Assignment 6. MLOps with AWS

# Project Setup

## Clone project repo:

I forked the original repo into my GitHub account ([repo link](https://github.com/MohmedMonsef/oxford-genai-llmops-project))

Then cloned it locally

git clone https://github.com/MohmedMonsef/oxford-genai-llmops-project.git

And now, the repo is cloned to my local machine: A screenshot of a computer

AI-generated content may be incorrect.

## A screenshot of a computer AI-generated content may be incorrect.Build the Dev Container

## Setup .env file

Create the .env file and complete missing environment variables related to OpenAI and Comet Opik

A screen shot of a computer

AI-generated content may be incorrect.

## Install dependencies

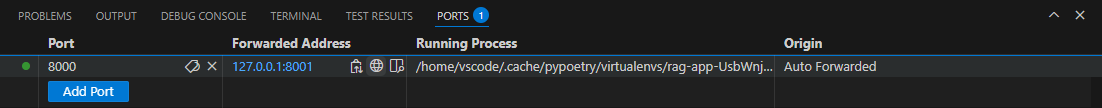
A computer screen shot of a program

AI-generated content may be incorrect.

## Run the application

A screenshot of a computer program

AI-generated content may be incorrect.

Inspect the server

A screenshot of a computer

AI-generated content may be incorrect.

# GitHub Actions

We have defined the following GitHub workflow to:

* Print the required message
* Install dependencies via poetry
* Run unit tests

name: CI Initialise

on:

    push:

      branches:

        - '\*\*'

jobs:

    initialise:

        runs-on: ubuntu-latest

        steps:

            - name: Checkout code

              uses: actions/checkout@v4

            - name: Print CI message

              run: echo "CI step initialising"

            - name: Setup python

              uses: actions/setup-python@v5

              with:

                python-version: '3.12'

            - name: Install poetry

              run: |

                curl -sSL https://install.python-poetry.org | python3 -

                echo "$HOME/.local/bin" >> $GITHUB\_PATH

            - name: Cache poetry dependencies

              uses: actions/cache@v4

              with:

                path: |

                    ~/.cache/pypoetry

                    ~/.cache/pip

                key: poetry-${{ runner.os }}-${{ hashFiles('\*\*/poetry.lock') }}

                restore-keys: |

                    poetry-${{ runner.os }}-

            - name: Install dependencies

              run: |

                cd rag-app

                poetry install --no-root

            - name: Run pytest

              run: |

                cd rag-app

                echo "${{ secrets.DUMMY\_ENV }}" > .env

                poetry run pytest

### Note

The tests provided in the original repo were buggy and did not pass. We solved these issues in [this](https://github.com/MohmedMonsef/oxford-genai-llmops-project/commit/892eb1897767ab4e43df1576d49e729538bdee22) commit.

Now, the tests are passing, and the CI workflow is running successfully.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

# Database setup

## Build the database

Outside the dev container.

$ cd rag-app

$ make build-db

A computer screen with text on it

AI-generated content may be incorrect.

## Test the database is working correctly

You need to install requirements for psql and make sure the .env file is written correctly to be able to source it.

$ cd rag-app

$ source .env

$ psql -h localhost -U $POSTGRES\_USER -d $POSTGRES\_DB -p $POSTGRES\_PORT

$ \dt

Output in the terminal

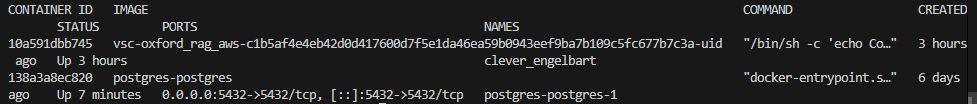
A screenshot of a computer

AI-generated content may be incorrect.

Also, we can test it is running via:

$ docker ps

Output in the terminal, you will find the postgres container running.



# Data Ingestion and Embedding

## Keyword for search

The keyword for search is defined in (rag-app/server/src/ingestion/arxiv\_client.py) A screen shot of a computer program

AI-generated content may be incorrect.

I have tried a different query keyword (llm) and downloaded the new data using

$ make download-data

A screen shot of a computer

AI-generated content may be incorrect.

Then ingest the data using

$ make run-ingestion

A screen shot of a computer

AI-generated content may be incorrect.

Make sure the rows are inserted in the database

$ psql -h localhost -U $POSTGRES\_USER -d $POSTGRES\_DB -p $POSTGRES\_PORT

mydb=# SELECT count(\*) FROM public.papers;

Output in the terminal:

A black screen with white text

AI-generated content may be incorrect.

Note, if you ran the ingestion twice, then you have inserted the 80 documents twice in the database. To clear the database and reset the auto-increment id, use the following command

$ psql -h localhost -U $POSTGRES\_USER -d $POSTGRES\_DB -p $POSTGRES\_PORT

mydb=# TRUNCATE TABLE public.papers RESTART IDENTITY;

This will reset the database to have 0 rows, and to fill it again, ingest the documents once again.

Output in the terminal:

A screen shot of a computer

AI-generated content may be incorrect.

