**CISC 691:Detailed Report: A Simple RAG: Design & Implementation**

This report presents an innovative project which covers the entire process of designing, implementing, and operating a Retrieval-Augmented Generation (RAG) pipeline. Through the combination of document ingestion and embedding generation alongside vector storage and retrieval capabilities with OpenAI's GPT-4 for response generation this pipeline has the potential to transform artificial intelligence and machine learning technologies. The sections below outline the system architecture along with its core components and discuss how testing and optimization strategies were applied to evaluate overall system performance based on the code developed for this assignment.

**Part 1: Design and Plan**

**Review of RAG Architecture:**

By integrating pertinent domain-specific information from external documents, the RAG system improves the output quality of language models such as GPT-4. The implemented RAG system has five main components including document ingestion, embedding generation, vector storage, query retrieval, and response generation. The document ingestion process consists of preprocessing raw documents to generate embeddings which transform text into vector embeddings that are stored in ChromaDB for similar search efficiency. The query retrieval step finds document chunks that best match the user's query, and the response generation step uses GPT-4 to create answers tailored to the context.

**Define System Requirements:**

The system processes multiple document formats such as plain text and CSV files through tokenization and chunking to maintain text within acceptable model limits. ChromaDB acts as the vector database which stores vectors and indexes embeddings to enable swift retrieval. FastAPI enables the creation of system endpoints which users can access to request document chunks and GPT-4 generated responses. GPT-4 generates enhanced responses by merging retrieved document context with user queries.

**Plan Implementation Strategy:**

The text undergoes tokenization and segmentation into smaller parts when embedding creation requires it. The text is converted into vectors by the embedding model which are then stored in ChromaDB and used by the system to execute similarity searches for relevant document chunks. The implementation of a clearly defined prompt engineering strategy follows these steps. The strategy integrates the retrieved context with user queries to create optimized input for GPT-4 which enables the production of relevant domain-specific responses.

**Part 2: Build and Test**

**Set Up Development Environment**:

The development environment setup encompasses Python as its main language along with dependencies such as the OpenAI API for GPT-4 operations, FastAPI for managing APIs, ChromaDB for storing vector data, and Ollama to connect with the language model. The available tools enhance pipeline step execution for seamless integration and functional operation.

**Implement Core Components:**

The primary components consist of document ingestion which involves reading raw text and preparing it through tokenization and chunking processes. The Embedding Preparer translates text into vector embedding by utilizing a pre-trained model such as BERT. The Embedding Loader saves these vector embeddings into ChromaDB so they can be retrieved later. The retriever component accesses ChromaDB to obtain document chunks that best match the user's search terms. The RAG Query Processor utilizes the documents it retrieves to build a prompt for GPT-4 answer generation.

**Develop RAG Pipeline:**

The full pipeline is implemented in five steps: The complete framework requires document ingestion followed by embedding creation and vector storage before retrieval and response production. Documents undergo ingestion and tokenization before being transformed into embeddings which are then stored in ChromaDB. The system retrieves relevant document chunks to deliver to GPT-4 when it receives a query for response generation. The pipeline's modular structure allows for independent optimization and testing of each individual step.

**Create Testing Framework**:

We have established a rigorous testing process to ensure users can trust our system's reliability. Unit tests confirm that every system part including ingestion processes embedding generation and vector storage operations operate effectively. Integration tests confirm that the full pipeline runs without issues from the initial document ingestion to the final response generation. Through performance benchmarks we measure system efficiency by evaluating latency and retrieval accuracy levels.

**Part 3: Optimize, Document, and Present**

**Optimize System Performance:**

The optimization of system performance includes tuning the chunking parameters and embedding settings together with vector search configuration to enhance both retrieval speed and response quality. Properly configuring document processing parameters through chunk size and overlap adjustments results in efficiency gains while ChromaDB optimization accelerates similarity searches. The process of prompt engineering undergoes iterative refinement to enhance response relevance and coherence. The implementation of caching mechanisms serves to prevent unnecessary computations while also minimizing latency.

**Evaluation and Benchmark:**

Performance evaluation of the system requires measurement of response latency together with retrieval accuracy and the quality of responses from GPT-4. The system monitors latency across the entire pipeline starting from document ingestion up to response generation. The accuracy of retrieval operations is determined by examining the alignment between retrieved document segments and user search queries. The RAG pipeline's response quality assessment involves comparing generated responses to GPT-4 baseline responses to guarantee better relevance and information depth.

**Document System Architecture:**

A detailed diagram documents the system architecture by showing the process flowing from document ingestion to response generation. API specifications outline system interaction endpoints and deployment instructions provide setup configuration guidance. The maintenance guidelines detail procedures for system updates as well as adding new documents and vector database maintenance to maintain optimal performance.

