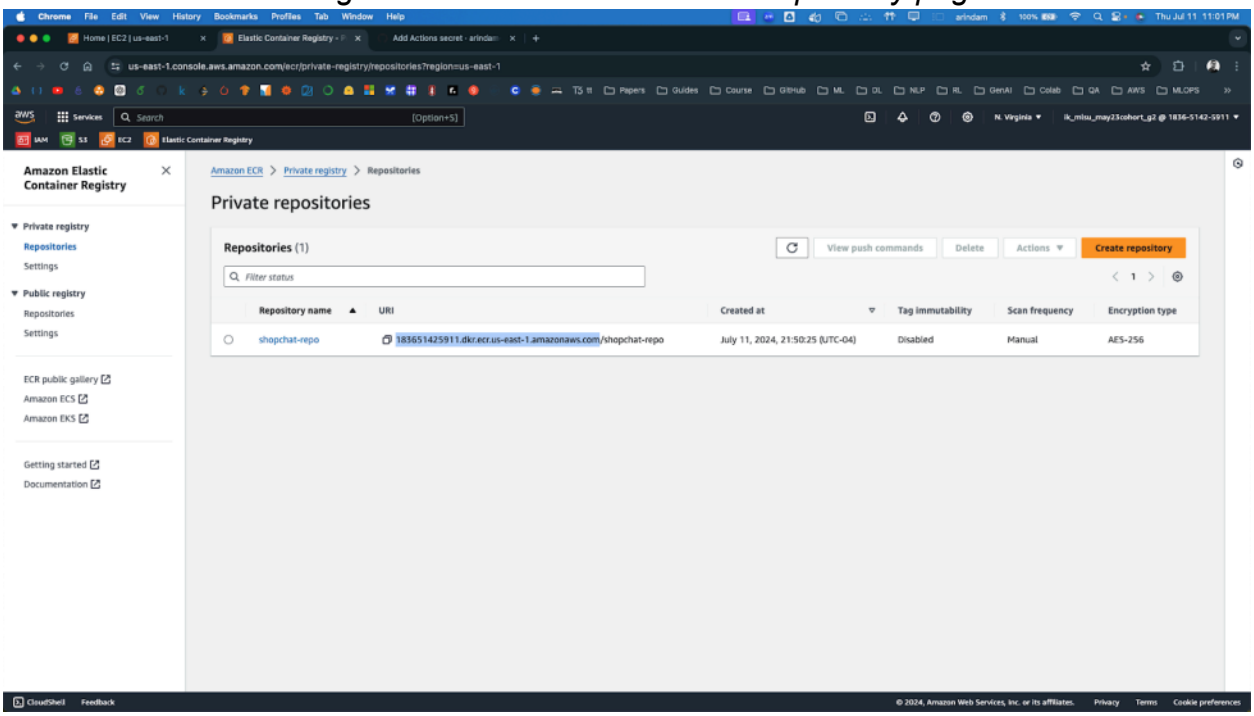
7.6 Add Secrets to GitHub Actions

Add following secrets. Got to GitHub repository -> Settings -> Securities -> Secrets and Variable -> Actions -> **Click on New repository secret**

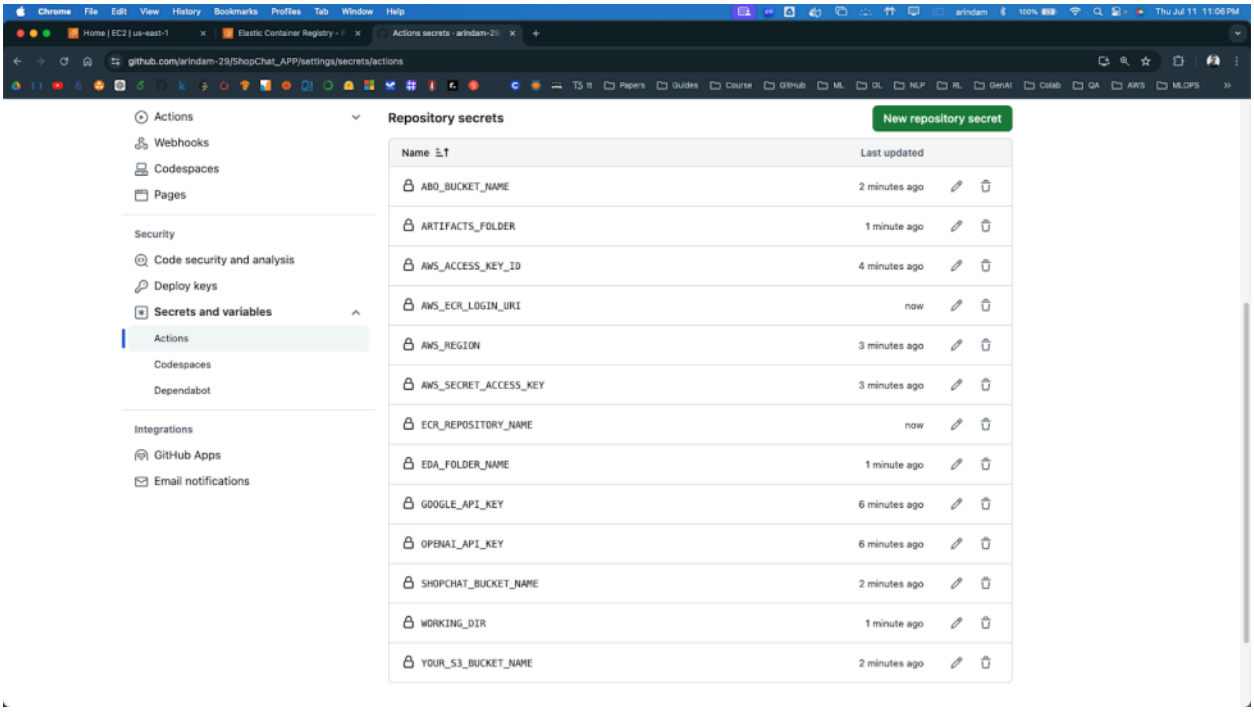
- GOOGLE_API_KEY =
- OPENAI_API_KEY =
- AWS_ACCESS_KEY_ID =
- AWS_SECRET_ACCESS_KEY =
- YOUR_S3_BUCKET_NAME = **shopchat-s3-buckect**
- ECR_REPOSITORY_NAME = **shopchat-repo**
- AWS_ECR_LOGIN_URI = **183651425911.dkr.ecr.us-east-1.amazonaws.com**
- AWS_REGION = **us-east-1**
- SHOPCHAT_BUCKET_NAME = **shopchat-data-bucket**
- ABO_BUCKET_NAME = **amazon-berkeley-objects**
- ARTIFACTS_FOLDER = **../artifacts/**
- WORKING_DIR = **../artifacts/downloads/**

- EDA_FOLDER_NAME = **EDA_FILES**

Note: ECR name and login URI can be found on ECR repository page:

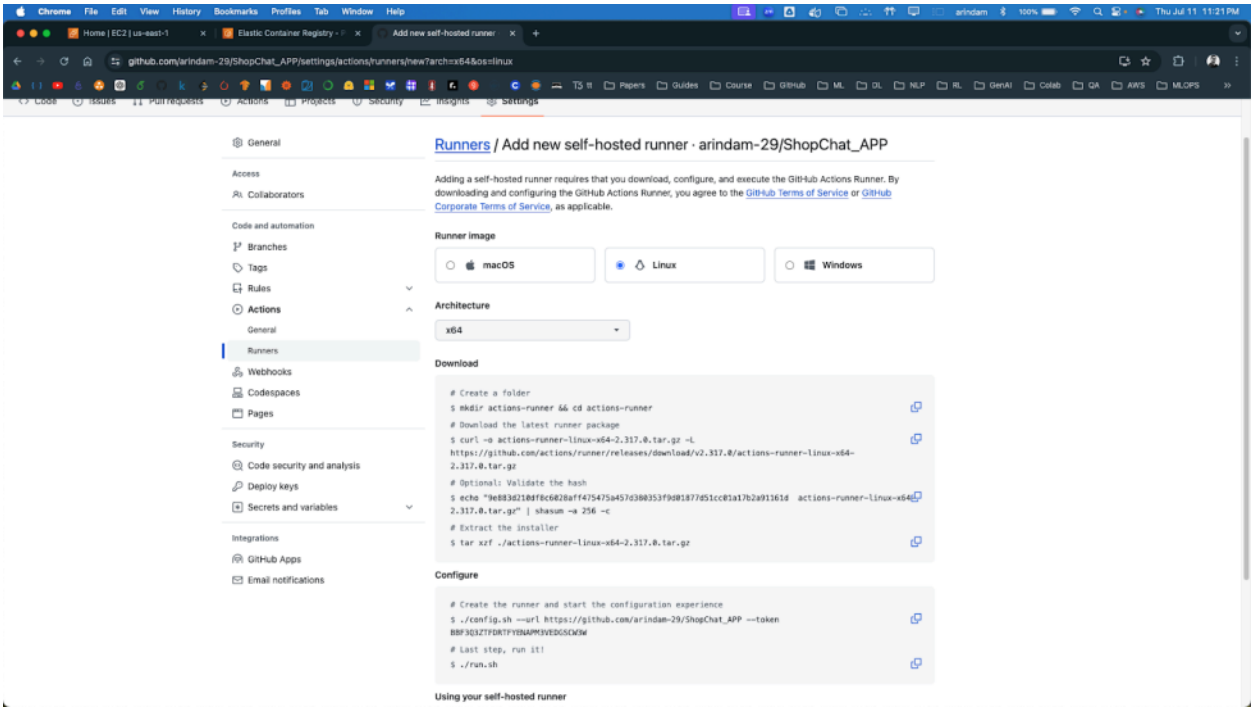


Secrets added



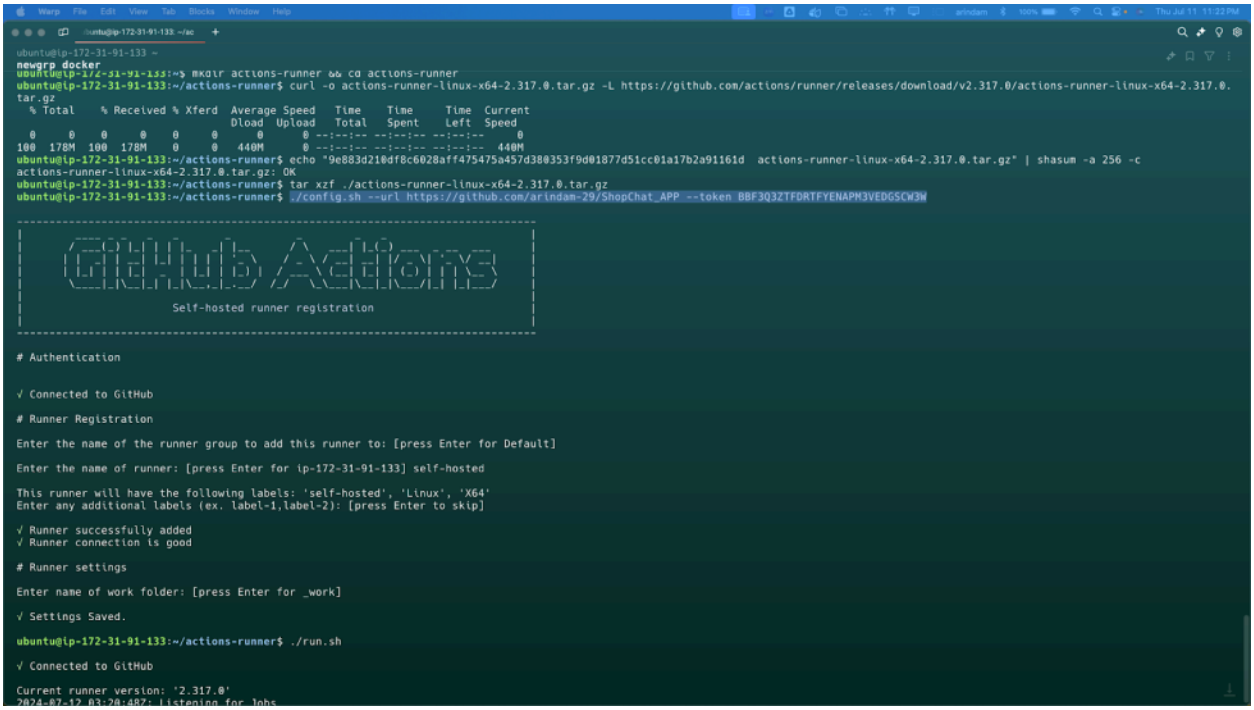
7.7 Add Runner

We will create a self-hosted runner to trigger the deployment pipeline. Got to GitHub repository -> Settings -> Actions -> Runners -> Click on **New self-hosted runner** and select **Linux** as **Runner image**. Copy all the commands and paste them in EC2 ssh client

