2etmjimw2

December 3, 2024

[ ]:
```
import os
import re
import io
import json
import uuid
import asyncio
import logging
import warnings
import tempfile
from typing import List, Dict, Any
from concurrent.futures import ThreadPoolExecutor
from fastapi import FastAPI, File, UploadFile, Form, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from fastapi.responses import JSONResponse

from pydantic import BaseModel

from requests_aws4auth import AWS4Auth
from requests import HTTPException

from PIL import Image

from opensearchpy import OpenSearch, RequestsHttpConnection

from azure.storage.blob import BlobServiceClient

from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.messages import HumanMessage
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnableLambda, RunnablePassthrough

from langchain_text_splitters import RecursiveCharacterTextSplitter

from langchain.vectorstores import OpenSearchVectorSearch
from langchain.embeddings import OpenAIEmbeddings
from langchain.schema import Document
from langchain.chat_models import ChatOpenAI
```

```
from langchain.retrievers.multi_vector import MultiVectorRetriever
from langchain.storage import InMemoryStore

from langchain_community.vectorstores import OpenSearchVectorSearch

import boto3
from IPython.display import HTML, display


# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)
```

1. Data Ingestion:

a. Provide a dataset (e.g., JSON, CSV, or unstructured text files) that includes a mix of structured and unstructured data.

b. Ask the candidate to create a pipeline to load this data into a database of their choice, ensuring the schema is optimized for querying.

Dataset: The code processes unstructured data like PDFs and extracts text, tables, and images. It also handles JSON files from APIs, fulfilling the requirement for mixed structured and unstructured data. Pipeline: The ingest_pdf endpoint ingests PDF files and processes them into JSON using the upload_pdf_and_download_json function.

The extracted data (text, tables, images) is transformed into a structured format and indexed in OpenSearch. This ensures the schema is optimized for querying.

