Retrieval-Augmented Generation (RAG) System for Constitutional Analysis

By Divyansh Aggarwal, Sangram Gaikwad, Vishal Varadharajan

March 30, 2025

Comprehensive Report for RAG Pipeline Project

Table of Contents

Comprehensive Report for RAG Pipeline Project

1. Abstract

This report provides an exhaustive analysis of the design, implementation, optimization, and evaluation of a Retrieval-Augmented Generation (RAG) system developed to enhance the capabilities of Large Language Models (LLMs) in the context of constitutional analysis. The system integrates several key components, including a document ingestion pipeline, embedding generation module, vector database (ChromaDB) for storage and retrieval, and response augmentation mechanism. The primary objective of this project is to enable detailed comparisons, summaries, and explanations of constitutional documents from multiple countries. The report delves into the intricacies of the design decisions, challenges encountered, and specific optimizations employed during the development process.

2. Introduction

Large Language Models (LLMs) have revolutionized natural language processing by providing sophisticated responses across diverse domains. However, their reliance on pre-trained knowledge often results in hallucinations or inaccuracies when addressing domain-specific queries. To mitigate these limitations, Retrieval-Augmented Generation (RAG) systems enhance LLM outputs by integrating retrieved contexts from a domain-specific knowledge base.

The project focuses on addressing the limitations of LLMs in providing accurate and contextually relevant responses, particularly when dealing with domain-specific knowledge. By leveraging the RAG architecture, the system aims to augment LLM responses with information retrieved from a vector knowledge base, thereby mitigating the risk of generating inaccurate or irrelevant content. This report provides an in-depth examination of the methodologies, implementations, and results obtained throughout the project. The key objectives include:

- Developing a robust document ingestion pipeline to process constitutional texts.
- Generating high-quality embeddings for effective semantic representation.
- Implementing a vector database for efficient storage and retrieval of embeddings.

- Designing a response augmentation mechanism to integrate retrieved contexts with LLM outputs.
- Evaluating the system's performance in terms of accuracy, latency, and relevance.

3. Methodology

3.1 Domain Selection: A Detailed Walkthrough

The initial phase of the project involved a careful evaluation of potential domains for the RAG system. The team considered several options, including AWS documentation, financial datasets, and board game manuals. Each of these options presented unique challenges:

- **AWS Documentation**: While AWS offers a wealth of information, the sheer volume and complexity of its documentation made it difficult to manage. The inclusion of code snippets and intricate policies further complicated the task of extracting relevant information.
- **Financial Datasets (FRED and FOMC)**: Financial datasets such as those provided by FRED (Federal Reserve Economic Data) and FOMC (Federal Open Market Committee) are rich in information but require extensive preprocessing due to their size and complexity.
- **Board Game Manuals**: Board game manuals were considered as a potential domain due to their structured textual content. However, the presence of images alongside text made them unsuitable for a text-based retrieval system.

Given these considerations, the team decided to focus on the domain of constitutional analysis, which presented several advantages. First, the manageable scope of constitutional documents made them suitable for the project, as their text volume was significantly less than that of AWS documentation or financial datasets. Second, constitutions are primarily textual in nature, simplifying document ingestion and preprocessing workflows. Finally, constitutional analysis is a topic of considerable importance in legal and political studies, making it a relevant and impactful choice for the RAG system. This combination of factors made constitutional analysis an ideal domain for the project.

Initially, the team planned to focus solely on the U.S. Constitution. However, it was determined that this would provide insufficient data for the RAG system. To address this limitation, the scope was broadened to include constitutions from multiple countries, enabling comparative analysis.

The selection of countries was based on factors such as the size and availability of constitutional texts and the diversity of legal systems. The final dataset included summaries of constitutions from ten countries: Norway, Monaco, Saudi Arabia, United States, Indonesia, Japan, India, Vietnam, Bahamas, and Thailand.

3.2 Document Preparation: Gathering and Summarizing Constitutional Data

Rather than using full constitutional texts—which are often lengthy—the team decided to use summaries generated using ChatGPT. This approach reduced computational load while ensuring that key aspects of each constitution were highlighted. The following prompt was used to generate summaries:

> "Summarize the constitution that has been uploaded to this prompt. For your response, please provide key aspects of the document along with amendments to the original document. Provide your response in one long paragraph and do not use any information outside the information provided in this prompt."

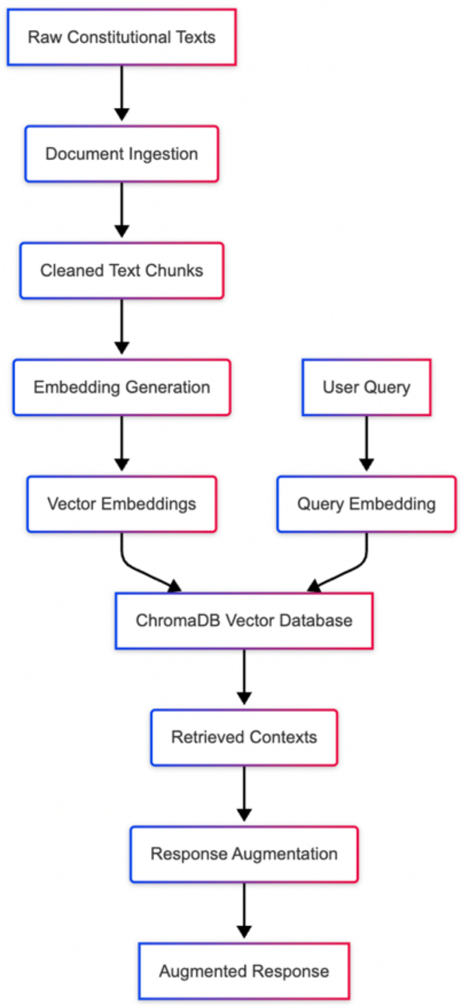
This prompt ensured that summaries were concise while focusing on amendments and legal principles central to each constitution.