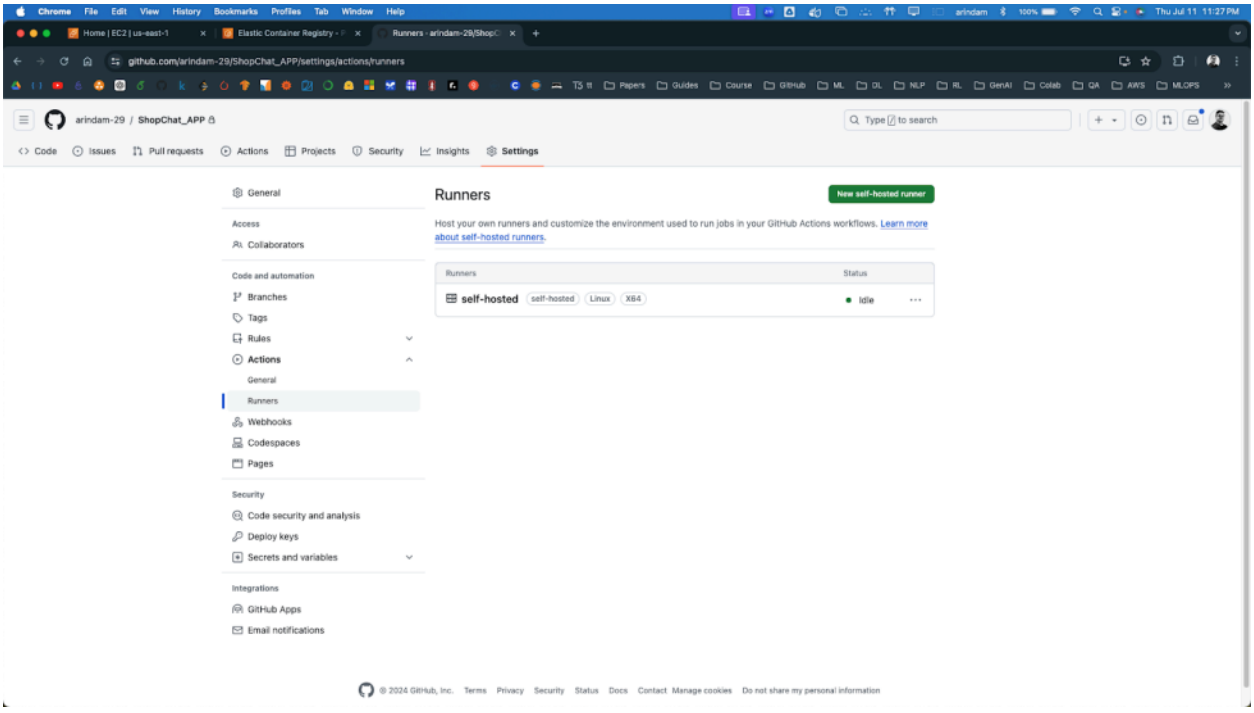


After pasting ./config.sh --url https://github.com/arindam-29/ShopChat_APP --token BBF3Q3ZTFDRTFYENAPM3VEDGSCW3W (first from the configure) use following:

- Enter the name of the runner group to add this runner to: [press Enter for Default] -> Hit Enter for Default
- Enter the name of runner: [press Enter for ip-172-31-91-133] -> Use "**sefl-hosted**"
- This runner will have the following labels: 'self-hosted', 'Linux', 'X64'
- Enter any additional labels (ex. label-1,label-2): [press Enter to skip] -> Hit Enter to skip
- Enter name of work folder: [press Enter for _work] -> Hi Enter

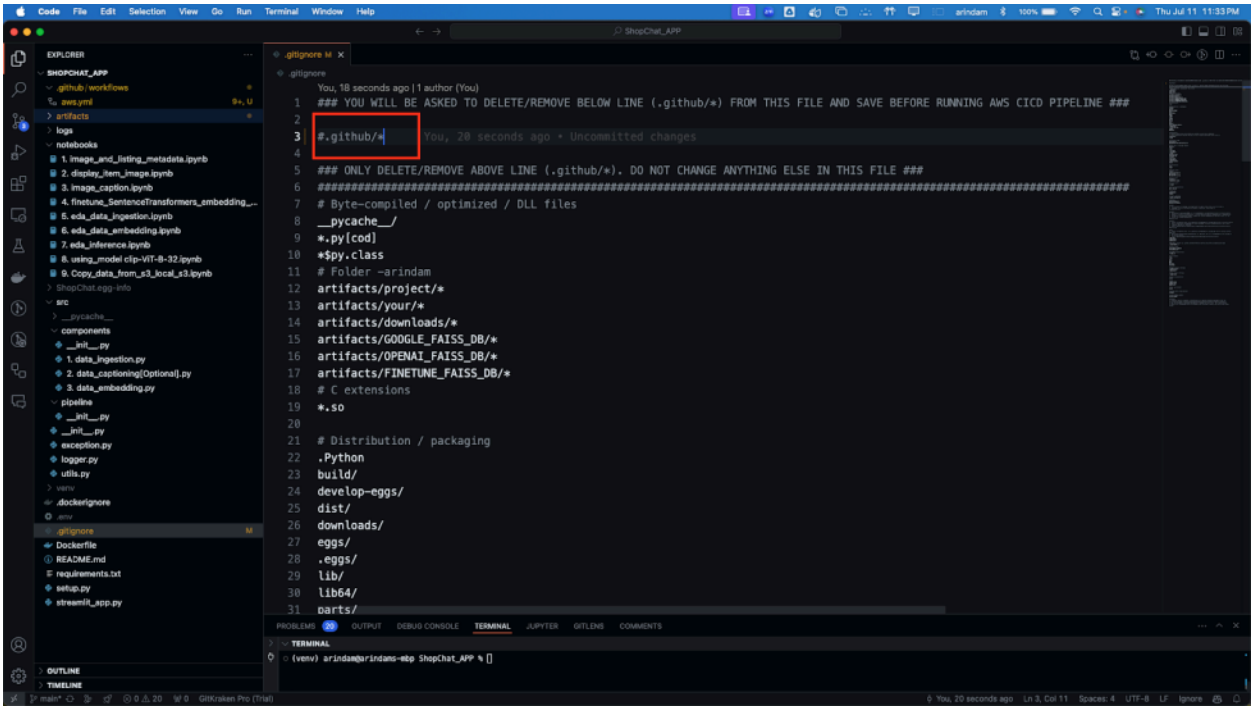


Sefl-hosted runner is now Connected to GitHub and listening for Jobs. You can check the status in GitHub repository -> Settings -> Actions -> Runners. It must be in **Idle** status



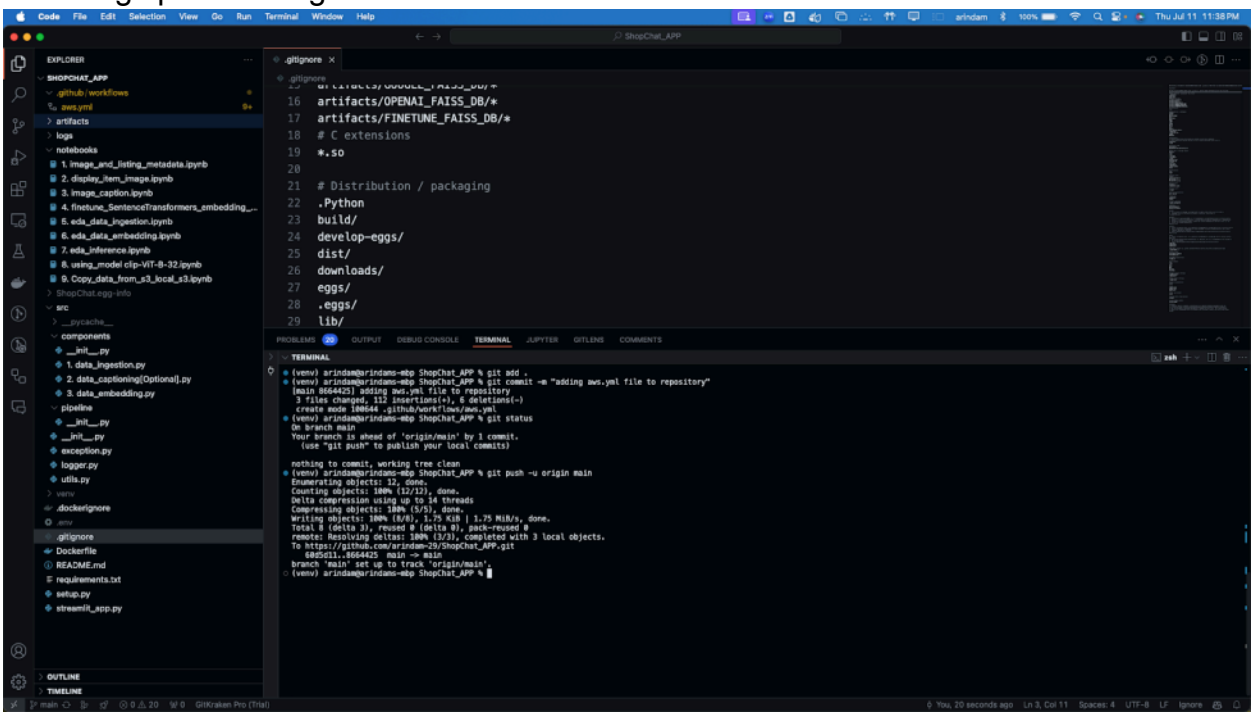
7.8 Add Workflow yml to GitHub repository

Goto the VSCode folder and open **.gitignore** file. Comment out the line **.github/*** and save it.



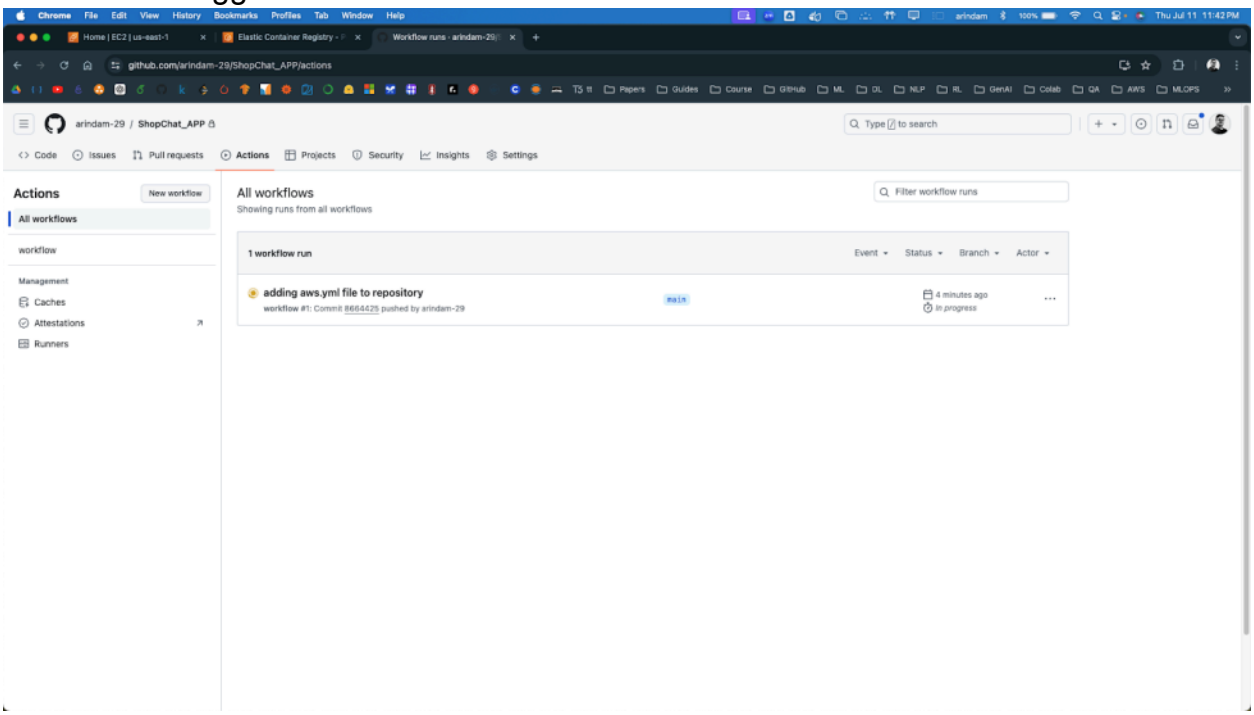
In the terminal, issue to follow git commands to add the change in GitHub repository

- git add .
- git commit -m “adding aws.yml file to repository”
- git push -u origin main