[ ]:
```
def upload_pdf_and_download_json(pdf_path, unique_id, TEMP_DIR):
    start_time = time.time()

    # Define payload parameters consistent with the working curl command
    payload = {
        'html_url': '',
        'summarize_figures_gemini': 'false',
        'summarize_tables': 'false',
        'extract_tables_with_gemini': 'true'
    }

    # Define headers, including the 'Accept' header
    headers = {
        'Accept': 'application/json'
    }

    # Prepare the file for upload
    with open(pdf_path, 'rb') as f:
        files = {
            'file': (os.path.basename(pdf_path), f, 'application/pdf')
```

```
        }

        # Log the request details
        logger.info(f"Uploading PDF: {pdf_path}")
        logger.info(f"POST {DOCUMENT_EXTRACTOR_API_URL}")
        logger.info(f"Payload: {payload}")
        logger.info(f"Headers: {headers}")

        try:
            # Send the POST request with both connect and read timeouts
            response = requests.post(
                DOCUMENT_EXTRACTOR_API_URL,
                data=payload,
                files=files,
                headers=headers,
                timeout=(3000, 3000)  # (connect_timeout, read_timeout) in␣
↪seconds
            )

            # Log the response status
            logger.info(f"Received response with status code: {response.
↪status_code}")

            # Raise an exception for HTTP error responses (4xx and 5xx)
            response.raise_for_status()

            # Log the response content type
            logger.info(f"Response Content-Type: {response.headers.
↪get('Content-Type')}")

        except requests.exceptions.Timeout:
            logger.error("Request timed out.")
            raise Exception("Failed to extract data: The request timed out.")
        except requests.exceptions.HTTPError as http_err:
            logger.error(f"HTTP error occurred: {http_err}")
            raise Exception(f"Failed to extract data: HTTP error occurred:␣
↪{http_err}")
        except requests.exceptions.RequestException as req_err:
            logger.error(f"Request exception occurred: {req_err}")
            raise Exception(f"Failed to extract data: Request exception␣
↪occurred: {req_err}")

    try:
        # Attempt to parse the JSON response
        data = response.json()
        logger.info("################ Data Extraction ########################")
        logger.info(json.dumps(data, indent=4))
```

```
    except json.JSONDecodeError:
        logger.error("Failed to decode JSON response.")
        raise Exception("Failed to extract data: Unable to decode JSON response.
 ↪")

    end_time = time.time()
    elapsed_time = end_time - start_time
    logger.info(f"Elapsed Time: {elapsed_time:.2f} seconds")

    # Define the path to save the JSON data
    json_filename = f"{unique_id}_extracted_data.json"
    json_path = os.path.join(TEMP_DIR, json_filename)

    # Ensure the TEMP_DIR exists
    os.makedirs(TEMP_DIR, exist_ok=True)

    # Save the JSON data to a file
    try:
        with open(json_path, 'w') as json_file:
            json.dump(data, json_file, indent=4)
        logger.info(f"Extracted data saved to {json_path}")
    except IOError as io_err:
        logger.error(f"IO error occurred while writing JSON file: {io_err}")
        raise Exception(f"Failed to save extracted data: {io_err}")

    return json_path



@app.post("/SFRAG/ingest")
def ingest_pdf(index_name: str = Form(...), file: UploadFile = File(...)):

    image_summaries = []
    text_summaries = []
    table_summaries = []
    import os
    base_dir = os.getcwd()

    BASE_OUTPUT_DIR = os.path.join(base_dir, "image_files")
    TEMP_DIR = os.path.join(base_dir, "pdf_files")


    if not os.path.exists(BASE_OUTPUT_DIR):
        os.makedirs(BASE_OUTPUT_DIR)

    if not os.path.exists(TEMP_DIR):
        os.makedirs(TEMP_DIR)
```

```
    dynamic_output_dir = None
    temp_pdf_path = None
    json_path = None

    try:
        unique_id = str(uuid.uuid4())
        dynamic_output_dir = os.path.join(BASE_OUTPUT_DIR, f"{unique_id}")
        temp_pdf_pathbase = os.path.join(TEMP_DIR, f"{unique_id}")
        temp_pdf_path = os.path.join(temp_pdf_pathbase, file.filename)
        print("temp_pdf_path", temp_pdf_path)

        if not os.path.exists(dynamic_output_dir):
            os.makedirs(dynamic_output_dir)
        if not os.path.exists(temp_pdf_pathbase):
            os.makedirs(temp_pdf_pathbase)


        with open(temp_pdf_path, 'wb') as f:
            f.write(file.file.read())

        file_name = file.filename

        file.file.seek(0)

        filenames_in_index = list_all_filenames_in_index(index_name)
        if file_name in filenames_in_index:
            print(f"File '{file_name}' already exists in index '{index_name}'.␣
↪Deleting existing documents.")
            delete_documents_by_filename(index_name, file_name)
        else:
            print(f"File '{file_name}' does not exist in index '{index_name}'.␣
↪Proceeding with ingestion.")


        container_name = f"{BASE_CONTAINER_NAME}/Project/{index_name}"


        upload_to_blob(file, STORAGE_ACCOUNT_NAME, container_name)
        input_file_url = get_presigned_url(file_name, STORAGE_ACCOUNT_NAME,␣
↪container_name)


        json_path = upload_pdf_and_download_json(temp_pdf_path, 30,␣
↪temp_pdf_pathbase, file_name, input_file_url)
        create_index_if_not_exists(index_name)
```

```
        with open(json_path, 'r') as f:
            data = json.load(f)
        print("data extraction", data)

        print("********input_file_url********", input_file_url)
        imageresult = extract_imageresult(data)
        tableresult = extract_tableresult(data)
        textresult = extract_textresult(data)

        print("imageresult", imageresult)
        print("tableresult", tableresult)
        print("textresult", textresult)

        sorted_keys = sorted(imageresult.keys(), key=int)
        imageurl_list = [imageresult[key]['url'] for key in sorted_keys if␣
↪'url' in imageresult[key]]

        imageno = 0
        for idx, blob_url in enumerate(imageurl_list, start=1):
            output_file_path = os.path.join(dynamic_output_dir, f"{idx}.png")
            download_blob(AZURE_CONN_STRING, AZURE_CONTAINER_NAME, blob_url,␣
↪output_file_path)

        img_base64_list, image_summaries =␣
↪generate_img_summaries(dynamic_output_dir)

        print("image_summaries")
        print(image_summaries)
        imageno =len(image_summaries)


        texts_list = [textresult[key]['output'] for key in sorted(textresult.
↪keys(), key=int) if 'output' in textresult[key]]
        table_list = [tableresult[key]['output'] for key in sorted(tableresult.
↪keys(), key=int) if 'output' in tableresult[key]]

        text_summaries, table_summaries = generate_text_summaries(
            texts_list, table_list, summarize_texts=True
        )
        print("text_summaries")
        print(text_summaries)
        tableno =len(table_summaries)
        print("table_summaries")
        print(table_summaries)
        tableno =len(table_summaries)
```

```
        image_metadata = create_image_metadata(imageresult, file_name,␣
↪input_file_url)
        table_metadata = create_table_metadata(tableresult, file_name,␣
↪input_file_url)
        text_metadata = create_text_metadata(textresult, file_name,␣
↪input_file_url)

        print("image_metadata")
        print(image_metadata)
        print("table_metadata")
        print(table_metadata)
        print("text_metadata")
        print(text_metadata)


        vectorstore = get_vectorstore(index_name)
        print("vectorstore",vectorstore)
        retriever_multi_vector_img = create_multi_vector_retriever(
            vectorstore,
            text_summaries, texts_list, text_metadata,
            table_summaries, table_list, table_metadata,
            image_summaries, img_base64_list, image_metadata, file_name,␣
↪index_name
        )
        print("Retriever and vector store setup completed.")

        return {"status": "Index ingested successfully", "index_name":␣
↪index_name, "imagecount":imageno, "tablecount":tableno}

    except Exception as e:

        print(f"Error during processing: {e}")
        return {"status": "Error", "message": str(e)}

    finally:

        if dynamic_output_dir and os.path.exists(dynamic_output_dir):
            shutil.rmtree(dynamic_output_dir)


        if temp_pdf_pathbase and os.path.exists(temp_pdf_pathbase):
            shutil.rmtree(temp_pdf_pathbase)

        if json_path and os.path.exists(json_path):
            os.remove(json_path)
```

1 2. Data Preprocessing:

a. The data may contain noise or require transformation (e.g., text cleaning, parsing nested JSON, handling missing values).

b. The candidate should demonstrate how they preprocess the data for efficient storage and later retrieval.

The code preprocesses text, tables, and image data: extract_textresult, extract_tableresult, and extract_imageresult parse and clean the extracted JSON.

Metadata is generated for each content type to ensure easy retrieval.

Missing or nested data is handled through checks in the respective extraction functions, ensuring robust preprocessing.