**Explanation of the code in brief and its functionality:**

**Detailed Explanation of the Code**

This Python code is designed to build and run a Retrieval-Augmented Generation (RAG) pipeline for document processing and generating responses from OpenAI's GPT-4. The step-by-step explanation of each component and its functionality:

**Imports and Setup**

* **Standard Libraries**: The code imports several standard libraries such as os, logging, argparse, csv, and io for file handling, logging, command-line argument parsing, and CSV processing.
* **Third-Party Libraries**:
  + **OpenAI**: This library is used to interact with OpenAI's API to generate responses using GPT-4.
  + **transformers**: This is from Hugging Face and provides the AutoTokenizer class used for tokenizing text.
  + **Pathlib**: This module helps manage filesystem paths in an OS-independent way.
  + **datetime**: Used for generating timestamps for logging.
* **Custom Modules**: Several custom modules (such as ConfigManager, DocumentIngestor, EmbeddingPreparer, etc.) are imported to handle specific tasks like configuration management, document ingestion, embedding preparation, and querying.

**Configuration Setup**

* **CONFIG\_FILE**: This is the path to the configuration file that contains various settings like model names, directories, etc.
* **ConfigManager**: This class is used to load and manage configuration settings from the specified config.json file.
* **OpenAI API Key**: The openai.api\_key is set to authenticate the requests made to OpenAI's GPT-4.

**Directory Setup**

* **raw\_input\_directory**: Specifies the directory where raw documents are stored.
* **cleaned\_text\_directory**: This is where cleaned text files (after preprocessing) will be stored.
* **embeddings\_directory**: Specifies the directory to store generated embeddings.
* **vectordb\_directory**: Directory for storing the vector database (ChromaDB).
* **collection\_name**: The collection name used for storing the embeddings in ChromaDB.

A screenshot of a computer program

AI-generated content may be incorrect.

**Functions**

**setup\_logging(log\_level)**

This function configures logging for the RAG pipeline, both to the console and to a file. It:

* Creates a "logs" directory to store log files.
* Sets the log level based on the log\_level parameter (defaults to DEBUG).
* Logs messages to both console and a timestamped file.
* Configure specific logging levels for the OpenAI and ChromaDB libraries to reduce verbosity.

**ensure\_directories\_exist(config)**

This function ensures that the necessary directories for input, output, embeddings, and the vector database exist. If they don't, it creates them. It uses the configuration loaded by the ConfigManager to check the paths.

A black screen with colorful text

AI-generated content may be incorrect.–

**step01\_ingest\_documents(args)**

This is the first step in the pipeline, responsible for reading and preprocessing the documents. It:

* Takes a list of input files (either a single file or all files from the directory).
* Initializes a tokenizer using the model specified in the configuration (defaults to bert-base-uncased).
* Splits documents into chunks if they exceed the model's token length limit (e.g., 512 tokens).
* Preprocesses CSV files by detecting the delimiter and converting each row into a structured text format.
* Writes the cleaned text (or chunks) to the specified cleaned\_text\_directory.

A screen shot of a computer

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

A black screen with text on it

AI-generated content may be incorrect.

**step02\_generate\_embeddings(args)**

This function is responsible for generating embeddings from the cleaned text files. It:

* Creates embeddings using the specified model.
* Call the EmbeddingPreparer class to process each file and save the embeddings to the embeddings\_directory.

A black background with colorful text

AI-generated content may be incorrect.

**step03\_store\_vectors(args)**

In this step, the embeddings generated in Step 2 are stored in a vector database (ChromaDB). It:

* Deletes any existing vector database (if necessary).
* Loads the embedding from the embedding\_directory and stores them in ChromaDB using the EmbeddingLoader class.
* Ensures that the vector database is ready for querying.

A black background with colorful text

AI-generated content may be incorrect.

**step04\_retrieve\_relevant\_chunks(args)**

This function performs the retrieval of relevant text chunks based on a user's query. It:

* Initializes a ChromaDBRetriever to query the vector database.
* Performs a search to retrieve the top-k relevant document chunks based on the query.
* Logs the results, including document IDs, scores, and context.
* This step essentially provides the context needed for GPT-4 to generate an accurate and relevant response.

A black screen with colorful text

AI-generated content may be incorrect.

**generate\_response\_with\_openai(query\_args)**

This helper function is intended to generate a response from OpenAI’s GPT-4 based on the context provided. However, this function is not used directly in the current implementation.



**RAGQueryProcessor Class**

This class handles the process of combining retrieved context with a user's question and generating a response using GPT-4:

* **\_\_init\_\_(self, retriever, llm\_client)**: Initializes the processor with a retriever (for querying the vector database) and a language model client (OpenAI's GPT-4).
* **query(self, question)**: Takes a question, retrieves relevant document chunks from the vector database, and constructs a final prompt combining the context with the question. It then calls OpenAI’s API to generate a response using the GPT-4 model.