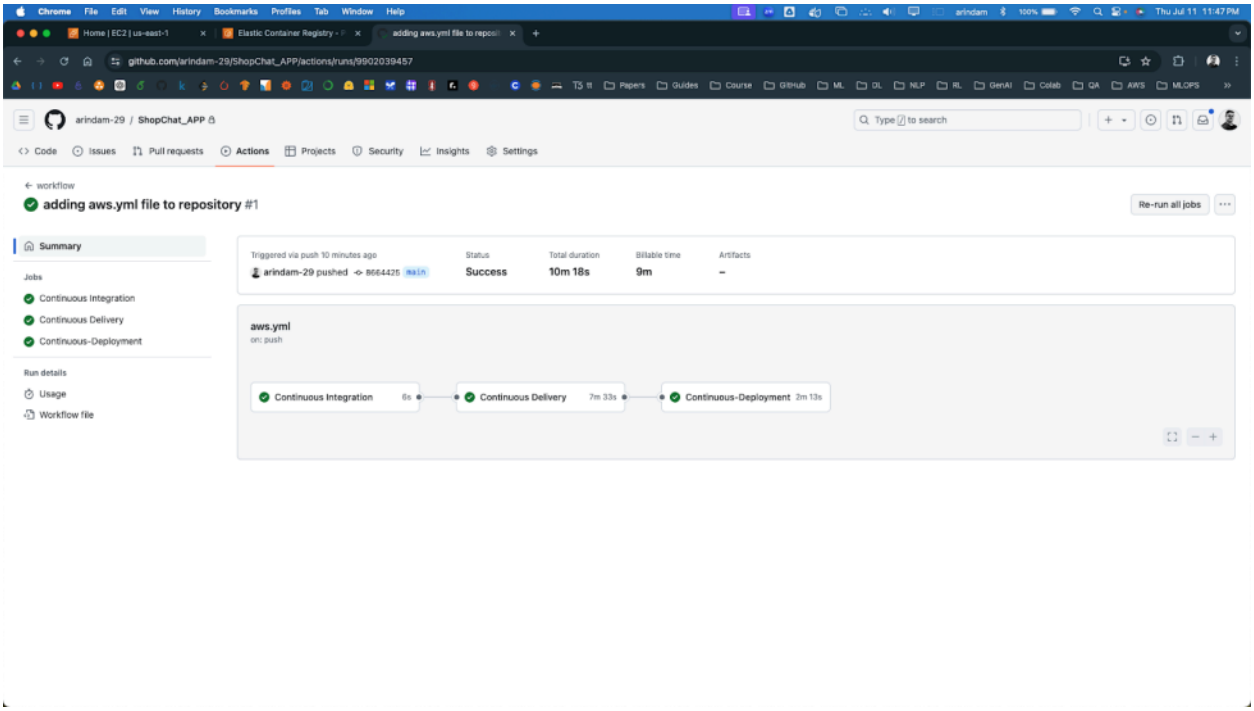


7.9 CICD Pipeline Triggers

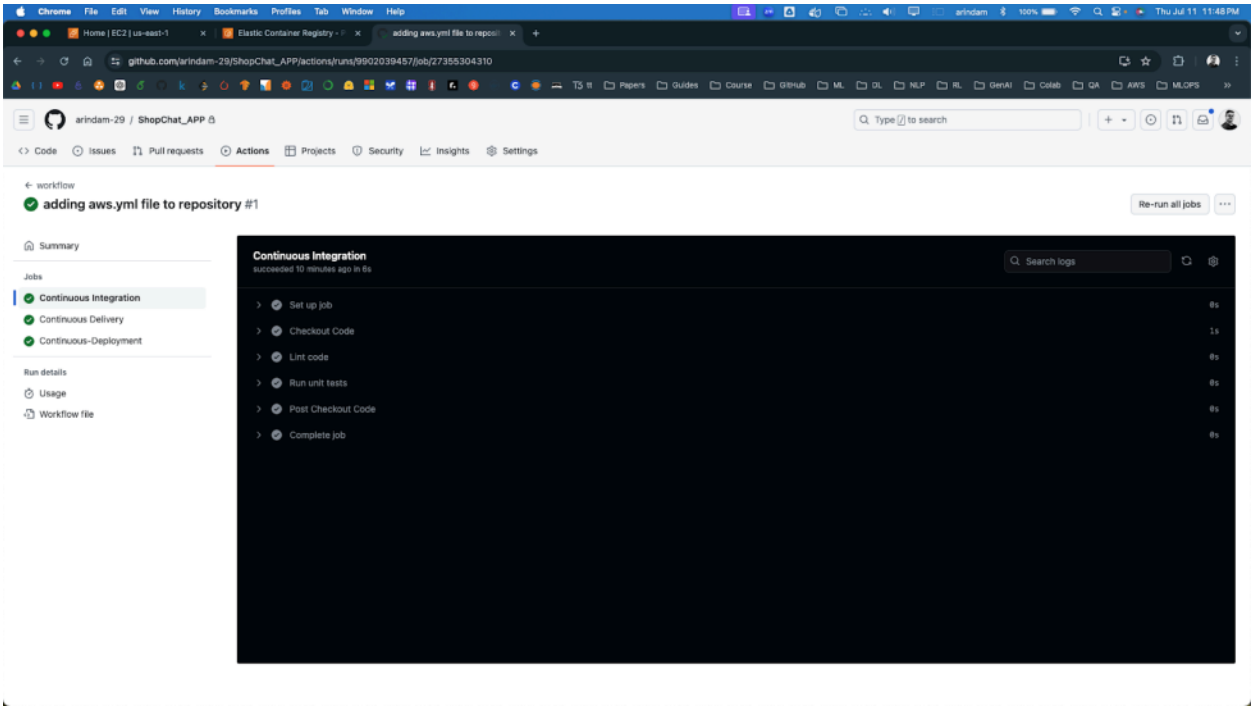
Once the aws.yml file is pushed; Go to GitHub repository -> Actions. You can see the workflow triggers.

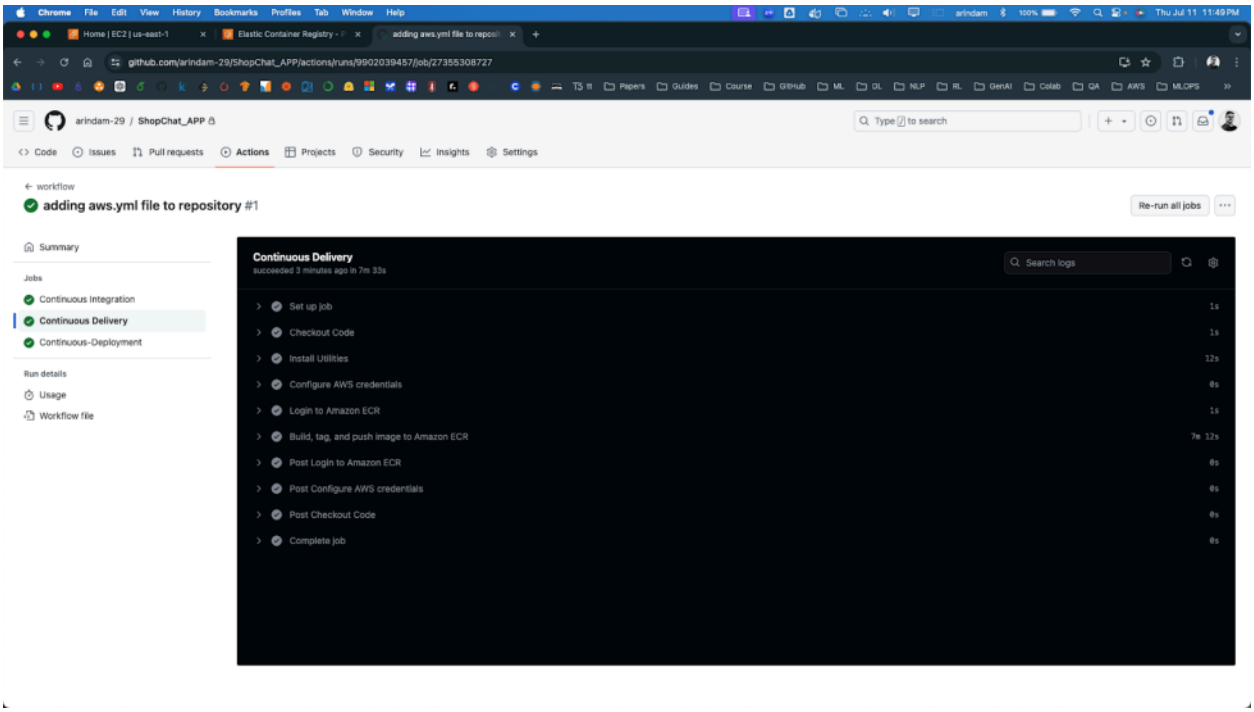


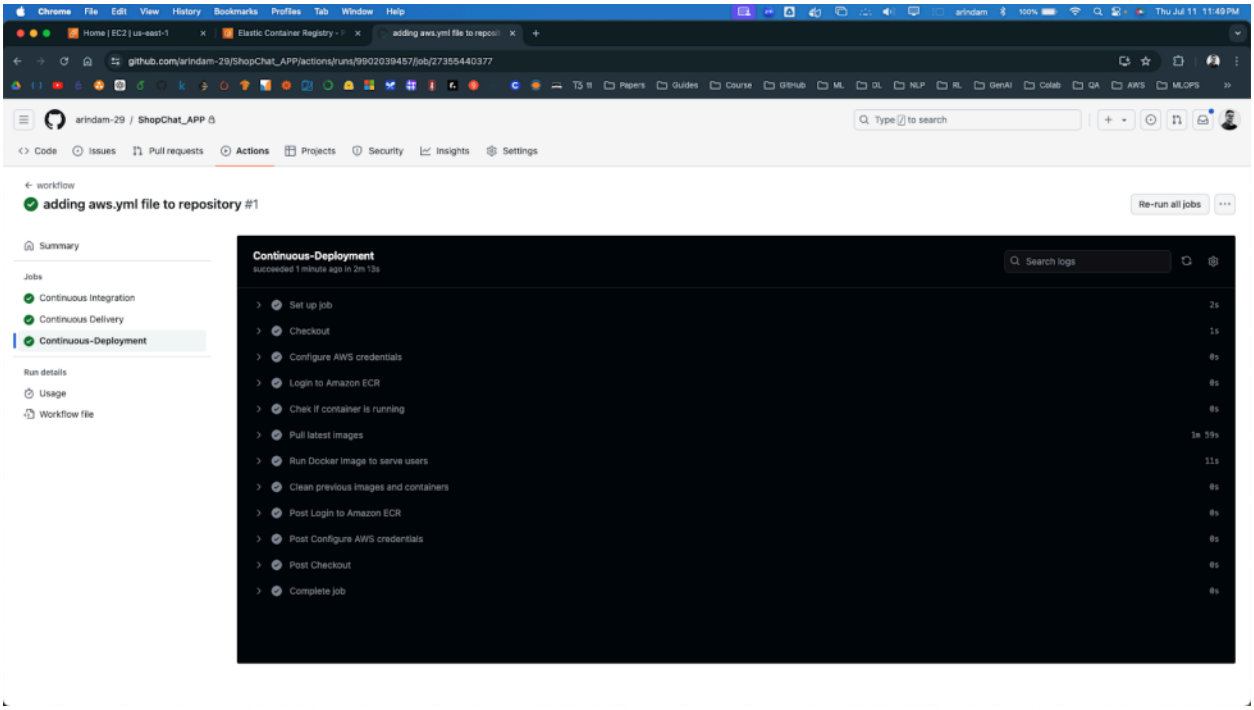
Click on the job. You can see the details of the pipeline



Also see the steps in each pipeline


github.com/arindam-29/ShopChat_APP/actions/runs/9902039457/job/27355304310
arindam-29 / ShopChat_APP
Code
Issues
Pull requests
Actions
Projects
Security
Insights
Settings
workflow
adding aws.yml file to repository #1
Re-run all jobs
Summary
Jobs
Continuous Integration
Continuous Delivery
Continuous-Deployment
Run details
Usage
Workflow file
Continuous Integration
succeeded 10 minutes ago in 6s
Search logs
Set up job
Checkout Code
Lint code
Run unit tests
Post Checkout Code
Complete job