# RAG Workflows

top-k is forced in “LIMIT %s”, and the used limit is passed in “execute()” method

A computer screen shot of text

AI-generated content may be incorrect.

## Exercise questions

### a. What does top\_k mean in this context (i.e., of RAG)?

top\_k specifies the number of most relevant document chunks to retrieve from the database based on their similarity to the query embedding. It controls how many context passages are fed to the language model.

### b. What will happen if I increase or decrease top\_k?

* **Increase top\_k** → More documents retrieved, potentially more relevant context but slower performance and higher token cost. Also, too large k may add irrelevant context that may confuse the LLM.
* **Decrease top\_k** → Fewer documents retrieved; faster and cheaper but might miss important context.

### c. What is the algorithm inside the database query and how does it work?

The algorithm is **cosine similarity** (implemented via the <=> operator in pgvector for vector types). It computes the angular distance between the query embedding and document embeddings:

* Cosine similarity = dot product of normalized vectors.
* Lower distance implies higher similarity.

### d. Why is the query to Postgres asking for results in ascending order?

Because the <=> operator returns **cosine distance**, not similarity — smaller values indicate closer (i.e., more similar) vectors. Sorting by ASC ensures the **most similar** chunks come first.

### There is an error control flow in the retrieve\_top\_k\_chunks\_endpoint that deals with a failure to retrieve chunks (i.e context). Under what scenarios could this happen?

Not sure about that, but some possibilities are:

* The papers table has no data
* Embeddings are missing or malformed
* The query embedding can't be compared (dimension mismatch)

### The process flow that happens when a user hits the “generate” endpoint with a query:

Steps:

* rag-app/server/src/controllers/generation.py => generate\_answer\_endpoint(…)
  + chunks = retrieve\_top\_k\_chunks(query)
  + generated\_response = generate\_response(query, chunks)
* return generated\_response

## Run the app

$ make run-app

Then navigate to <http://127.0.0.1:8000/docs>

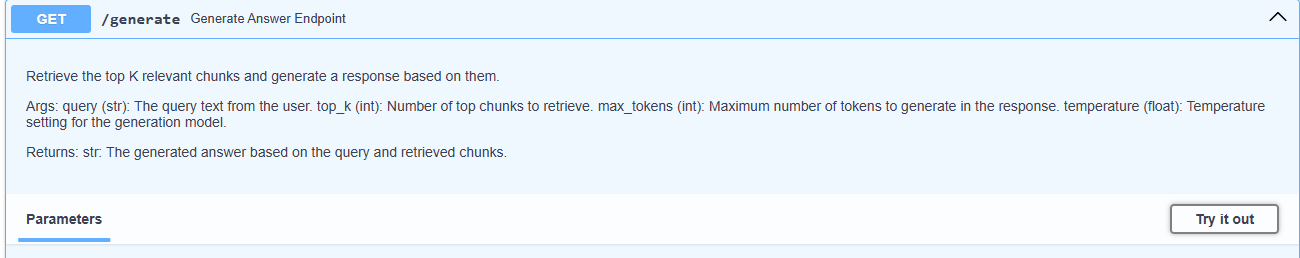
You will find swagger docs

A screenshot of a computer

AI-generated content may be incorrect.

## Generate

Try the generation endpoint



Insert your query and other parameters

A screenshot of a computer

AI-generated content may be incorrect.

Execute to get the response

A screenshot of a computer program

AI-generated content may be incorrect.

## Frontend

You can do the same but with a front-end client

A screen shot of a computer

AI-generated content may be incorrect.

navigate to <http://127.0.0.1:8502/>, you will find streamlit app running

A screenshot of a computer

AI-generated content may be incorrect.

# Tracing and Tracking

### It is possible to run some logic upon the initialization of your FastAPI server that will persist specific objects or configurations throughout the lifetime of the session. Can you see where this is done in the server/src/main.py file? What has been initialized, why is this a good place to run these commands?

Yes, this is done inside the lifespan\_context async function:

* **Where**: In server/src/main.py, it's done in the lifespan\_context function.
* **What is initialized**:
  + opik.configure() — a global config setup
  + embedding\_model = SentenceTransformer("all-MiniLM-L6-v2") — a sentence embedding model loaded once into memory.
* **Why is it a good place**:
  + The lifespan context is the **ideal place for startup logic** because it's run **only once** when the server starts (not on every request), making it efficient for heavy initialization like loading models.
  + It also allows proper **cleanup** at shutdown via the “finally” block.
  + Keeping models in memory here allows **reusing expensive resources** without reloading them per request, improving performance and scalability.

### Add some tracking and tracing to other services and functions in the application. Check that these are successfully logged into your Opik instance in the online portal.

We Tracked the get\_db\_connection(…) function

A computer screen shot of a black screen

AI-generated content may be incorrect.

It is successfully tracked in opik

A screenshot of a computer

AI-generated content may be incorrect.

Also, other functions are being tracked

A screenshot of a computer

AI-generated content may be incorrect.

### Online Evaluation

Created a custom evaluation metric

A screenshot of a computer

AI-generated content may be incorrect.

And check it is running

A screenshot of a computer

AI-generated content may be incorrect.