pixegami / deploy-rag-to-aws

Code Issues 1 Pull requests 1 Actions Projects Security Insights

A sample project for deploying a RAG app to AWS

deploy-rag-to-aws.vercel.app

64 stars 50 forks 6 watching Branches Activity
Tags

Public repository

frontend-tutorial 3 Branches 0 Tags Go to file t Go to file Add file Code

This branch is 14 commits ahead of main .

pixegami update: README f3d3e83 · last year

image	feat: add CORS headers to the API	last year
rag-app-frontend	feat: replace the layout with the new components	last year
rag-cdk-infra	feat: update DB model with user_id and ttl, and a...	last year
test	feat: add the list_queries API	last year
.gitignore	feat: Add Dockerfile	last year
README.md	update: README	last year

Deploy RAG/AI App To AWS

Getting Started

Configure AWS

You need to have an AWS account, and AWS CLI set up on your machine. You'll also need to have Bedrock enabled on AWS (and granted model access to Claude or whatever you want to use).

Update .env File with AWS Credentials

Create a file named `.env` in `image/`. Do NOT commit the file to `.git`. The file should have content like this:

```
AWS_ACCESS_KEY_ID=XXXXX
AWS_SECRET_ACCESS_KEY=XXXXX
AWS_DEFAULT_REGION=us-east-1
TABLE_NAME=YourTableName
```

This will be used by Docker for when we want to test the image locally. The AWS keys are just your normal AWS credentials and region you want to run this in (even when running locally you will still need access to Bedrock LLM and to the DynamoDB table to write/read the data).

You'll also need a TABLE_NAME for the DynamoDB table for this to work (so you'll have to create that first).

README

```
pip install -r image/requirements.txt
```

Building the Vector DB

Put all the PDF source files you want into `image/src/data/source/`. Then go `image` and run:

```
# Use "--reset" if you want to overwrite an existing DB.
python populate_database.py --reset
```

Running the App

```
# Execute from image/src directory
cd image/src
python rag_app/query_rag.py "how much does a landing page cost?"
```

Example output:

```
Answer the question based on the above context: How much does a landing page cost to develop?

Response:  Based on the context provided, the cost for a landing page service offered by Galaxy Design Agency is $4,820.
Specifically, under the "Our Services" section, it states "Landing Page for Small Businesses ($4,820)" when describing the
landing page service. So the cost listed for a landing page is $4,820.
Sources: ['src/data/source/galaxy-design-client-guide.pdf:1:0', 'src/data/source/galaxy-design-client-guide.pdf:7:0',
'src/data/source/galaxy-design-client-guide.pdf:7:1']
```

Starting FastAPI Server

```
# From image/src directory.
python app_api_handler.py
```

Then go to `http://0.0.0.0:8000/docs` to try it out.

Using Docker Image

Build and Test the Image Locally

These commands can be run from `image/` directory to build, test, and serve the app locally.

```
docker build --platform linux/amd64 -t aws_rag_app .
```

This will build the image (using linux amd64 as the platform — we need this for `pysqlite3` for Chroma).

```
# Run the container using command `python app_work_handler.main`
docker run --rm -it \
    --entrypoint python \
    --env-file .env \
    aws_rag_app app_work_handler.py
```

This will test the image, seeing if it can run the RAG/AI component with a hard-coded question (see `app_work_handler.py`). But since it uses Bedrock as the embeddings and LLM platform, you will need an AWS account and have all the environment variables for your access set (`AWS_ACCESS_KEY_ID`, etc).

You will also need to have Bedrock's models enabled and granted for the region you are running this in.

Running Locally as a Server

Assuming you've build the image from the previous step.

```
docker run --rm -p 8000:8000 \
    --entrypoint python \
    --env-file .env \
    aws_rag_app app_api_handler.py
```

Testing Locally

After running the Docker container on localhost, you can access an interactive API page locally to test it: `http://0.0.0.0:8000/docs` .

```
curl -X 'POST' \
  'http://0.0.0.0:8000/submit_query' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "query_text": "How much does a landing page for a small business cost?"
}'
```

Unit Testing

Once you have a server running locally on `localhost:8000` , you can run the unit tests in `test/` from the root folder. You'll need to have `pytest` installed (`pip install pytest`).

```
pytest  # Run all tests
```

```
pytest -k test_can_submit_query -s  # Run a specific test. Print output.
```

Deploy to AWS

I have put all the AWS CDK files into `rag-cdk-infra/` . Go into the folder and install the Node dependencies.

```
npm install
```

Then run this command to deploy it (assuming you have AWS CLI already set up, and AWS CDK already bootstrapped). I recommend deploying to `us-east-1` to start with (since all the AI models are there).

```
cdk deploy
```

Front End

Install Tools to Generate API Client

```
npm install @openapitools/openapi-generator-cli -g
```

There is a command script in the package.json file to generate the client library for the API.

```
{
  "generate-api-client": "openapi-generator-cli generate -i http://0.0.0.0:8000/openapi.json -g typescript-fetch -o src/api-cli
}
```

To use it, it will fetch the OpenAPI schema from `http://0.0.0.0:8000` (assuming it's a FastAPI server and makes it available). And generate a TypeScript client to `src/api-client`.

We'll need to make sure it's generated each time.

Generate API Client

Generate the client into `src/api-client/` first.

```
npm run generate-api-client
```

Component Library

Using shadcn/ui. I don't think you need to run this, it's already part of the project via Git — but here's what I had to run, just for reference.

```
npx shadcn-ui@latest init
```

Then install each component separately.

```
npx shadcn-ui@latest add button
npx shadcn-ui@latest add textarea
npx shadcn-ui@latest add card
npx shadcn-ui@latest add skeleton
```

Releases

No releases published

Packages

No packages published

Deployments 11

Production last year

Preview

+ 9 deployments

Languages

Python 68.9% **TypeScript** 20.6% **JavaScript** 7.1% **Dockerfile** 3.4%