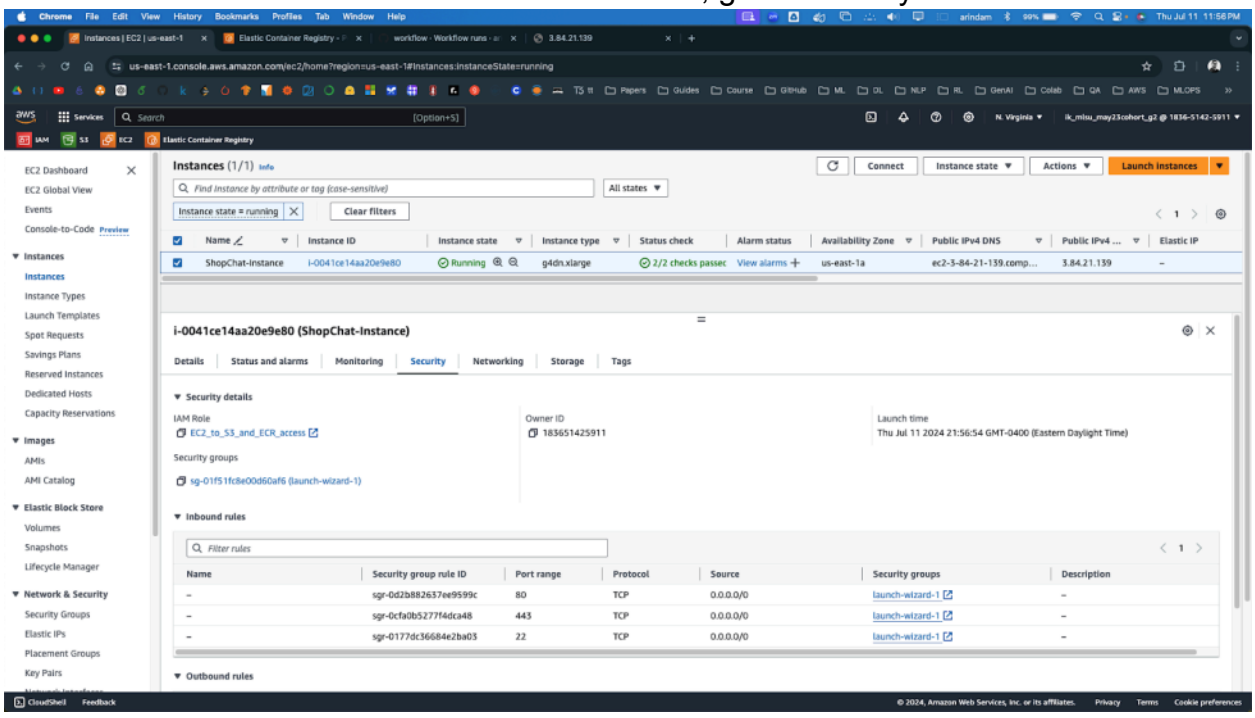

github.com/arindam-29/ShopChat_APP/actions/runs/9902039457/job/27355308727
arindam-29 / ShopChat_APP
Code
Issues
Pull requests
Actions
Projects
Security
Insights
Settings
workflow
adding aws.yml file to repository #1
Re-run all jobs
Summary
Jobs
Continuous Integration
Continuous Delivery
Continuous-Deployment
Run details
Usage
Workflow file
Continuous Delivery
succeeded 3 minutes ago in 7m 33s
Search logs
Set up job
Checkout Code
Install Utilities
Configure AWS credentials
Login to Amazon ECR
Build, tag, and push image to Amazon ECR
Post Login to Amazon ECR
Post Configure AWS credentials
Post Checkout Code
Complete job



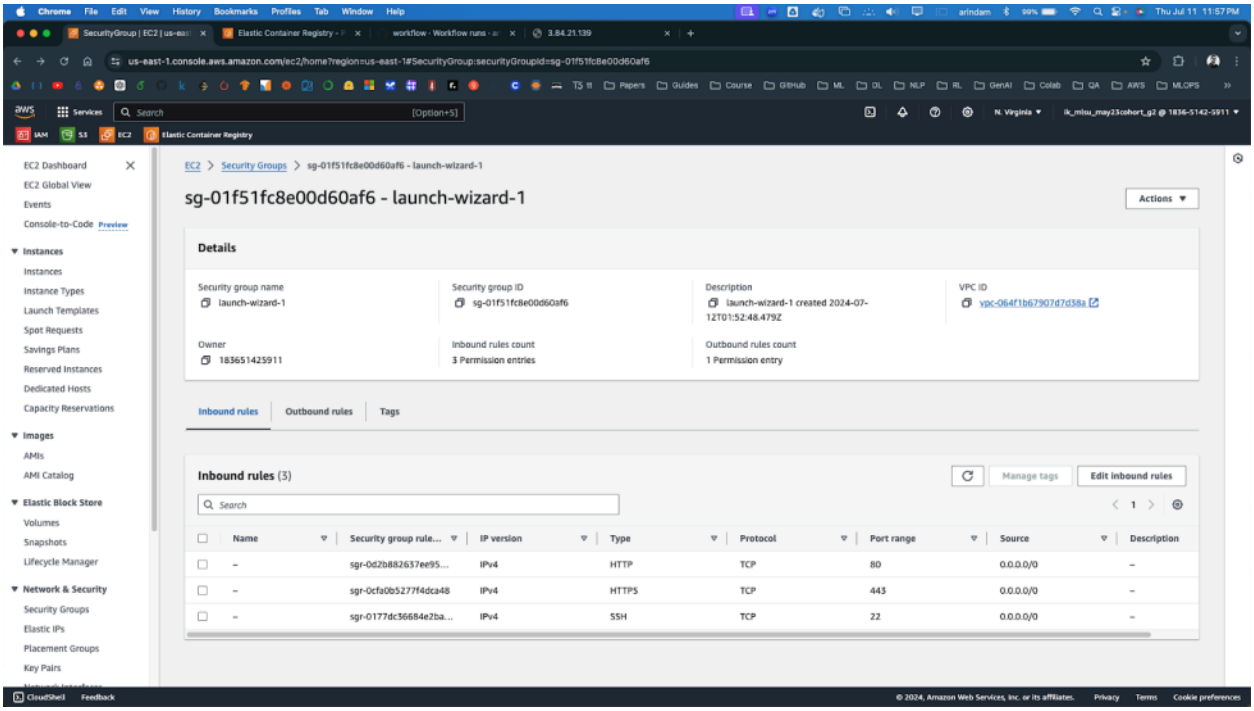
Our docker image is now running EC2 instance

7.10 Open port 8501 in EC2

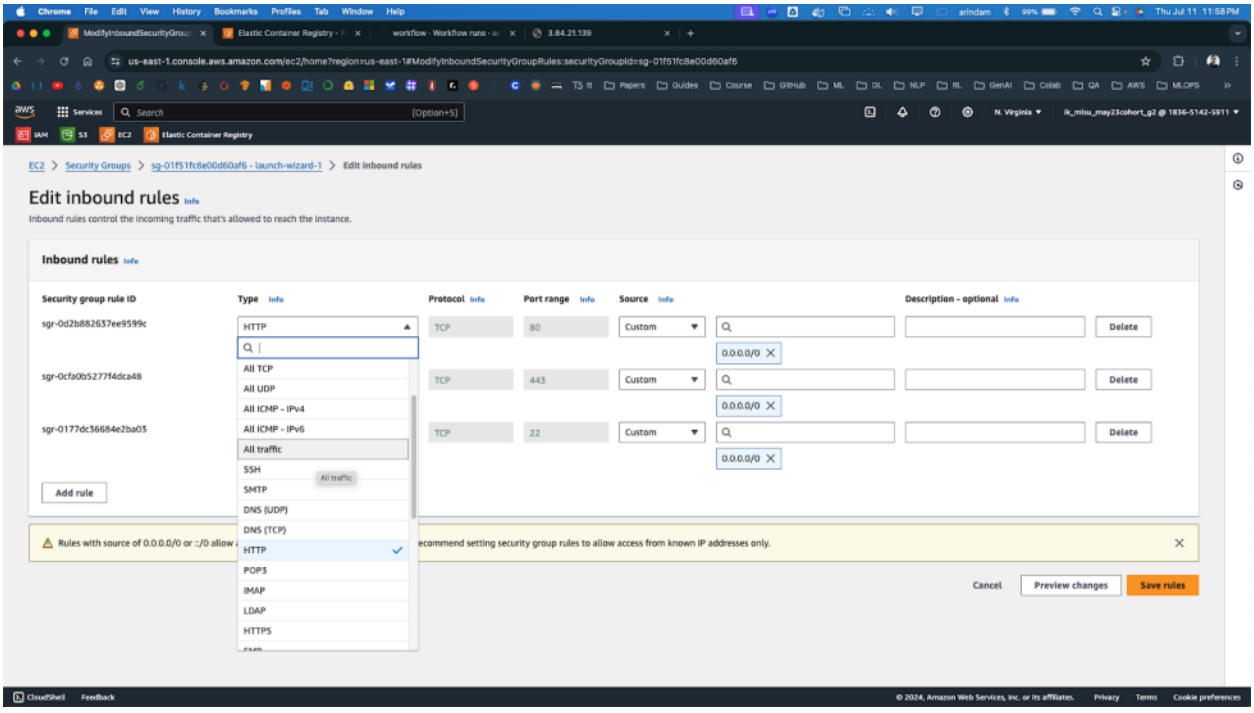
Go to the AWS EC2 instance. Select our instance, go to Security

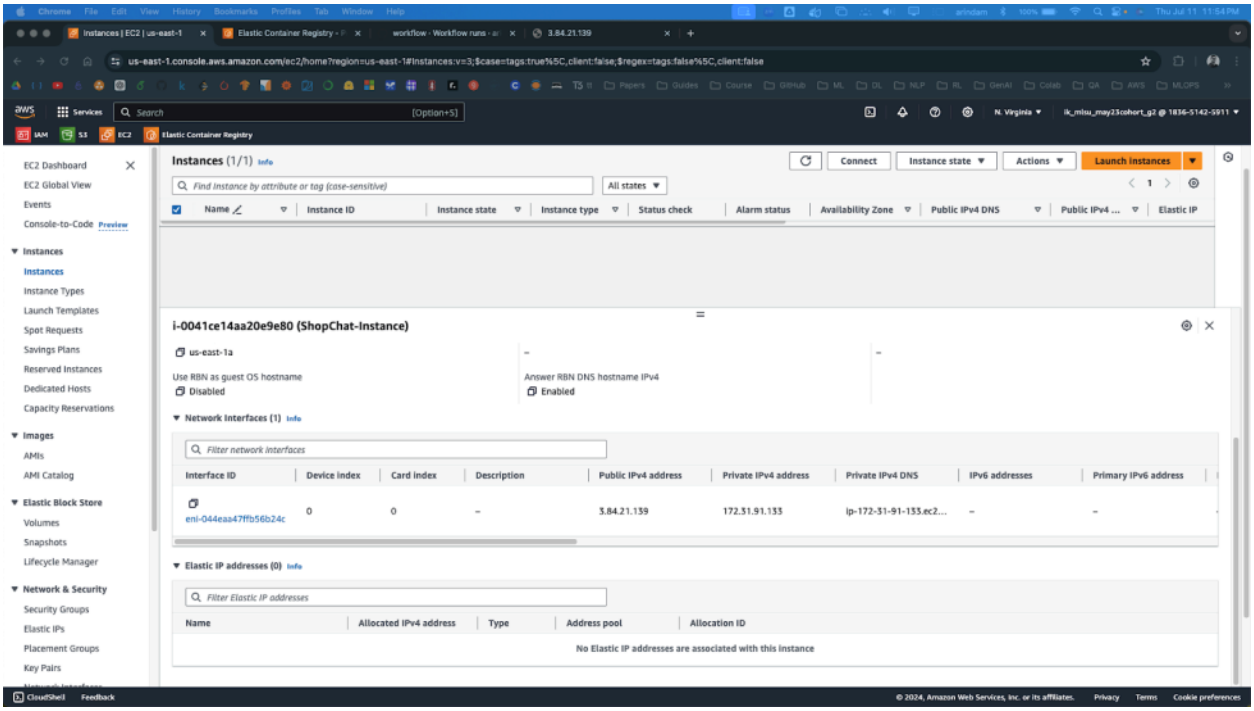


Click on Security Group -> click on Edit Inbound rules



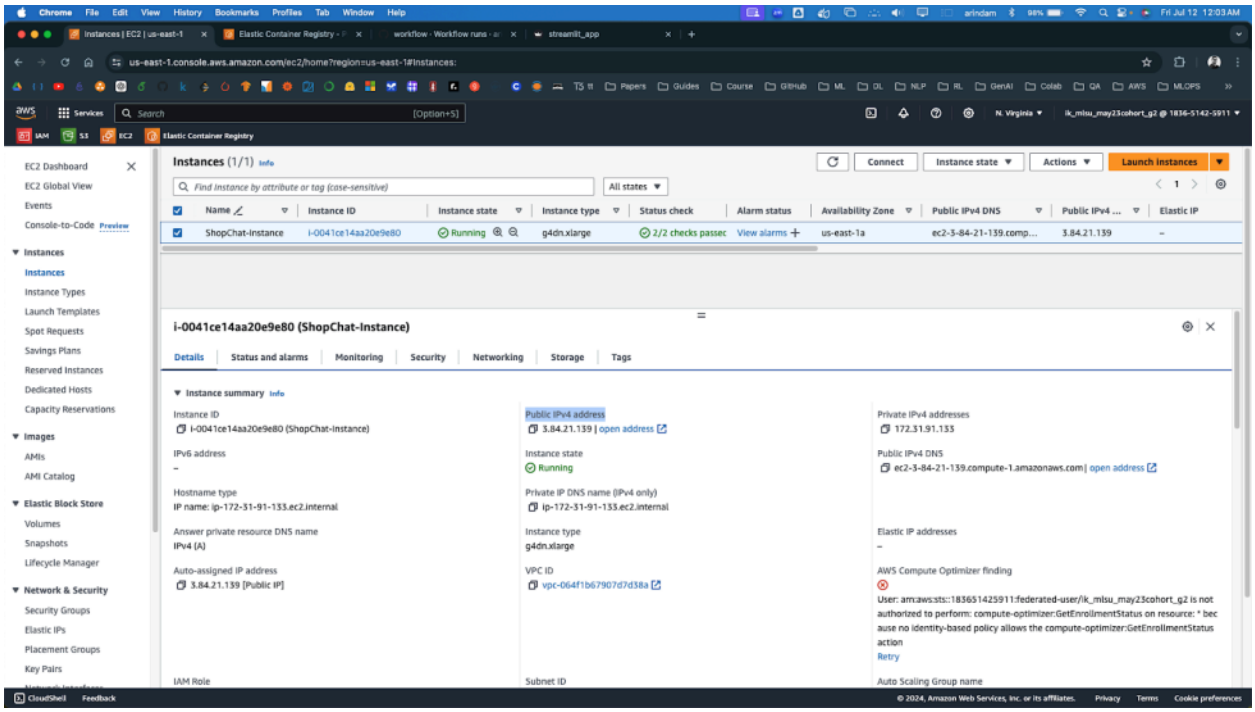
For the first security group rule ID, update the Type to "**All traffic**" and click on Save rules. *Note: lets open all ports here and not only 8501*