[ ]:
```python
# Process textresult
def extract_textresult(data):
# Initialize variables
    sections = []
    current_section = None
    section_counter = 0

    # Get only the page number keys
    page_keys = [k for k in data.keys() if k.isdigit()]

    # Build a mapping from page numbers to page URLs
    page_urls = {}
    for page_num_str in sorted(page_keys, key=int):
        page_num = int(page_num_str)
        page_content = data[page_num_str]
        bbox_img_url = page_content.get('bbox_img_url')
        if bbox_img_url:
            page_urls[page_num] = bbox_img_url

    # Iterate over the pages in order
    for page_num_str in sorted(page_keys, key=int):
        page_num = int(page_num_str)
        page_content = data[page_num_str]
        bboxes_info = page_content.get('bboxes_info', [])
        for item in bboxes_info:
            # Ensure 'item' is a dictionary
            if isinstance(item, dict):
                label = item.get('label')
                output = item.get('output', '')

                # Handle 'output' being a list or string
                if isinstance(output, list):
                    output = '\n'.join(output)
                elif isinstance(output, str):
```

```
                output = output.strip()
            else:
                output = str(output)

            if label == 'TITLE':
                if current_section is not None:
                    sections.append(current_section)
                section_counter += 1
                current_section = {
                    'output': f"title- {output}\n",
                    'page_numbers': set([page_num])
                }
            elif label in ('TEXT', 'FORMULA', 'FIGURE', 'TABLE'):
                if current_section is None:
                    # Initialize a default section before any TITLE
                    section_counter += 1
                    current_section = {
                        'output': '',
                        'page_numbers': set([page_num])
                    }
                if label == 'TEXT':
                    current_section['output'] += f"{output}\n"
                elif label == 'FORMULA':
                    current_section['output'] += f"formula- {output}\n"

                current_section['page_numbers'].add(page_num)
            else:
                # Skip other labels
                continue

    # After loop ends, save the last section if it exists
    if current_section is not None:
        sections.append(current_section)

    # Build the result dictionary
    textresult = {}
    for i, section in enumerate(sections, start=1):
        page_numbers = sorted(section['page_numbers'])
        urls = [page_urls[page_num] for page_num in page_numbers if page_num in␣
↪page_urls]
        textresult[str(i)] = {
            'output': section['output'],
            'page_numbers': page_numbers,
            'url': urls,
            'type': "TEXT"
        }
```

```
    return textresult



# Process tableresult
def extract_tableresult(data):
    # Initialize the result dictionary
    tableresult = {}
    count = 1

    # Iterate over the pages
    for page_num_str, page_content in data.items():
        # Skip keys that are not page numbers (e.g., 'bbox_pdf_url',␣
 ↪'file_name')
        if not page_num_str.isdigit():
            continue
        page_num = int(page_num_str)
        # Get the list of 'bboxes_info'
        bboxes_info = page_content.get('bboxes_info', [])
        # Iterate over the bboxes_info
        for item in bboxes_info:
            # Ensure 'item' is not None and is a dictionary
            if isinstance(item, dict):
                # Check if 'label' exists and equals 'TABLE'
                if item.get('label') == 'TABLE':
                    output = item.get('output')
                    # If 'output' is missing, skip this item
                    if not output:
                        continue
                    # Handle 'output' being a list or string
                    if isinstance(output, list):
                        output = '\n'.join(output)
                    elif isinstance(output, str):
                        output = output.strip()
                    else:
                        output = str(output)
                    # Get 'page_num' from item or use 'page_num'
                    page_num_item = item.get('page_num', page_num)
                    # Get 'img_url' from the TABLE item
                    img_url = item.get('img_url', '')
                    # Add to result
                    tableresult[str(count)] = {
                        'output': output,
                        'page_numbers': [page_num_item],
                        'url': img_url,
                        'type': "TABLE"
                    }
```

```
                    count += 1

    return tableresult



# Process imageresult
def extract_imageresult(data):
    # Initialize the result dictionary
    imageresult = {}
    count = 1

    # Iterate over the pages
    for page_num_str, page_content in data.items():
        # Skip keys that are not page numbers (e.g., 'bbox_pdf_url',␣
 ↪'file_name')
        if not page_num_str.isdigit():
            continue
        page_num = int(page_num_str)

        # Get the list of 'bboxes_info'
        bboxes_info = page_content.get('bboxes_info', [])

        # Iterate over the bboxes_info
        for item in bboxes_info:
            # Ensure 'item' is not None and is a dictionary
            if isinstance(item, dict):
                label = item.get('label')

                # Check if 'label' exists and equals 'FIGURE' or 'TABLE'
                if label in ('FIGURE', 'TABLE'):
                    img_url = item.get('img_url')

                    # If 'img_url' is missing, skip this item
                    if not img_url:
                        continue

                    # Get 'page_num' from item or use the current page number
                    page_num_item = item.get('page_num', page_num)

                    # Set 'output' to the label type ('FIGURE' or 'TABLE')
                    output_type = label

                    # Add to result
                    imageresult[str(count)] = {
                        'output': None,
                        'type': output_type,
```

```
                        'page_numbers': [page_num_item],
                        'url': img_url
                    }
                    count += 1
    return imageresult
```

2 3. Vectorization:

a. Using a pre-trained language model or embeddings model, ask the candidate to convert the unstructured text into embeddings.

b. Store these embeddings in a vector storage solution of their choice, ensuring the pipeline can handle batch processing for larger datasets.

Embeddings:

Text, table summaries, and image descriptions are vectorized using OpenAIEmbeddings.

The processed data is stored in an OpenSearch vector database, supporting batch processing and efficient embedding-based queries.

Batch Processing:

Summaries and metadata for text, tables, and images are created in batches using parallel processing where applicable.

