Deploy a RAG application with vector search in Firestore

Step1 - First create a Firestore database with default settings.

Step 2 - For **tasks 1, 2 and 3** use below code. Do change (Highlighted) bucket names as mentioned in the challenge lab.

```
# Install required python packages
!pip install --quiet --upgrade google_cloud_firestore google_cloud_aiplatform langchain langchain-google-vertexai langchain_community langchain_experimental pymupdf

# Restart the runtime

# Import the following packages by running the following command:
import vertexai
from vertexai.language_models import TextEmbeddingModel
from vertexai.generative_models import GenerativeModel
import pickle
from IPython.display import display, Markdown
from langchain_google_vertexai import VertexAIEmbeddings
from langchain_community.document_loaders import PyMuPDFLoader
from langchain_experimental.text_splitter import SemanticChunker
from google.cloud import firestore
from google.cloud.firestore_v1.vector import Vector
from google.cloud.firestore_v1.base_vector_query import DistanceMeasure

# Initialize Vertex AI with your project-id and a location
PROJECT_ID = ! gcloud config get-value project
PROJECT_ID = PROJECT_ID[0]
LOCATION = "us-central1" # @param {type:"string"}
print(PROJECT_ID)
vertexai.init(project=PROJECT_ID, location=LOCATION)

# Populate a variable named embedding_model with an instance of the
# langchain_google_vertexai class VertexAIEmbeddings.

from langchain_google_vertexai import VertexAIEmbeddings
embedding_model = VertexAIEmbeddings(model_name="text-embedding-004")

# Download the New York City Department of Health and Mental Hygiene's Food
# Protection Training Manual. This document will serve as our RAG source content.
```

```
!gcloud storage cp gs://<bucket>/nyc_food_safety_manual.pdf .

# Use the LangChain class PyMuPDFLoader to load the contents of the PDF
from langchain_community.document_loaders import PyMuPDFLoader
loader = PyMuPDFLoader("./nyc_food_safety_manual.pdf")
data = loader.load()

# Create a function to do some basic cleaning on artifacts found in this particular document.
def clean_page(page):
  return page.page_content.replace("-\n","")\
                .replace("\n"," ")\
                .replace("\x02","")\
                .replace("\x03","")\
                .replace("fo d P R O T E C T I O N T R A I N I N G M A N U A L","")\
                .replace("N E W Y O R K C I T Y D E P A R T M E N T O F H E A L T H & M E N T A L H Y G
I E N E","")

# Create a variable called cleaned_pages that is a list of strings, with each string being a page of
content cleaned by above function.
cleaned_pages = []
for pages in data:
  cleaned_pages.append(clean_page(pages))

# Use LangChain's SemanticChunker with the embedding_model created earlier to split the first
five pages of cleaned_pages into text chunks.
text_splitter = SemanticChunker(embedding_model)
docs = text_splitter.create_documents(cleaned_pages[0:4])
chunked_content = [doc.page_content for doc in docs]

# Use the embedding_model to generate embeddings of the text chunks, saving them to a list called
chunked_embeddings.
chunked_embeddings = embedding_model.embed_documents(chunked_content)

# Above code only chunks & create embeddings of a short section of the
# document for demo purpose. To get the chunks & corresponding embeddings for
# the full document, run the following code to download pre-created chunks
# & embeddings
!gsutil cp gs://<bucket>/chunked_content.pkl .
!gsutil cp gs://<bucket>/chunked_embeddings.pkl .
chunked_content = pickle.load(open("chunked_content.pkl", "rb"))
chunked_embeddings = pickle.load(open("chunked_embeddings.pkl", "rb"))

# Create a Firestore database using console with the default name of (default)
# in Native Mode and leave the other settings to default.
```

```
# Populate a db variable with a Firestore Client.
db = firestore.Client(project=PROJECT_ID)
```