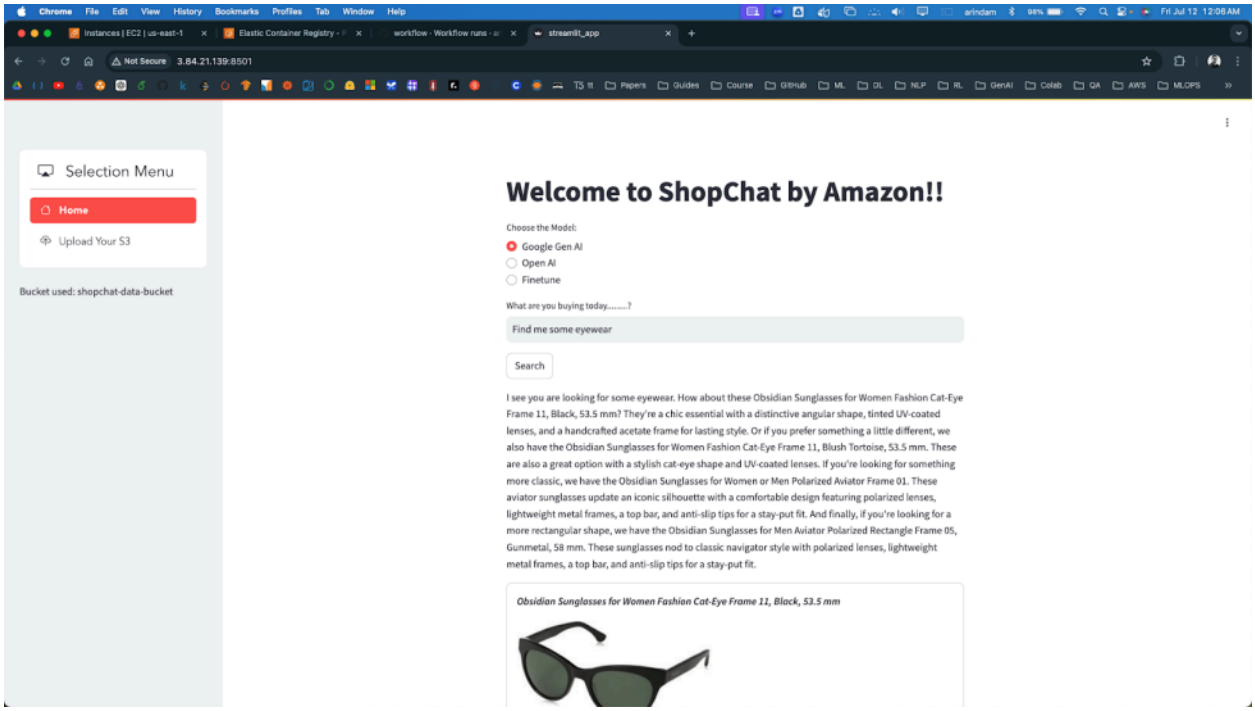


7.11 ShopChat App inference from AWS using public URL

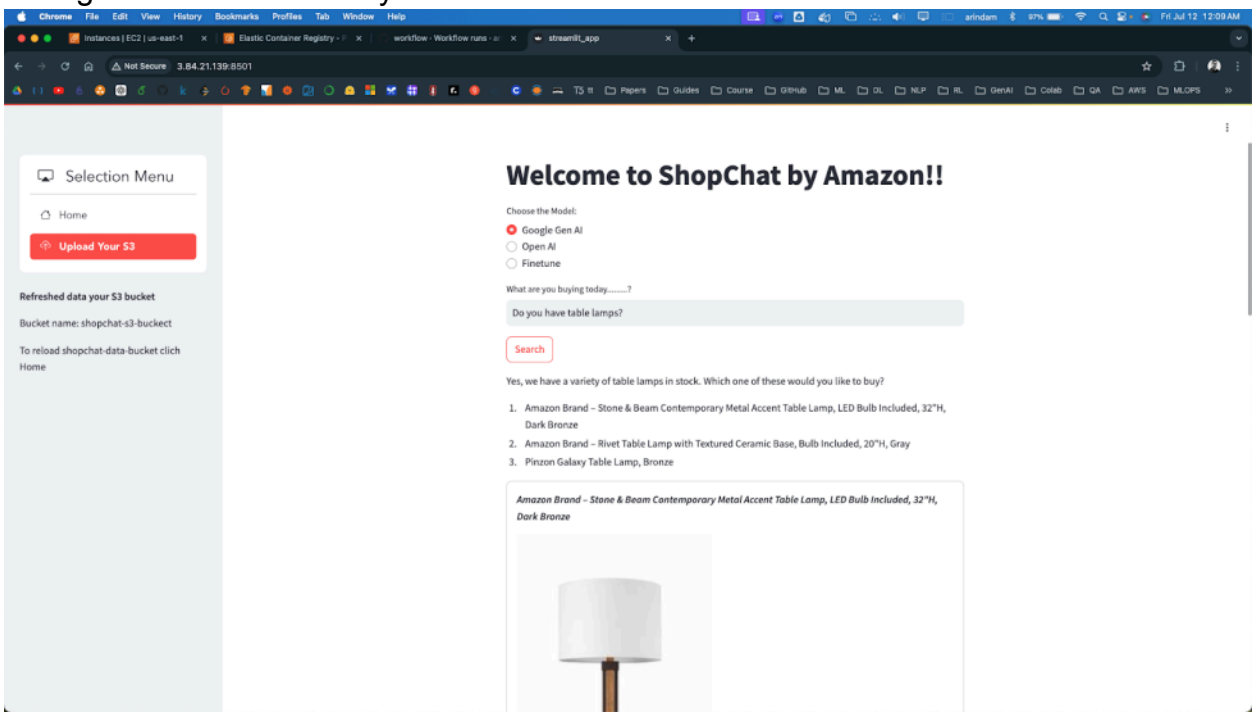
Go to the AWS EC2 instance. Select Public IPv4 address



Goto the Public URL and start interacting with ShopChat App ☺
URL for this instance with port number-> **3.84.21.139**:8501

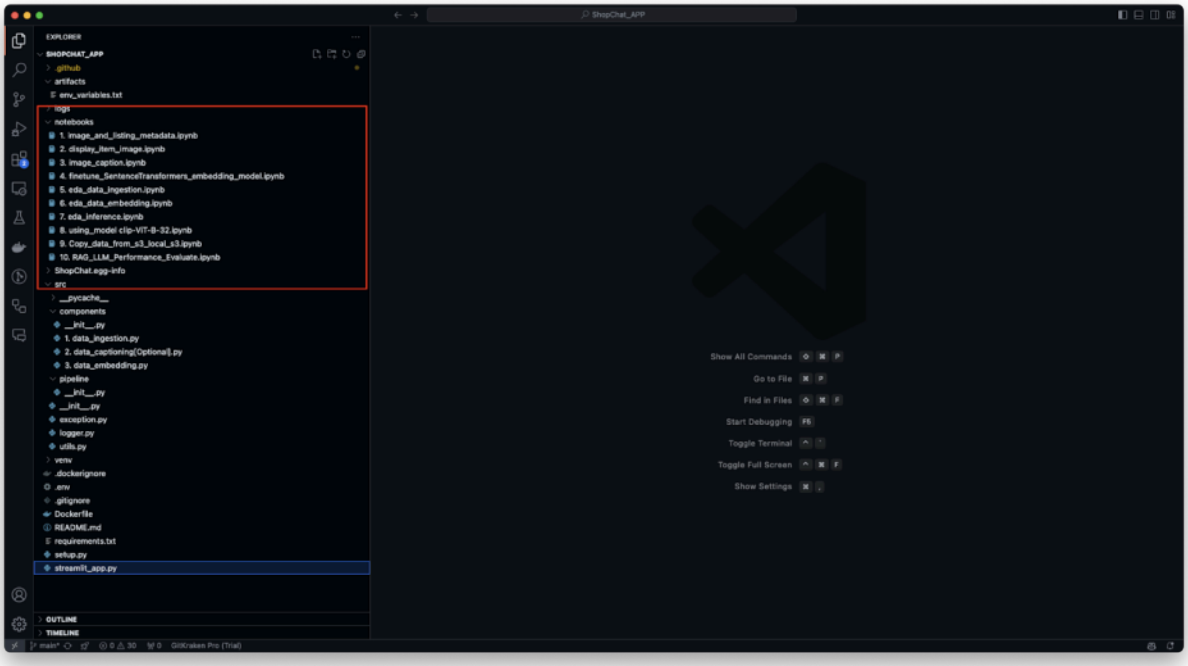


Changed the S3 bucket to your S3 bucket

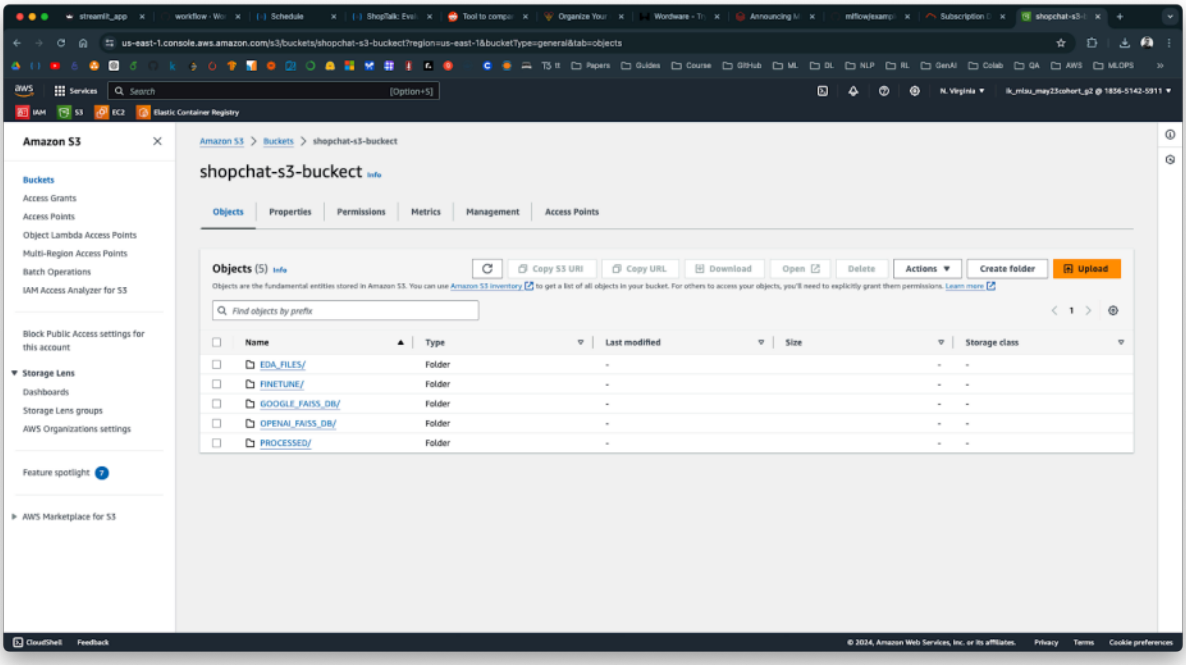


8 EDA and Data Preparation

All EDA notebooks can be found under Notebook folder. The detailed data preparation, cleaning transformation are described in each notebooks