[ ]:
```
# Embeddings:
#  Vectorization of Text, Table Summaries, and Image Descriptions

# The OpenAIEmbeddings class is used to generate embeddings for text, table␣
 ↪summaries, and image descriptions. These embeddings are then stored in an␣
 ↪OpenSearch vector database.

from langchain.vectorstores import OpenSearchVectorSearch
from langchain.embeddings import OpenAIEmbeddings
from langchain.schema import Document

# Initialize OpenAI embeddings
embeddings = OpenAIEmbeddings()

# Create vector store using OpenSearch
vectorstore = OpenSearchVectorSearch.from_documents(
    documents=documents,  # List of Document objects with content and metadata
    embedding=embeddings,  # Embedding function to convert content into vectors
    opensearch_url=OPENSEARCH_URL,  # OpenSearch endpoint
    http_auth=awsauth,
    use_ssl=True,
    verify_certs=True,
    connection_class=RequestsHttpConnection,
```

```
    index_name=index_name,  # Target index for storing embeddings
    timeout=300,
    max_retries=3,
    retry_on_timeout=True
)
```

2.0.1 How it works:

The from_documents function takes a list of Document objects (containing text, table summaries, and image descriptions) and generates embeddings using the OpenAIEmbeddings model. These embeddings are stored in OpenSearch, which supports efficient vector-based queries.

[ ]:
```
# 2. Batch Processing
# Summaries and metadata are processed in batches to optimize performance.

# Code for Batch Processing Summaries
# The summaries for text, tables, and images are generated in batches using the␣
 ↪following chain

def generate_text_summaries(texts, tables, summarize_texts=True):

    prompt_text = """
    You are an advanced AI assistant tasked with summarizing tables and text␣
 ↪for optimized retrieval in structured databases or search systems.
    Your goal is to create a detailed yet concise summary that maximizes the␣
 ↪relevance and accuracy of embedding-based retrieval.
    When summarizing tables or text, include the following details:

    1. **For Tables**:
        - **Headers and Structure**: Summarize the table structure, listing all␣
 ↪headers and describing what each column represents.
        - **Key Data Points**: Highlight important values, trends, or outliers␣
 ↪within the table.
        - **Relationships and Patterns**: Identify notable relationships␣
 ↪between rows and columns.
        - **Contextual Information**: Explain the overall purpose of the table␣
 ↪and its type of data.

    2. **For Text**:
        - **Main Themes and Key Points**: Provide a concise summary of the␣
 ↪central theme or argument.
        - **Important Keywords**: Include relevant keywords or phrases for␣
 ↪retrieval.
        - **Context and Purpose**: Describe the purpose of the text and its␣
 ↪audience.
```

```
    3. **Optimization for Retrieval**:
        - Ensure the summary emphasizes distinctive and searchable features for␣
↪embedding-based search.
    Table or text: {element}
    """

    prompt = ChatPromptTemplate.from_template(prompt_text)

    # Text summary chain
    model = ChatOpenAI(temperature=0, model="gpt-4")
    summarize_chain = {"element": lambda x: x} | prompt | model |␣
↪StrOutputParser()

    # Initialize empty summaries
    text_summaries = []
    table_summaries = []

    # Apply to text if texts are provided and summarization is requested
    if texts and summarize_texts:
        text_summaries = summarize_chain.batch(texts, {"max_concurrency": 3})
    elif texts:
        text_summaries = texts

    # Apply to tables if tables are provided
    if tables:
        table_summaries = summarize_chain.batch(tables, {"max_concurrency": 3})

    return text_summaries, table_summaries
```

Key Explanation:

Batch Processing:

The summarize_chain.batch method processes multiple items (texts/tables) concurrently with max_concurrency set to 3.

This improves efficiency for larger datasets.

Inputs:

texts: List of text segments to summarize.

tables: List of table contents to summarize.

Outputs:

Returns text_summaries and table_summaries, which are concise summaries of the provided content.

[ ]:
```
# 3. Adding Documents to Vector Store
```

```python
# Each summary and its metadata are encapsulated as a Document object and␣
 ↪stored in OpenSearch.

def create_documents_from_summaries(
    text_summaries,
    texts_list,
    text_metadata,
    table_summaries,
    table_list,
    table_metadata,
    image_summaries,
    img_base64_list,
    image_metadata
):
    """
    Generate Document objects from summaries and original data.
    """
    documents = []

    # Helper function to create Document objects
    def create_docs(summaries, contents, metadata_list):
        for i, summary in enumerate(summaries):
            metadata = metadata_list[i] if metadata_list else {}
            doc = Document(page_content=summary, metadata=metadata)
            documents.append(doc)

    # Add text documents
    if text_summaries:
        create_docs(text_summaries, texts_list, text_metadata)

    # Add table documents
    if table_summaries:
        create_docs(table_summaries, table_list, table_metadata)

    # Add image documents
    if image_summaries:
        create_docs(image_summaries, img_base64_list, image_metadata)

    return documents


# Create vector store and add documents
documents = create_documents_from_summaries(
    text_summaries,
    texts_list,
    text_metadata,
    table_summaries,
```

```
    table_list,
    table_metadata,
    image_summaries,
    img_base64_list,
    image_metadata
)

# Initialize OpenSearch Vector Store and add documents
vectorstore = OpenSearchVectorSearch.from_documents(
    documents=documents,
    embedding=embeddings,
    opensearch_url=OPENSEARCH_URL,
    http_auth=awsauth,
    use_ssl=True,
    verify_certs=True,
    connection_class=RequestsHttpConnection,
    index_name=index_name,
    timeout=300,
    max_retries=3,
    retry_on_timeout=True
)
```

Key Steps in the Code:

2.0.2 Create Documents:

The create_documents_from_summaries function takes:

Summaries (text_summaries, table_summaries, image_summaries).

Original contents (texts_list, table_list, img_base64_list).

Associated metadata.

Each summary and its metadata is encapsulated as a Document object.

2.0.3 Add to Vector Store:

The OpenSearchVectorSearch.from_documents method is used to ingest the created Document objects into OpenSearch.