Note – Replace project=PROJECT_ID with actual project ID in case of any errors

```
# Use a variable called collection to create a reference to a collection named food-safety.
collection = db.collection('food-safety')

# Using a combination of our lists chunked_content and chunked_embeddings,
# add a document to your collection for each of your chunked documents.
for i, (content, embedding) in enumerate(zip(chunked_content, chunked_embeddings)):
  doc_ref = collection.document(f"doc_{i}")
  doc_ref.set({
    "content": content,
    "embedding": Vector(embedding)
  })

# Create a vector index for your collection using your embedding field using gcloud firestore indexes command
!gcloud firestore indexes composite create \
--collection-group=food-safety \
--query-scope=COLLECTION \
--field-config field-path=embedding,vector-config='{"dimension":"768", "flat": "{}"}' \
--project=PROJECT_ID

# Create a function to receive a query, get its embedding, and compile a context
# consisting of the text from the 5 documents with the most similar embeddings.
def search_vector_database(query: str):
 context = ""
 query_embedding = embedding_model.embed_query(query)
 vector_query = collection.find_nearest(
  vector_field="embedding",
  query_vector=Vector(query_embedding),
  distance_measure=DistanceMeasure.EUCLIDEAN,
  limit=5,
 )
 docs = vector_query.stream()
 context = [result.to_dict()['content'] for result in docs]
 return context

# Call the function with a sample query to confirm it's functionality.
search_vector_database("How should I store food?")
```

Follow the below steps for **task 4** –

Note – Execute all the below mentioned commands in CloudShell

Step3 – Execute the below command.

gcloud storage cp -r gs://partner-genai-bucket/genai069/gen-ai-assessment .

cd gen-ai-assessment

python3 -m pip install -r requirements.txt

Step4 – Run the application locally

python3 main.py

Step5 – Enable Artifact Registry

gcloud services enable artifactregistry.googleapis.com run.googleapis.com aiplatform.googleapis.com

Step6 – Create Artifact Registry - cymbal-artifact-repo

```
gcloud artifacts repositories create cymbal-artifact-repo\
  --repository-format=docker \
  --location=us-central1 \
  --description="Docker repository for food-safety image"
```

