Optimized Retrieval-Augmented Generation (RAG) Pipeline with Code Highlig

This document provides an overview of the Optimized Retrieval-Augmented Generation (RAG) pipeline
with code examples for each section. Code is highlighted to make it easier to read and follow,
showing how the system interacts with semi-structured PDF data to generate responses based on user queries.

1. Data Ingestion & Embedding

- Preprocessing: Extract text from PDFs and filter irrelevant content.
- Dynamic Chunking: Text is split into manageable chunks for embedding.
- Embedding: Chunks are converted into vector embeddings and stored.

Below is the code to handle the data ingestion and embedding process:

```
from langchain.text_splitter import RecursiveCharacterTextSplitter


@app.route('/upload', methods=['POST'])
def upload_and_embed():
    global vector_db
    pdf_text = request.json.get('pdf_text')

    # Preprocess: Filter irrelevant content
    pdf_text = pdf_text.replace("\n", " ").strip()
```

```
    # Dynamic chunking
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
    chunks = text_splitter.split_text(pdf_text)

    # Add metadata like page numbers (assume each chunk has metadata)
    documents = [{"page_content": chunk, "metadata": {"page": idx + 1}} for idx, chunk
in enumerate(chunks)]

    # Embed and store chunks
    vector_db = FAISS.from_documents(documents, OpenAIEmbeddings())

    return jsonify({"message": "PDF embedded successfully", "chunks_count":
len(chunks)})
```

2. Query Handling & Optimization

- Hybrid Search: Combines keyword search and vector similarity search.

- Query Embedding: Converts the query into a vector for better retrieval.

Below is the code to handle user queries with optimizations:

```
from langchain.chains import ConversationalRetrievalChain

@app.route('/query', methods=['POST'])
def handle_query():
    global vector_db
    if not vector_db:
```

```
        return jsonify({"error": "No PDF data uploaded"}), 400


    user_query = request.json.get('query')


    # Hybrid retriever
    retriever = vector_db.as_retriever(search_type="similarity", search_kwargs={"k": 5})


    # Use ConversationalRetrievalChain for better query handling
    chain = ConversationalRetrievalChain.from_llm(
        OpenAI(model="gpt-4"), retriever=retriever, return_source_documents=True
    )
    response = chain.run(user_query)


    return jsonify({"response": response['answer'], "sources":
response['source_documents']})
```

3. Comparison Queries

- Extract relevant data fields from multiple PDFs.

- Compare extracted data and display results.

Below is the code to handle comparison queries:

```
@app.route('/compare', methods=['POST'])
def handle_comparison():
    global vector_db
    if not vector_db:
```

```
        return jsonify({"error": "No PDF data uploaded"}), 400


    comparison_query = request.json.get('query')
    results = vector_db.similarity_search(comparison_query, k=10)


    # Extract key data points for comparison
    comparison_data = []
    for result in results:
        page_content = result.page_content
        # Example: Regex for extracting numerical comparisons
        degree_data = re.findall(r"(Bachelor's|Master's|PhD).*?(\d+\.\d+%)",
page_content)
        if degree_data:
            comparison_data.extend(degree_data)


    return jsonify({"comparison_data": comparison_data})
```

4. LLM Integration & Optimizations

- Contextual Prompts: Add retrieved data to the model's input prompt for accurate responses.

- Model Integration: Use of GPT models for response generation.

Below is the code to integrate LLMs into the pipeline:

```
from langchain.prompts import PromptTemplate


@app.route('/query', methods=['POST'])
```

```
def handle_query():
    user_query = request.json.get('query')

    # Retrieve top-k chunks
    retrieved_docs = vector_db.similarity_search(user_query, k=5)

    # Prepare a dynamic context prompt
    context = "
".join([doc.page_content for doc in retrieved_docs])
    prompt_template = PromptTemplate(
        input_variables=["context", "question"],
        template="You are an assistant. Context:
{context}

Question:
{question}"
    )
    final_prompt = prompt_template.format(context=context, question=user_query)

    response = llm(final_prompt)  # Use OpenAI or other LLM

    return jsonify({"response": response})
```