This includes embedding the documents using OpenAIEmbeddings and optimizing them for vector-based retrieval.

###Integration with OpenSearch:

Documents are stored in the specified OpenSearch index (index_name) with parameters for retry, timeout, and SSL verification.

3 4. Query and Retrieve:

a. Create a simple API or script that allows querying based on a given text prompt. The query should retrieve similar embeddings from the vector store and return the corresponding records from the database.

b. Include a use case for Retriever-Augmented Generation (RAG), where the retrieved data is used to generate a summary or response based on the query.

[ ]:
```
# Main API route
@app.post("/SFRAG/retrieval")
async def multi_modal_query(query_request: QueryRequest = Body(...), request:␣
↪Request = None):
    # Retrieve documents relevant to the user query
    vectorstore = get_vectorstore(query_request.index_name)
    retriever = vectorstore.as_retriever(search_kwargs={"k": 5})

    # Since retriever.get_relevant_documents might be synchronous, run it in a␣
↪thread
    retrieved_docs = await asyncio.to_thread(retriever.get_relevant_documents,␣
↪query_request.user_query)
    logger.info(f"Retrieved Docs: {retrieved_docs}")

    # Preprocess metadata for citation
    processed_metadata = preprocess_metadata(retrieved_docs)

    # Split image and text content
    context_data = split_image_text_types(retrieved_docs)

    # Prepare the messages
    messages = img_prompt_func({"context": context_data, "question":␣
↪query_request.user_query})

    # Generate response and get citations asynchronously
    async def generate_response():
        callback = AsyncIteratorCallbackHandler()
        model = ChatOpenAI(temperature=0, streaming=True,␣
↪model_name="gpt-4-vision-preview", callbacks=[callback])

        # Start the LLM generation task
        llm_task = asyncio.create_task(model.agenerate(messages=[messages]))

        # Start the citations task
        citations_task = asyncio.create_task(get_citations(query_request.
↪user_query, retrieved_docs, processed_metadata))

        buffer = ""
        try:
```

```
            async for chunk in callback.aiter():
                buffer += chunk
                # Send data when buffer reaches certain size or contains a␣
↪newline
                if len(buffer) > 50 or '\n' in buffer:
                    yield f"{buffer}"
                    buffer = ""
            # Send any remaining data in the buffer
            if buffer:
                yield f"{buffer}"
        except Exception as e:
            logger.error(f"Caught exception: {e}")
        finally:
            callback.done.set()
        # Wait for the LLM task to complete
        await llm_task
        # Wait for citations to be ready
        citations = await citations_task
        # After streaming the response, yield the citations
        yield f"\n<<CITATIONS_START>>{json.dumps(citations)}<<CITATIONS_END>>\n"

    # Determine if the client can accept streaming responses
    accept_header = request.headers.get('accept', '')
    user_agent = request.headers.get('user-agent', '').lower() if request else␣
↪''
    if 'text/event-stream' in accept_header or 'curl' in user_agent:
        # Return streaming response
        return StreamingResponse(generate_response(), media_type="text/plain")
    else:
        # Non-streaming response
        response_text = ""
        async for chunk in generate_response():
            response_text += chunk

        if "<<CITATIONS_START>>" in response_text and "<<CITATIONS_END>>" in␣
↪response_text:
            response_body, citations_part = response_text.
↪split("<<CITATIONS_START>>", 1)
            citations_json, _ = citations_part.split("<<CITATIONS_END>>", 1)
            citations_json = citations_json.strip()
            try:
                citations = json.loads(citations_json)
            except json.JSONDecodeError:
                logger.error("Error parsing citations JSON.")
                citations = []
        else:
            response_body = response_text
```

```
            citations = []

        # Build the response structure
        full_response = {
            "response": {
                "content": response_body.strip()
            },
            "citation": citations
        }
        return JSONResponse(content=full_response)
```

3.0.1 Query and Retrieve

Uses the multi_modal_query function.

Retrieves similar embeddings from the vector store using retriever.

get_relevant_documents.

Formats the results with content and metadata.

3.0.2 Retriever-Augmented Generation

The same multi_modal_query function handles RAG.

Retrieved documents are formatted and passed as context to a GPT model (ChatOpenAI).

The GPT model generates a summary or response using the context.

4 5. Documentation:

a. The candidate should document their code, the thought process behind their design choices, and any trade-offs they considered (e.g., schema design, vector storage approach, etc.).

4.0.1 Thought Process and Design Choices

4.0.2 API Framework - FastAPI:

Reason: FastAPI is chosen for its asynchronous capabilities, which are crucial when handling multiple I/O-bound operations like interacting with OpenSearch, Azure Blob Storage, and calling APIs.

Trade-off: Requires familiarity with Python async patterns, which can add complexity but offers significant performance benefits.

4.0.3 Vector Storage - OpenSearch:

Reason: OpenSearch is used for storing and retrieving high-dimensional vectors. It provides efficient nearest-neighbor search with k-NN and supports scalable indexing.

Trade-off: Configuration and maintenance of OpenSearch can be challenging, especially for large-scale datasets.

4.0.4 Embedding Function - OpenAI Embeddings:

Reason: The embeddings are pre-trained and optimized for semantic similarity, making them suitable for RAG (Retriever-Augmented Generation) use cases.

Trade-off: Dependency on OpenAI API and associated costs.

Multi-Modal Retrieval:

Design: The code supports both text and images by separating image-based content and text-based content from documents.

Reason: Enables a unified approach for querying across different content types.

Trade-off: More complex preprocessing and metadata handling are required to manage the dual modality.