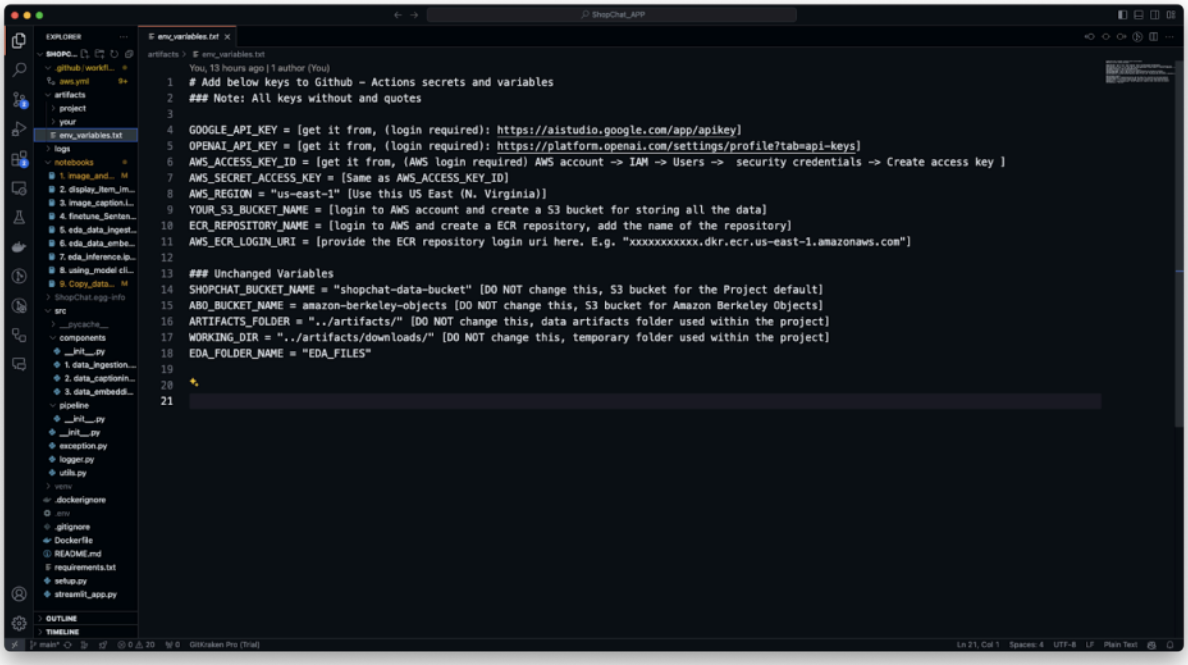
On running the Notebook files EDA and Finetune folders are created in your S3 bucket



9 Supporting scripts and files

9.1 env_variables.txt

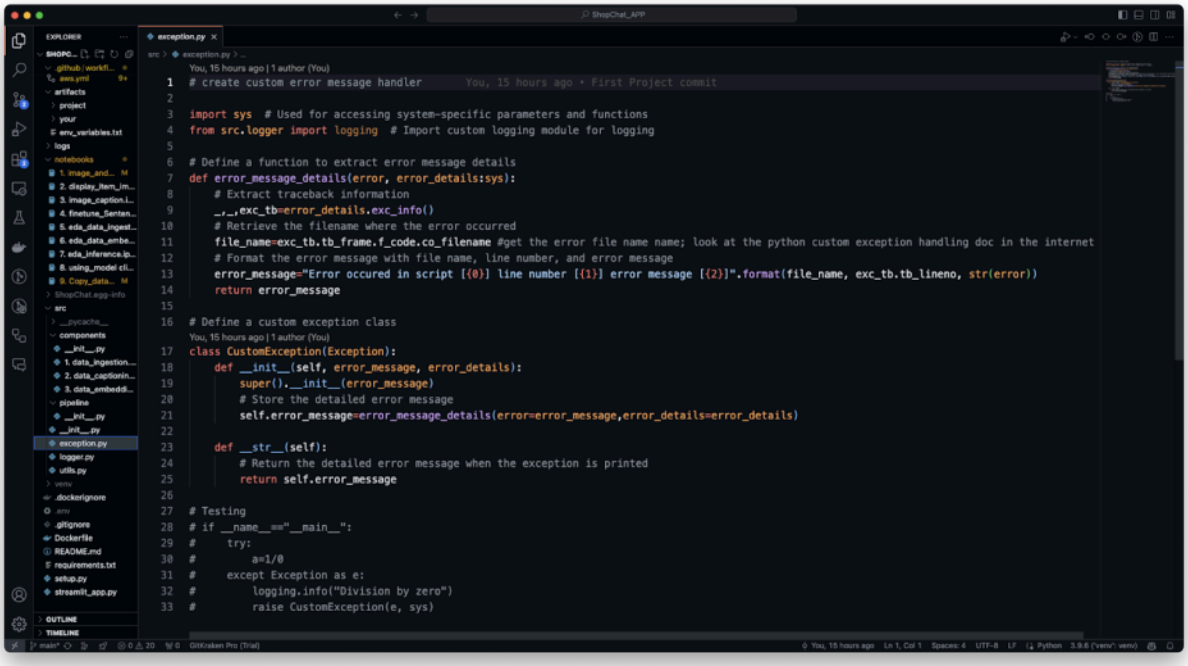
This document under artifacts gives the detailed secret keys to be added in GitHub action



```
# Add below keys to Github - Actions secrets and variables
### Note: All keys without and quotes

GOOGLE_API_KEY = [get it from, (login required): https://aistudio.google.com/app/apikey]
OPENAI_API_KEY = [get it from, (login required): https://platform.openai.com/settings/profile?tab=api-keys]
AWS_ACCESS_KEY_ID = [get it from, (AWS login required) AWS account -> IAM -> Users ->  security credentials -> Create access key ]
AWS_SECRET_ACCESS_KEY = [Same as AWS_ACCESS_KEY_ID]
AWS_REGION = "us-east-1" [Use this US East (N. Virginia)]
YOUR_S3_BUCKET_NAME = [login to AWS account and create a S3 bucket for storing all the data]
ECR_REPOSITORY_NAME = [login to AWS and create a ECR repository, add the name of the repository]
AWS_ECR_LOGIN_URI = [provide the ECR repository login uri here. E.g. "xxxxxxxxxxx.dkr.ecr.us-east-1.amazonaws.com"]

### Unchanged Variables
SHOPCHAT_BUCKET_NAME = "shopchat-data-bucket" [DO NOT change this, S3 bucket for the Project default]
ABO_BUCKET_NAME = amazon-berkeley-objects [DO NOT change this, S3 bucket for Amazon Berkeley Objects]
ARTIFACTS_FOLDER = "../artifacts/" [DO NOT change this, data artifacts folder used within the project]
WORKING_DIR = "../artifacts/downloads/" [DO NOT change this, temporary folder used within the project]
EDA_FOLDER_NAME = "EDA_FILES"
```

9.2 exception.py

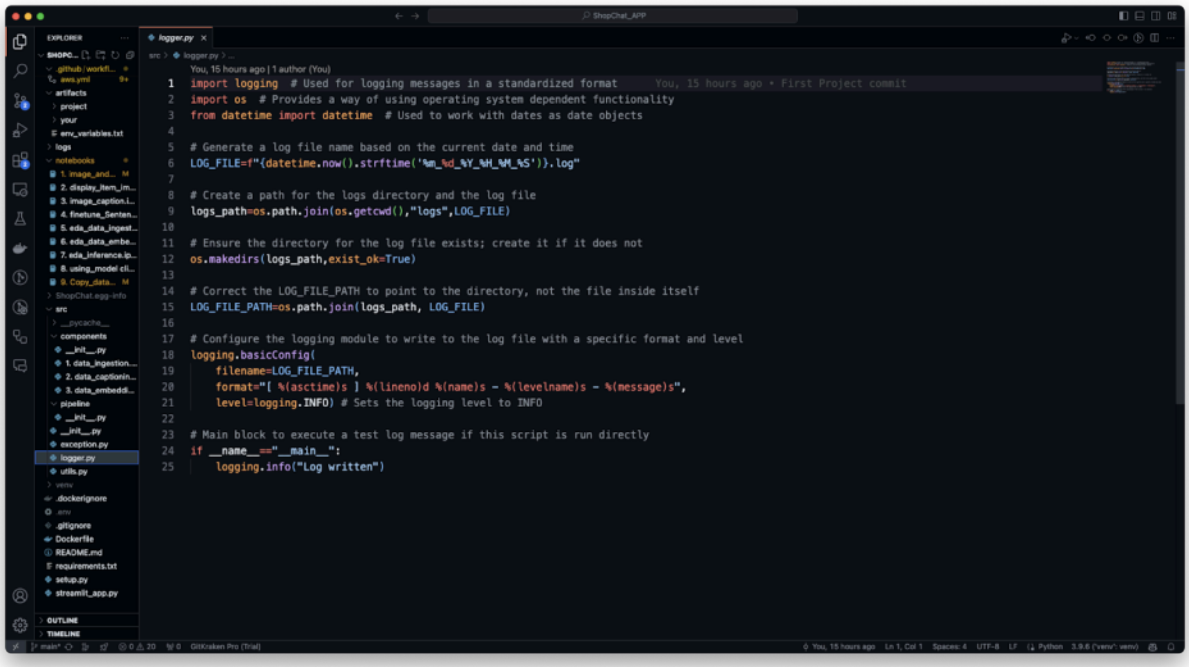
This custom exception scripts raises exception if occurs during the running of any scrips



```
# create custom error message handler

import sys  # Used for accessing system-specific parameters and functions
from src.logger import logging  # Import custom logging module for logging

# Define a function to extract error message details
def error_message_details(error, error_details:sys):
    # Extract traceback information
    _,_,exc_tb=error_details.exc_info()
    # Retrieve the filename where the error occurred
    file_name=exc_tb.tb_frame.f_code.co_filename #get the error file name name; look at the python custom exception handling doc in the internet
    # Format the error message with file name, line number, and error message
    error_message="Error occured in script [{0}] line number [{1}] error message [{2}]".format(file_name, exc_tb.tb_lineno, str(error))
    return error_message

# Define a custom exception class
class CustomException(Exception):
    def __init__(self, error_message, error_details):
        super().__init__(error_message)
        # Store the detailed error message
        self.error_message=error_message_details(error=error_message,error_details=error_details)

    def __str__(self):
        # Return the detailed error message when the exception is printed
        return self.error_message

# Testing
# if __name__=="__main__":
#     try:
#         a=1/0
#     except Exception as e:
#         logging.info("Division by zero")
#         raise CustomException(e, sys)
```

9.3 logging.py

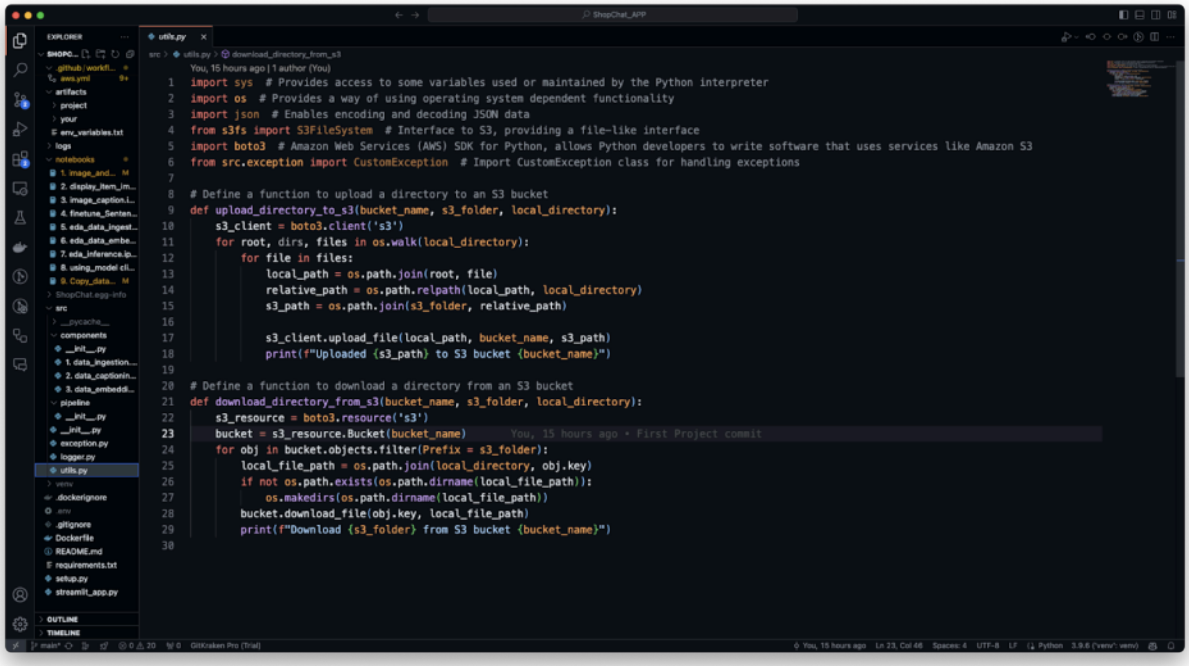
This is to log the info in log files



```python
import logging  # Used for logging messages in a standardized format
import os  # Provides a way of using operating system dependent functionality
from datetime import datetime  # Used to work with dates as date objects

# Generate a log file name based on the current date and time
LOG_FILE=f"{datetime.now().strftime('%m_%d_%Y_%H_%M_%S')}.log"

# Create a path for the logs directory and the log file
logs_path=os.path.join(os.getcwd(),"logs",LOG_FILE)

# Ensure the directory for the log file exists; create it if it does not
os.makedirs(logs_path,exist_ok=True)

# Correct the LOG_FILE_PATH to point to the directory, not the file inside itself
LOG_FILE_PATH=os.path.join(logs_path, LOG_FILE)

# Configure the logging module to write to the log file with a specific format and level
logging.basicConfig(
    filename=LOG_FILE_PATH,
    format="[ %(asctime)s ] %(lineno)d %(name)s - %(levelname)s - %(message)s",
    level=logging.INFO) # Sets the logging level to INFO

# Main block to execute a test log message if this script is run directly
if __name__=="__main__":
    logging.info("Log written")
```