A black screen with colorful text

AI-generated content may be incorrect.

**step05\_generate\_response(args)**

This step generates the final response by calling the RAGQueryProcessor:

* Initializes a ChromaDBRetriever and RAGQueryProcessor.
* Uses the processor to generate a response based on the user's query.
* Outputs the generated response or an error message if no response is generated.

A black background with a black border

AI-generated content may be incorrect.

**Main Function**

The main() function is the entry point of the pipeline. It:

* Parses command-line arguments using argparse to determine which step to execute (step01\_ingest, step02\_generate\_embeddings, step03\_store\_vectors, step04\_retrieve\_chunks, step05\_generate\_response).
* Calls the corresponding step function based on the parsed arguments.
* Configure logging and prints details about the execution.

A screen shot of a computer

AI-generated content may be incorrect.

**Command-Line Arguments**

* step: Specifies which step to execute.
* --input\_filename: Optionally specifies a file to process. Defaults to "amazon.csv".
* --query\_args: Optionally provides a search query for the retrieval and response generation steps.

**Execution Workflow & Results**

1. **Step 1**: Ingest documents, preprocess them (tokenization, chunking, CSV processing).

& "c:/Users/vikramp/OneDrive - School Health Corporation/Desktop/Assignment Files CISC 691/A04/rag\_pipeline\_env/Scripts/python.exe" "c:/Users/vikramp/OneDrive - School Health Corporation/Desktop/Assignment Files CISC 691/A04/A04\_A-Simple-RAG-Design-Implementation/hu\_sp25\_691\_a03/FullCode.py" step01\_ingest --input\_filename "amazon.csv"

A black background with many small lines

AI-generated content may be incorrect.

1. **Step 2**: Generate embeddings from the cleaned text files.

& "c:/Users/vikramp/OneDrive - School Health Corporation/Desktop/Assignment Files CISC 691/A04/rag\_pipeline\_env/Scripts/python.exe" "c:/Users/vikramp/OneDrive - School Health Corporation/Desktop/Assignment Files CISC 691/A04/A04\_A-Simple-RAG-Design-Implementation/hu\_sp25\_691\_a03/FullCode.py" step02\_generate\_embeddings --input\_filename "amazon.csv"

A computer screen with text on it

AI-generated content may be incorrect.

1. **Step 3**: Store the generated embeddings in a vector database.

& "c:/Users/vikramp/OneDrive - School Health Corporation/Desktop/Assignment Files CISC 691/A04/rag\_pipeline\_env/Scripts/python.exe" "c:/Users/vikramp/OneDrive - School Health Corporation/Desktop/Assignment Files CISC 691/A04/A04\_A-Simple-RAG-Design-Implementation/hu\_sp25\_691\_a03/FullCode.py" step03\_store\_vectors --input\_filename "amazon.csv"

A screen shot of a computer program

AI-generated content may be incorrect.

1. **Step 4**: Retrieve the most relevant document chunks based on a query.

& "c:/Users/vikramp/OneDrive - School Health Corporation/Desktop/Assignment Files CISC 691/A04/rag\_pipeline\_env/Scripts/python.exe" "c:/Users/vikramp/OneDrive - School Health Corporation/Desktop/Assignment Files CISC 691/A04/A04\_A-Simple-RAG-Design-Implementation/hu\_sp25\_691\_a03/FullCode.py" step04\_retrieve\_chunks --query\_args "best product reviews"

A screen shot of a computer program

AI-generated content may be incorrect.

1. **Step 5**: Use GPT-4 to generate a response based on the context retrieved.

& "c:/Users/vikramp/OneDrive - School Health Corporation/Desktop/Assignment Files CISC 691/A04/rag\_pipeline\_env/Scripts/python.exe" "c:/Users/vikramp/OneDrive - School Health Corporation/Desktop/Assignment Files CISC 691/A04/A04\_A-Simple-RAG-Design-Implementation/hu\_sp25\_691\_a03/FullCode.py" step05\_generate\_response --query\_args "show me top rated electronic products?"

A computer screen shot of a black screen

AI-generated content may be incorrect.

Each step can be executed individually using the command-line interface.

**Conclusion:**

The code implements a modular and extensible RAG pipeline that processes document, generates embeddings, stores them in a vector database, retrieves relevant chunks based on a query, and generates a context-aware response using GPT-4. The pipeline is flexible and can be customized by modifying the configuration file, the embeddings model, or the query parameters.

**References:**

* 1. Professor. Don Ohara sample assignment code.
  2. ChatGPT API.
  3. <https://ieeexplore.ieee.org/document/10574972>

**GitHub Repository Link:** <https://github.com/VVPPower/A04_A-Simple-RAG-Design-Implementation.git>