[ ]:
```
# Index Creation (create_index_if_not_exists):

# Purpose: Ensures an OpenSearch index is created if it doesn't already exist, with configuration for k-NN vector search.

# Key Design:
# Configures the vector field with knn_vector type for high-dimensional data.
# Supports L2 distance metric for similarity searches.
# Trade-off: Requires upfront knowledge of vector dimensions and OpenSearch schema.


#Initialize OpenSearch and Create Index (if not exists)
def create_index_if_not_exists(index_name):
    # AWS authentication for OpenSearch
    awsauth = AWS4Auth(AWS_ACCESS_KEY, AWS_SECRET_KEY, AWS_REGION, AWS_SERVICE)
    opensearch_host = OPENSEARCH_URL.split("//")[-1].split(":")[0]  # Extract host without protocol and port

    # Initialize OpenSearch client
    opensearch_client = OpenSearch(
        hosts=[{'host': opensearch_host, 'port': 443}],
        http_auth=awsauth,
        use_ssl=True,
        verify_certs=True,
        connection_class=RequestsHttpConnection,
        timeout=300,  # Increased timeout to 300 seconds
        max_retries=3,
        retry_on_timeout=True
    )

    # Define vector field mapping
    vector_field = "vector_field"
```

```
    dim = 1536  # Specify the dimensions of your vectors
    index_mapping = {
        "settings": {"index": {"knn": True, "knn.algo_param.ef_search": 512}},
        "mappings": {
            "properties": {
                vector_field: {
                    "type": "knn_vector",
                    "dimension": dim,
                    "method": {
                        "name": "hnsw",
                        "space_type": "l2",
                        "engine": "nmslib",
                        "parameters": {"ef_construction": 512, "m": 16},
                    },
                }
            }
        },
    }

    # Create the index if it does not exist
    if not opensearch_client.indices.exists(index=index_name):
        opensearch_client.indices.create(index=index_name, body=index_mapping)
        print(f"Index '{index_name}' created successfully.")
    else:
        print(f"Index '{index_name}' already exists.")

    # Verify the mapping
    mapping = opensearch_client.indices.get_mapping(index=index_name)
    print("Index mapping:", mapping)
```

[ ]:
```
# Document Storage and Metadata Handling:

# Purpose: Manages storage of image, table, and text data as structured␣
 ↪metadata for efficient retrieval.
# Key Design:
# Separates content types (TEXT, TABLE, FIGURE) into dedicated metadata␣
 ↪structures.
# Preprocesses metadata by parsing and standardizing fields like URLs and page␣
 ↪numbers.
# Trade-off: Adds complexity to metadata extraction and storage logic.

# Preprocess metadata synchronously
def preprocess_metadata(chunks):
    preprocessed_data = []
    for chunk in chunks:
        metadata = chunk.metadata.copy()
        if 'doc_id' in metadata:
```

```
            del metadata['doc_id']
        if 'url' in metadata and isinstance(metadata['url'], str):
            metadata['url'] = metadata['url'].split(',')
        if 'page_numbers' in metadata and isinstance(metadata['page_numbers'],␣
 ↪str):
            metadata['page_numbers'] = metadata['page_numbers'].split(',')
        preprocessed_data.append(metadata)
    return preprocessed_data
```

[ ]:
```
# Query and Retrieval (multi_modal_query):

# Purpose: Handles user queries by retrieving similar embeddings and preparing␣
 ↪relevant context for response generation.
# Key Design:
# Integrates vector retrieval with context assembly for multi-modal inputs.
# Uses retriever.get_relevant_documents for fetching top-k matches.


# Main API route
@app.post("/SFRAG/retrieval")
async def multi_modal_query(query_request: QueryRequest = Body(...), request:␣
 ↪Request = None):
    # Retrieve documents relevant to the user query
    vectorstore = get_vectorstore(query_request.index_name)
    retriever = vectorstore.as_retriever(search_kwargs={"k": 5})

    # Since retriever.get_relevant_documents might be synchronous, run it in a␣
 ↪thread
    retrieved_docs = await asyncio.to_thread(retriever.get_relevant_documents,␣
 ↪query_request.user_query)
```

[ ]:
```
# Multi-Modal Context Preparation (split_image_text_types):

# Purpose: Separates text content and base64-encoded images for multi-modal␣
 ↪processing.
# Key Design:
# Handles image resizing and validation for embedding optimization.

# Separate images and texts
def split_image_text_types(docs):
    b64_images = []
    texts = []
    for doc in docs:
        content = doc.page_content if isinstance(doc, Document) else doc
        if looks_like_base64(content) and is_image_data(content):
            content = resize_base64_image(content, size=(1300, 600))
```

```
            b64_images.append(content)
        else:
            texts.append(content)
    return {"images": b64_images, "texts": texts}
```

5 6. Bonus:

a. Implement monitoring or logging for the data pipeline to track the data flow and identify potential bottlenecks.

b. Optimize the pipeline for scalability, such as handling larger files or parallel processing.

[ ]:
```
# Monitoring and Logging
# The provided code implements a robust logging mechanism, which helps monitor␣
 ↪the data pipeline. Below are the features and their relevance:

# Logging Configuration:

# The logging module is configured at the INFO level to capture essential␣
 ↪events, including process initiation, execution flow, and exceptions.

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)
```

[ ]:
```
# Event Tracking:

# Logs are placed strategically to track major steps:
# File upload: "Uploading PDF: {pdf_path}"
# API request-response cycle: "POST {DOCUMENT_EXTRACTOR_API_URL}" and "Received␣
 ↪response with status code: {response.status_code}"
# Data extraction and processing: "Extracted data saved to {json_path}"
# These logs help identify bottlenecks in API calls, file handling, and␣
 ↪metadata extraction.


# Log the request details
logger.info(f"Uploading PDF: {pdf_path}")
logger.info(f"POST {DOCUMENT_EXTRACTOR_API_URL}")
logger.info(f"Payload: {payload}")
logger.info(f"Headers: {headers}")
logger.info(f"Received response with status code: {response.status_code}")
logger.info(f"Extracted data saved to {json_path}")
```

[ ]:
```
# Error Handling:

# Error messages are logged for issues like timeouts, JSON decoding errors, and␣
 ↪file I/O failures:
```

```
logger.error(f"HTTP error occurred: {http_err}")
raise Exception(f"Failed to extract data: HTTP error occurred: {http_err}")
```

Parallel Processing:

The concurrent.futures.ThreadPoolExecutor is utilized for parallelizing tasks like file downloads and metadata processing.