9.4 utils.py

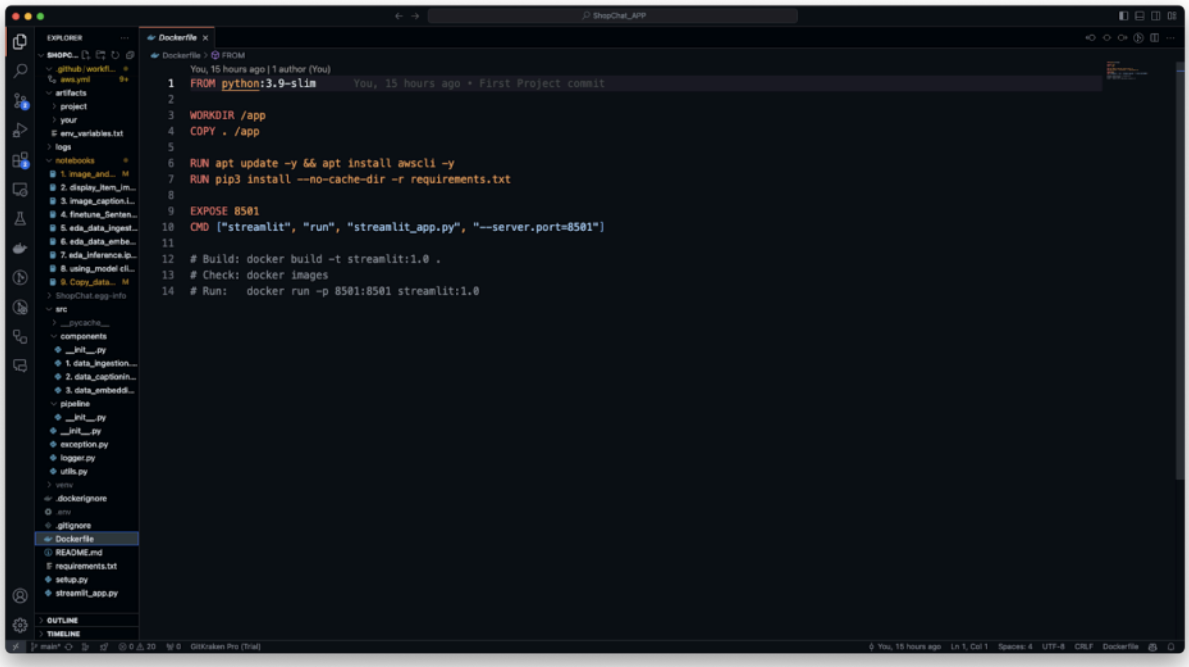
Utility script. There are two commonly used functions defined here. Loading and downloading folders from S3



```python
import sys  # Provides access to some variables used or maintained by the Python interpreter
import os  # Provides a way of using operating system dependent functionality
import json  # Enables encoding and decoding JSON data
from s3fs import S3FileSystem  # Interface to S3, providing a file-like interface
import boto3  # Amazon Web Services (AWS) SDK for Python, allows Python developers to write software that uses services like Amazon S3
from src.exception import CustomException  # Import CustomException class for handling exceptions

# Define a function to upload a directory to an S3 bucket
def upload_directory_to_s3(bucket_name, s3_folder, local_directory):
    s3_client = boto3.client('s3')
    for root, dirs, files in os.walk(local_directory):
        for file in files:
            local_path = os.path.join(root, file)
            relative_path = os.path.relpath(local_path, local_directory)
            s3_path = os.path.join(s3_folder, relative_path)

            s3_client.upload_file(local_path, bucket_name, s3_path)
            print(f"Uploaded {s3_path} to S3 bucket {bucket_name}")

# Define a function to download a directory from an S3 bucket
def download_directory_from_s3(bucket_name, s3_folder, local_directory):
    s3_resource = boto3.resource('s3')
    bucket = s3_resource.Bucket(bucket_name)
    for obj in bucket.objects.filter(Prefix = s3_folder):
        local_file_path = os.path.join(local_directory, obj.key)
        if not os.path.exists(os.path.dirname(local_file_path)):
            os.makedirs(os.path.dirname(local_file_path))
        bucket.download_file(obj.key, local_file_path)
        print(f"Download {s3_folder} from S3 bucket {bucket_name}")
```

9.5 Dockerfile

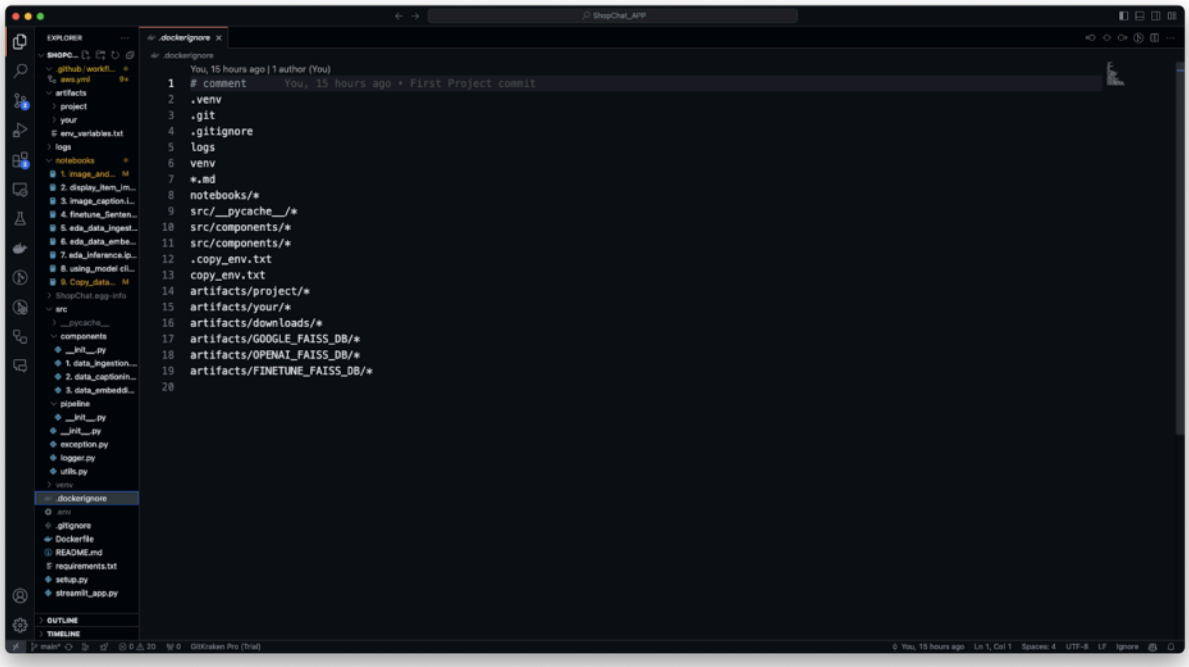
Scripts to containerize the application to create and run docker image



```
FROM python:3.9-slim

WORKDIR /app
COPY . /app

RUN apt update -y && apt install awscli -y
RUN pip3 install --no-cache-dir -r requirements.txt

EXPOSE 8501
CMD ["streamlit", "run", "streamlit_app.py", "--server.port=8501"]

# Build: docker build -t streamlit:1.0 .
# Check: docker images
# Run:   docker run -p 8501:8501 streamlit:1.0
```

9.6 .dockerignore

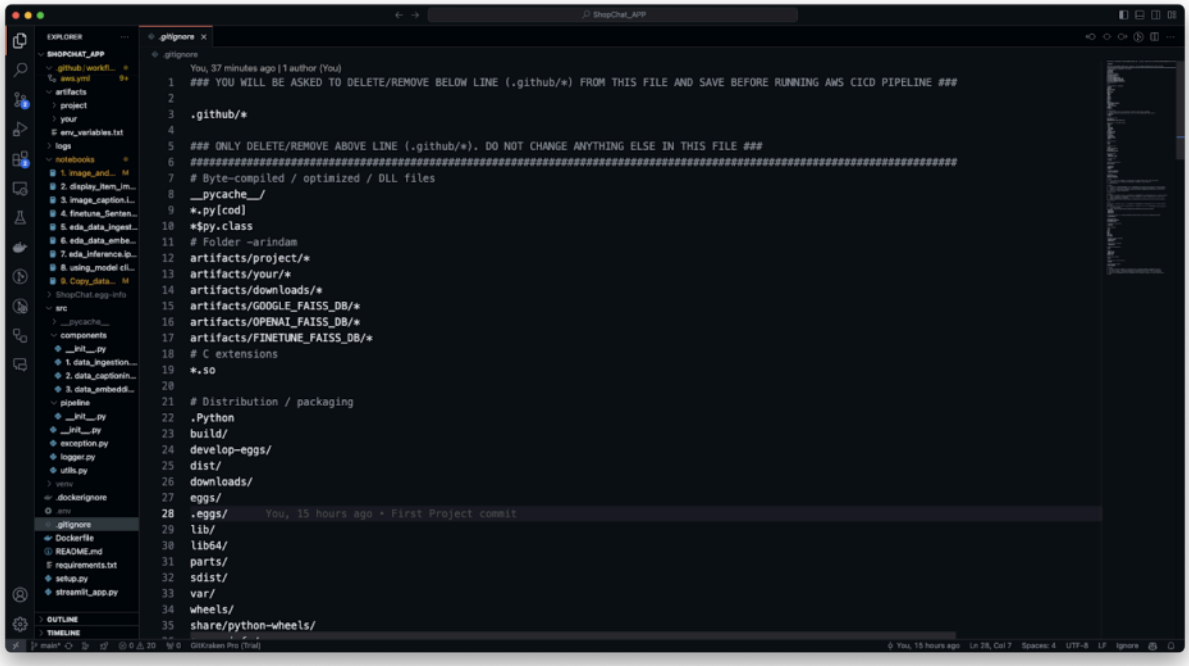
Contains the list of the files and folders to be ignored while building docker image



```
# comment
.venv
.git
.gitignore
logs
venv
*.md
notebooks/*
src/__pycache__/*
src/components/*
src/components/*
.copy_env.txt
copy_env.txt
artifacts/project/*
artifacts/your/*
artifacts/downloads/*
artifacts/GOOGLE_FAISS_DB/*
artifacts/OPENAI_FAISS_DB/*
artifacts/FINETUNE_FAISS_DB/*
```