Step7 – Authenticate your Docker client

gcloud auth configure-docker us-central1-docker.pkg.dev

Step8 – Update the main.py file located in the folder **"generative-ai-assessment"** with the below code.

```
import os

import yaml

import logging

import google.cloud.logging

from flask import Flask, render_template, request


from google.cloud import firestore
```

```
from google.cloud.firestore_v1.vector import Vector
from google.cloud.firestore_v1.base_vector_query import DistanceMeasure

import vertexai
from vertexai.generative_models import (
    GenerativeModel,
    GenerationConfig,
    HarmCategory,
    HarmBlockThreshold,
    SafetySetting,
)
from vertexai.language_models import TextEmbeddingInput, TextEmbeddingModel
from langchain_google_vertexai import VertexAIEmbeddings

# Configure Cloud Logging
logging_client = google.cloud.logging.Client()
logging_client.setup_logging()
logging.basicConfig(level=logging.INFO)

# Read application variables
BOTNAME = "FreshBot"
SUBTITLE = "Restaurant safety expert bot"

# Set up the dangerous content safety config to block only high-probability dangerous content
safety_config = [
    SafetySetting(
        category=HarmCategory.HARM_CATEGORY_DANGEROUS_CONTENT,
        threshold=HarmBlockThreshold.BLOCK_ONLY_HIGH,
    ),
```

```
]

app = Flask(__name__)

# Initializing the Firebase client
db = firestore.Client()

# Instantiate a collection reference
collection = db.collection('food-safety')

# Instantiate an embedding model here
embedding_model = VertexAIEmbeddings(model_name="text-embedding-004")

# Instantiate a Generative AI model here
gen_model = GenerativeModel("gemini-1.5-pro-001",
   generation_config=GenerationConfig(temperature=0),
   safety_settings=safety_config,)

# Function to return relevant context from vector database
def search_vector_database(query: str):
   # Initialize context as an empty string
   context = ""

   # Step 1: Generate the embedding of the query
   query_embedding = embedding_model.embed_query(query)

   # Step 2: Get the 5 nearest neighbors from collection using Firestore's find_nearest operation
   vector_query = collection.find_nearest(
   vector_field="embedding",
```

```
    query_vector=Vector(query_embedding),
    distance_measure=DistanceMeasure.EUCLIDEAN,
    limit=5,
    )

    # Step 3: Iterate over document snapshots and combine them into a single context string
    docs = vector_query.stream()
    context = [result.to_dict()['content'] for result in docs]

    # Don't delete this logging statement.
    logging.info(
        context, extra={"labels": {"service": "food-safety-service", "component": "context"}}
    )
    return context


# Pass Gemini the context data, generate a response, and return the response text.
def ask_gemini(question):

    # 1. Create a prompt_template with instructions to the model
    # to use provided context info to answer the question.
    prompt_template = """
    You are a restaurant safety expert bot. Use the following context to answer the user's question.

    Context: {context}

    Question: {question}

    Answer as accurately and helpfully as possible.
```

```
    """

    # 2. Use search_vector_database function to retrieve context relevant to the question.
    context = search_vector_database(question)

    # 3. Format the prompt template with the question & context
    prompt = prompt_template.format(context=context, question=question)

    # 4. Pass the complete prompt template to Gemini and get the text of its response.
    response = gen_model.generate_content(
        prompt,
    ).text

    return response


# The Home page route
@app.route("/", methods=["POST", "GET"])
def main():

    # The user clicked on a link to the Home page
    # They haven't yet submitted the form
    if request.method == "GET":
        question = ""
        answer = "Hi, I'm FoodBot, what can I do for you?"

    # The user asked a question and submitted the form
    # The request.method would equal 'POST'
    else:
```

```
        question = request.form["input"]
        # Do not delete this logging statement.
        logging.info(
            question,
            extra={"labels": {"service": "food-safety-service", "component": "question"}},
        )

        # Ask Gemini to answer the question using the data
        # from the database
        answer = ask_gemini(question)

    # Do not delete this logging statement.
    logging.info(
        answer, extra={"labels": {"service": "food-safety-service", "component": "answer"}}
    )
    print("Answer: " + answer)

    # Display the home page with the required variables set
    config = {
        "title": BOTNAME,
        "subtitle": SUBTITLE,
        "botname": BOTNAME,
        "message": answer,
        "input": question,
    }

    return render_template("index.html", config=config)

if __name__ == "__main__":
```

```
    app.run(debug=True, host="0.0.0.0", port=int(os.environ.get("PORT", 8080)))
```

Step9 – Now let's build docker image and push to artifact registry.

```
export PROJECT_ID=$(gcloud config get-value project)


docker build -t us-central1-docker.pkg.dev/$PROJECT_ID/cymbal-artifact-repo/cymbal-image:latest -f Dockerfile .

docker push us-central1-docker.pkg.dev/$PROJECT_ID/cymbal-artifact-repo/cymbal-image:latest
```

Step10 – Deploy the service to Cloud Run

Note – Give the exact service name as per the instructions in the lab. Example, you might need to change the service name from **cymbal-freshbot** to **cymbal-freshbot-service** etc

```
gcloud run deploy cymbal-freshbot \
  --image=us-central1-docker.pkg.dev/$PROJECT_ID/cymbal-artifact-repo/cymbal-image:latest \
  --region=us-central1 \
  --allow-unauthenticated \
  --min-instances=0 \
  --max-instances=1
```

Step11 – Allow application access to all users

```
gcloud beta run services add-iam-policy-binding \
  --region=us-central1 \
  --member=allUsers \
  --role=roles/run.invoker \
  cymbal-freshbot
```