[ ]:
```
# Retry Mechanisms:

# OpenSearch client configuration includes retries for fault tolerance:

max_retries=3,
retry_on_timeout=True
```

6 API Calling using Swagger UI

Input: Parameters:

index_name: A required string parameter where the user specifies the index name. For example, the index_name provided is shyamadhikari.

file: A required file parameter where the user uploads a PDF document. The file uploaded in the example is named Quarterly_Flint_Blood_Lead_Report-Q1-2022.pdf, which contains tables, images, and text.

Request Body:

The file and index_name are sent to the API as a multipart/form-data request.

Request Method:

A POST request is made to the endpoint /SFRAG/ingest.

Output:

Status Code: 200:

Indicates that the request was successfully processed.

Response Body:

“status”: “Index ingested successfully”: Confirms that the data from the PDF document was successfully processed and ingested into the specified index.

“index_name”: “shyamadhikari”: Echoes the index name provided in the input.

“imagecount”: 5: Indicates that 5 images were detected and processed from the uploaded PDF.

“tablecount”: 5: Indicates that 5 tables were detected and processed from the uploaded PDF.

Summary:

Input: The user provides an index name (shyamadhikari) and uploads a PDF file with tables, images, and text.

Output: The API processes the file, extracts content (images, tables, and text), and returns a summary of the number of images and tables ingested, along with the success status.

This process demonstrates the capability of the application to handle multimodal content (text, tables, and images) from a PDF file and index it for further use.

Query Retrieval:

Once the index was created, the query endpoint /SFRAG/retrieval was used to retrieve specific information from the document by passing:

Input:

user_query: "What is the purpose of this report?"

index_name: "shyamadhikari" (to refer to the previously created index).

session_id: A session identifier (17021) for conversational tracking.

Output:

The system returned a response:

Content: "The purpose of this report is to track blood lead level test results in Flint, Michigan."

Citations: Provided metadata pointing to the relevant sections of the document (e.g., a table from page 5 and text from page 1). This included:

type: Type of the cited content (e.g., "TABLE", "TEXT").

url: Links to the extracted images of the table and text stored in Azure Blob Storage.

page_numbers: Location of the content within the document.

filename: The name of the original document for reference.

Key Observations:

The retrieval system was able to:

Interpret the user query: It identified the intent behind the question.

Locate relevant content: Using embeddings and metadata, the system fetched the exact text and table that answered the query.

Cite specific sections: Provided structured references for further validation (e.g., links and page numbers).

Benefits of Conversational History:

If the session ID (17021) was previously used for querying the same document, the system could use past interactions to refine its results. For instance: If earlier queries had already identified the document's structure or context, the system could skip redundant processing steps. In a

conversational flow, this ensures that the user receives contextually relevant answers based on prior interactions.

Small Summary from that output :

The query "What is the purpose of this report?" was submitted, and the system retrieved a detailed response: "The purpose of this report is to track blood lead level test results in Flint, Michigan."

The response also included links to the extracted text and table data stored in Azure Blob Storage. These files contain visual data like charts and tabular summaries of blood lead test results.

Key Content:

The table retrieved from blob storage presents data on blood lead levels over several years for Michigan, Genesee County, and Flint zip codes. It includes the number of adults tested, those with blood lead levels (BLL) 5 g/dL, and the corresponding percentages.

This table helps track trends in lead exposure and assess the impact of mitigation measures in these regions over the years. :

7 Overall Summary

This FastAPI application facilitates the ingestion and processing of multimodal data, such as PDF files, for retrieval-augmented generation (RAG). It integrates tools like OpenSearch for vector search, Azure Blob Storage for managing file storage, and OpenAI's GPT models for generating embeddings and summarizations. The main API endpoint, /SFRAG/retrieval, accepts PDF files and processes their contents by extracting text, tables, and images. Extracted data is summarized and indexed into a vector database to enable efficient semantic search and retrieval. The application uses OpenSearchVectorSearch for managing the vectorized embeddings of documents and employs LangChain modules for splitting and structuring text data for embedding generation. Summaries are prepared for extracted text, tables, and images to optimize their use in retrieval systems, emphasizing searchable and meaningful content.

The application's functionality is highly modular, covering multiple tasks like validating and uploading files, extracting structured data from PDFs, and summarizing data using AI models like GPT. For images, it identifies and resizes base64-encoded images, while for tables and text, it generates detailed summaries that include structural and contextual information. Metadata associated with each data type is also enriched for efficient search and retrieval. The application further integrates AWS OpenSearch Service for indexing and storing data while maintaining robust configurations for SSL, retries, and timeouts. Additionally, it provides comprehensive error handling for operations like file uploads, data extraction, and indexing. The entire system is designed to deliver a seamless and efficient pipeline for processing and retrieving multimodal data in enterprise applications.