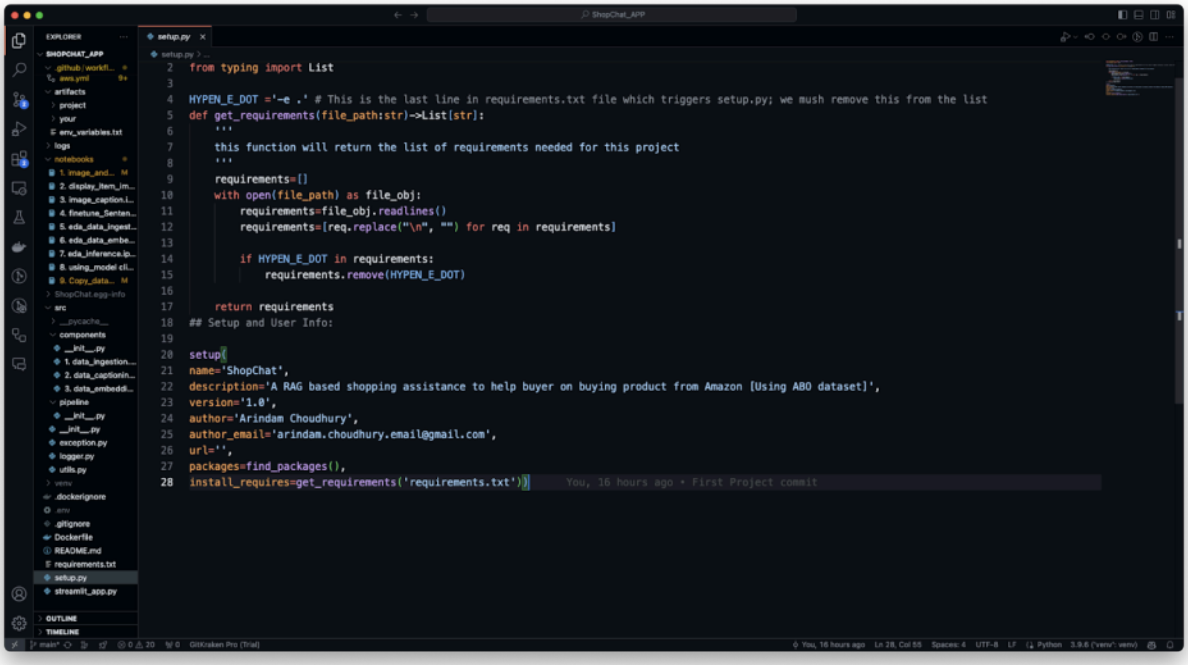
9.7 .gitignore

Contains the list of the files and folders to be ignored in GitHub repository



9.8 setup.py

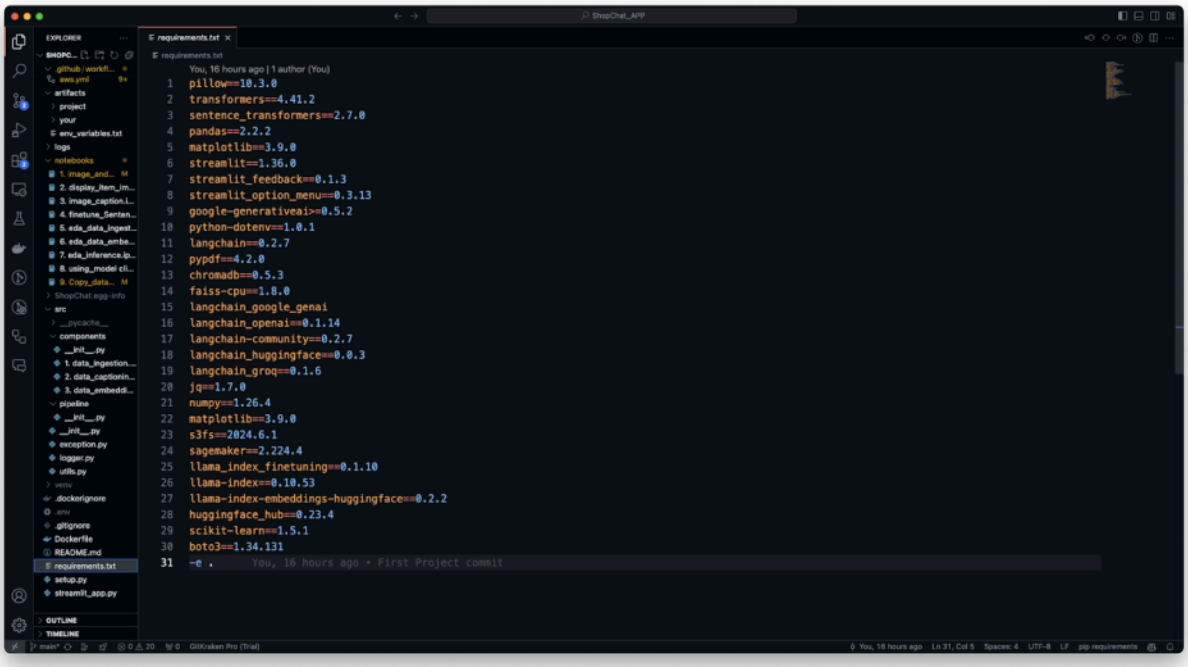
This Python script is designed for setting up a Python package using setuptools. It includes a function to read and process a list of requirements from a requirements.txt file, and it uses this list in the package setup configuration



```python
from typing import List

HYPEN_E_DOT ='-e .' # This is the last line in requirements.txt file which triggers setup.py; we mush remove this from the list
def get_requirements(file_path:str)->List[str]:
    '''
    this function will return the list of requirements needed for this project
    '''
    requirements=[]
    with open(file_path) as file_obj:
        requirements=file_obj.readlines()
        requirements=[req.replace("\n", "") for req in requirements]

        if HYPEN_E_DOT in requirements:
            requirements.remove(HYPEN_E_DOT)

    return requirements
## Setup and User Info:

setup(
name='ShopChat',
description='A RAG based shopping assistance to help buyer on buying product from Amazon [Using ABO dataset]',
version='1.0',
author='Arindam Choudhury',
author_email='arindam.choudhury.email@gmail.com',
url='',
packages=find_packages(),
install_requires=get_requirements('requirements.txt'))
```

9.9 requirements.txt

List of Python packages and dependencies to run the project



```
pillow==10.3.0
transformers==4.41.2
sentence_transformers==2.7.0
pandas==2.2.2
matplotlib==3.9.0
streamlit==1.36.0
streamlit_feedback==0.1.3
streamlit_option_menu==0.3.13
google-generativeai>=0.5.2
python-dotenv==1.0.1
langchain==0.2.7
pypdf==4.2.0
chromadb==0.5.3
faiss-cpu==1.8.0
langchain_google_genai
langchain_openai==0.1.14
langchain-community==0.2.7
langchain_huggingface==0.0.3
langchain_groq==0.1.6
jq==1.7.0
numpy==1.26.4
matplotlib==3.9.0
s3fs==2024.6.1
sagemaker==2.224.4
llama_index_finetuning==0.1.10
llama-index==0.10.53
llama-index-embeddings-huggingface==0.2.2
huggingface_hub==0.23.4
scikit-learn==1.5.1
boto3==1.34.131
-e .
```

10 Observations and Issues

There are many rooms for improvements here, given the time permits. We could have added other feature such as 3D, 360-degree view, geometry, material etc. but here we have taken only the images to create this app.
Better finetuned model, hyper parameter tuning, embedding finetuning etc. also could have been implemented if would have gotten more time.
Throughput could have been reduced with proper design of UI with better logical flow and data handling mechanism.

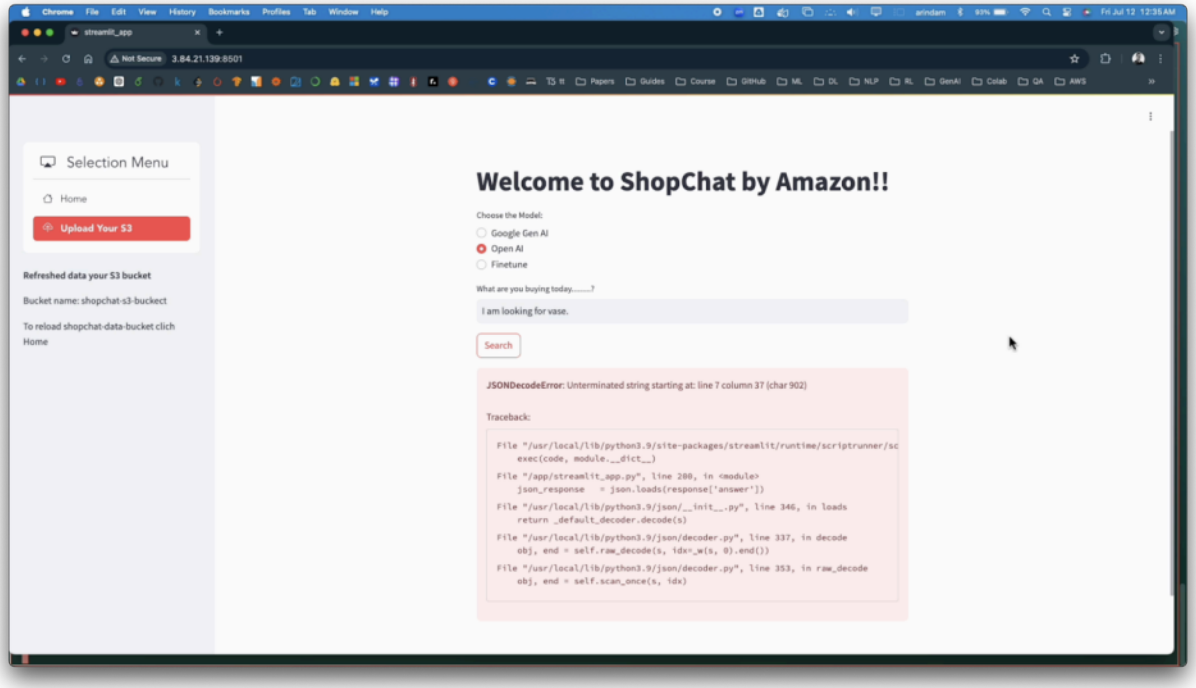
10.1 Finetune Embedding and LLM models

Synthetic questions generated by OpenAI model were not so buyer friendly and hence the finetuned model did not perform well on inference.

1. Synthetic questions generated by OpenAI model were not so buyer friendly and hence the finetuned model did not perform well on inference
2. Use human annotator to create better question-and-answer dataset
3. Create better questions and answers from the ABO data using better prompt
4. Try other Question-and-answer generation model to improve performance
5. Train for more epochs and more question-and-answer dataset
6. Try other Fine Model which can also improve on performance

10.2 Inference with OpenAI LLM model

Sometimes, we were facing issues with OpenAI model while inferencing, it cannot create the response in proper Json format due to the response context length. Using proper prompting and hyper parameters tuning we can control the response and avoid getting the error. One of the examples shown below. Also added in working model video demo



10.3 Response Thruput

Both these LLM provides fast response and response time is captured and displayed in terminal while running the app. The overall response time is faster because of loading and unloading the full vector databases and operation time within Streamlit app and can be reduced by creating proper REST interface API for inference

10.4 Bucket selection

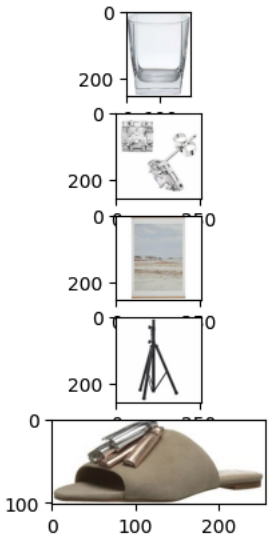
In the selection menu, after we change the S3 location, Streamlit loads the entire page including saving of models and vector database which loads fresh every time we change the bucket. This can also be avoided by creating proper REST interface API for inference

10.5 Image Caption

Captions generated by the Saleforce BLIP model is not trained to generate caption on the products. Hence the by adding caption to the dataset and doing embedding did not improve response performance. Few example of image captions are shown below:

	item_id	item_name_in_en_us	image_caption
0	B0896LJNLH	AmazonBasics Serene 16-Piece Old Fashioned and...	a close up of a glass of water on a white back...
1	B07HCR1LSQ	[Find] Amazon Collection Platinum Plated Sterl...	a pair of earrings with a square cut diamond a...
2	B075DQBBJZ	Arizona Desert Sand Horizon Photo with Wood Ha...	a white and brown photograph of a desert lands...
3	B07S74D9T7	AmazonBasics Adjustable Speaker Stand - 3.8 to...	a black tripod speaker stands on a white backg...
4	B01N27SMXC	Amazon Brand - The Fix Women's Foley Tassel Sl...	a close up of a pair of sandals with a metal b...

for the images:



11 Challenges

11.1 Data Handling

Handling large volumes of product data from multiple sources was challenging. Ensuring consistency and accuracy in the dataset required robust preprocessing and cleaning mechanisms and processing power

11.2 Language and Localization

Filtering data to ensure language consistency (e.g., retaining only English text) was crucial and challenging, given the multilingual nature of some data fields.

11.3 Integration of AI Models

Integrating different AI models for generating embeddings and ensuring they operate efficiently under varying loads was complex.

11.4 Scalability and Performance

Ensuring the system could scale effectively to handle real-time user queries without latency was a significant technical hurdle.

11.5 User Experience Design

Designing a chat interface that was both intuitive and effective at guiding users through the product search process required iterative design and testing.

12 Future Improvements

12.1 Model Enhancement

Upgrading to more advanced versions of AI models like GPT-4 or other latest models for better understanding and generating more contextually relevant responses.

12.2 Multi-language Support

Expanding the system to support multiple languages would make the platform more accessible to a global audience.

12.3 Personalization

Integrating user preferences and purchase history to tailor recommendations more closely to individual users.

12.4 Visual Search Capabilities

Implementing image recognition and visual search functionalities could enhance the user experience by allowing queries based on images rather than text.

12.5 Analytics and Feedback Loop

Incorporating analytics to track user satisfaction and system performance, using this data to refine AI responses and improve the overall system.