[ ]:

Parameters Cancel Reset

No parameters

Request body required multipart/form-data

index_name * required
`string`
shyamadhikari

file * required
`string($binary)`
Choose File Quarterly_Flint_Blood…ad_Report-Q1-2022.pdf

Execute Clear

Responses

Curl

```
curl -X 'POST' \
  'http://172.174.161.97:8100/SFRAG/ingest' \
  -H 'accept: application/json' \
  -H 'Content-Type: multipart/form-data' \
  -F 'index_name=shyamadhikari' \
  -F 'file=@Quarterly_Flint_Blood_Lead_Report-Q1-2022.pdf;type=application/pdf'
```

Request URL

```
http://172.174.161.97:8100/SFRAG/ingest
```

Server response

Code	Details
200	Response body

```
{
  "status": "Index ingested successfully",
  "index_name": "shyamadhikari",
  "imagecount": 5,
  "tablecount": 5
}
```

Download

POST `/SFRAG/retrieval` Multi Modal Query

Parameters | Cancel | Reset

No parameters

Request body required | application/json

```
{
  "user_query": "What is the purpose of this report?",
  "index_name": "shyamadhikari",
  "session_id": "17021"
}
```

Execute | Clear

Responses

Curl

```
curl -X 'POST' \
  'http://172.174.161.97:8100/SFRAG/retrieval' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "user_query": "What is the purpose of this report?",
  "index_name": "shyamadhikari",
  "session_id": "17021"
}'
```

Request URL

```
http://172.174.161.97:8100/SFRAG/retrieval
```

Server response

Code	Details
200	Response body


```
{
  "response": {
    "content": "The purpose of this report is to track blood lead level test results in Flint, Michigan."
  },
  "citation": [
    {
      "type": "TABLE",
      "url": [
        "https://experimentsstorage1.blob.core.windows.net/extractedcontent/Quarterly_Flint_Blood_Lead_Report-Q1-2022_1_TABLE_5_1_20241203180004.png"
      ],
      "page_numbers": [
        "5"
      ],
      "filename": "Quarterly_Flint_Blood_Lead_Report-Q1-2022.pdf",
      "pdf_url": "https://experimentsstorage1.blob.core.windows.net/sf-bot/Project/shyamadhikari/Quarterly_Flint_Blood_Lead_Report-Q1-2022.pdf"
    },
    {
      "type": "TEXT",
      "url": [
        "https://experimentsstorage1.blob.core.windows.net/extractedcontent/Quarterly_Flint_Blood_Lead_Report-Q1-2022_1_1_draw_bbox_20241203175949.png"
      ],
      "page_numbers": [
        "1"
      ],
      "filename": "Quarterly_Flint_Blood_Lead_Report-Q1-2022.pdf",
      "pdf_url": "https://experimentsstorage1.blob.core.windows.net/sf-bot/Project/shyamadhikari/Quarterly_Flint_Blood_Lead_Report-Q1-2022.pdf"
    }
  ]
```


Download

Blood Lead Level Test Results for Selected Flint Zip Codes, Genesee County, and the State of Michigan

Executive Summary for Quarter 1, Calendar Year 2022

Data as of July 11, 2022

This report is generated by MDHHS to track blood lead level test results in Flint, Michigan.

Background

- Blood lead level testing is an important part of efforts to identify people who have been harmed by drinking water that contained lead and from other sources of lead exposure in the home. MDHHS recognizes that the full community of Flint must be the focus of the public health response.
- The CDC announced a change in the blood lead reference value (BLRV) from 5 µg/dL (micrograms lead per deciliter of blood) to 3.5 µg/dL in October 2021. MDHHS adopted the new reference value on May 1, 2022 (for more information see https://www.michigan.gov/mdhhs/inside-mdhhs/newsroom/2022/04/29/mdhhs-updating-definition-of-elevated-blood-lead-levels). Reports beginning in Quarter 2 of 2022 will include counts of blood lead levels (BLL) above the BLRV using both 3.5 µg/dL and 5 µg/dL as the blood lead reference values.

Data Presentation

- Each person is included only once per time period in this table.
  - If a person had multiple tests in the time period, the highest blood lead level from a venous test was counted.
  - If no venous test was performed, the highest blood lead level from a capillary test was counted.
  - If the type of test was unknown, the highest blood lead level obtained from an unknown sample type was counted.

Summary of Findings

- Between 10/1/2015 and 03/30/2022, 46,479 people had a blood lead test in the City of Flint in zip codes 48501-48507.
- Since 10/1/2015, 399 children under age 18 in Flint in zip codes 48501-48507 had blood lead levels > BLRV: blood lead levels greater than or equal to 5 µg/dL (micrograms lead per deciliter of blood).
- Of the 13,998 children younger than 6 years old tested since 10/1/2015, 2.5% had blood lead levels greater than or equal to 5 µg/dL.

Year	Michigan			Genesee County			Flint in ZIPs 48	
	# Adults tested for lead	# with BLL ≥ 5 µg/dL[1]	% with BLL ≥ 5 µg/dL[1]	# Adults tested for lead	# with BLL ≥ 5 µg/dL[1]	% with BLL ≥ 5 µg/dL[1]	# Adults tested for lead	# with ≥ 5 µg/
2014	10,761	1,182	11.0	437	44	10.1	112	
2015	11,808	1,137	9.6	1,250	48	3.8	810	
2016	34,526	1,371	4.0	20,219	305	1.5	15,832	
2017	18,212	1,123	6.2	5,363	90	1.7	3,227	
2018	18,587	980	5.3	4,669	54	1.2	2,536	
2019	17,459	1,059	6.1	3,675	55	1.5	1,954	
2020	13,754	897	6.5	1,948	28	1.4	1,050	
2021	15,236	877	5.8	1,607	34	2.1	880	
2022*	2,657	187	7.0	197	8	4.1	109	