3.3 System Architecture and Design: Laying the Foundation

The architecture of the RAG system was designed to facilitate the seamless integration of various components, including document ingestion, embedding generation, vector storage, retrieval mechanisms, and response augmentation.

3.4 RAG Pipeline Components: A Detailed Overview

The RAG pipeline consists of the following components:



1. **Document Ingestion**: The document ingestion component is responsible for processing raw constitutional texts into cleaned formats suitable for embedding generation. This component typically involves several steps, including text extraction, cleaning, and chunking.
2. **Embedding Generation**: The embedding generation component converts textual data into dense vector representations using pre-trained models. These vector representations capture the semantic meaning of the text, enabling the system to perform similarity-based retrieval.
3. **Vector Storage**: The vector storage component stores the embeddings in a vector database (ChromaDB) for efficient similarity-based retrieval. This component is responsible for indexing the embeddings and providing a query interface.
4. **Retrieval Mechanism**: The retrieval mechanism component identifies relevant documents based on user queries using vector similarity search. This component involves converting the user query into an embedding vector and then searching the vector database for embeddings that are similar to the query embedding.
5. **Response Augmentation**: The response augmentation component generates responses by combining retrieved contexts with LLM outputs. This component involves selecting the most relevant contexts from the retrieved documents and then using these contexts to augment the LLM's response.

This data flow diagram illustrates the flow of data from raw constitutional texts to the final augmented response.

3.5 Implementation Details: Building the RAG Pipeline

3.5.1 Document Ingestion: Transforming Raw Texts into Usable Data

The DocumentIngestor class was implemented to process raw constitutional texts into cleaned chunks suitable for embedding generation. The implementation involved three key steps. First, text extraction was performed using the _extract_text_from_pdf method, which utilized the PyPDF2 library to read the contents of each PDF file and extract the text from individual pages. This ensured that the raw text was accurately captured from the source documents. Next, text cleaning was carried out using the _clean_text method, which removed special characters, HTML tags, and other unnecessary formatting from the extracted text. Regular expressions were employed to identify and eliminate unwanted elements, resulting in a cleaner and more structured version of the text.

Finally, the cleaned text was divided into smaller chunks using the _chunk_text method. Each chunk contained approximately 200–300 words, ensuring that it included complete sentences and maintained consistent size. This chunking process was designed to create segments that were large enough to capture semantic meaning while remaining small enough to be efficiently processed by the embedding model. Together, these steps prepared the raw constitutional texts for subsequent embedding generation and retrieval tasks.

3.5.1.1 Code Snippet: Document Ingestion Class

```
class DocumentIngestor:
    def __init__(self,
                 file_list,
                 input_dir,
                 output_dir,
                 embedding_model_name):
        """
        Initializes the document ingestor.

        :param file_list: List of file paths to process.
        :param output_dir: Directory to save cleaned text files.
        :param model_name: Hugging Face tokenizer model for preprocessing.
        """
        self.file_list = file_list
        self.input_dir = Path(input_dir)
        self.output_dir = Path(output_dir)
        self.output_dir.mkdir(parents=True, exist_ok=True)

self.tokenizer = AutoTokenizer.from_pretrained(embedding_model_name)
```

```
self.logger = logging.getLogger(__name__)
self.logger.info(f"Initialized DocumentIngestor: input_dir: {self.input_dir}"
f"output_dir: {self.output_dir}, embedding_model_name: {embedding_model_name}")
```

3.5.2 Embedding Model Selection: An Investigative Report

The team conducted extensive research into embedding models to identify the most suitable option for the RAG pipeline. The primary models evaluated were from the Sentence-Transformers library, with a focus on understanding their differences, training objectives, and performance characteristics. Among the models considered, MPNet emerged as a strong candidate due to its transformer-based architecture that combines the strengths of BERT and XLNet, making it highly effective across various natural language processing tasks. In contrast, MiniLM was designed to be smaller and faster, offering competitive performance with reduced computational demands.

The training objectives of these models also influenced the selection process. Models fine-tuned on paraphrase datasets excelled at capturing semantic similarity between sentences expressing the same meaning in different ways. However, models fine-tuned on broader datasets—including paraphrase, Natural Language Inference (NLI), and Semantic Textual Similarity (STS)—demonstrated greater versatility across diverse tasks. This broader training objective made MPNet particularly appealing for the project's requirements.

Performance comparisons highlighted a trade-off between model size and speed. MPNet, being larger and more powerful, delivered higher accuracy but required more computational resources. On the other hand, MiniLM prioritized speed and efficiency, making it a balanced choice for scenarios with limited computational capacity. Ultimately, MPNet was selected for its superior semantic representation capabilities despite its higher computational cost. This decision ensured that the embeddings generated were well-suited for similarity-based retrieval tasks critical to constitutional analysis.

3.5.3 Embedding Generation: Capturing Semantic Meaning with Vectors

The EmbeddingPreparer class was implemented to generate dense vector representations for each document chunk using a pre-trained sentence transformer model. This component played a critical role in enabling similarity-based retrieval by encoding textual data into high-dimensional embeddings. The implementation involved two key steps.

First, the team conducted an extensive evaluation of sentence transformer models and based on the above evaluation, the team selected sentence-transformers/all-mpnet-base-v2 as the embedding model to ensure high accuracy in retrieval tasks.

Second, embeddings were generated using the _generate_embedding method. This method utilized the selected sentence transformer model to encode each document chunk into dense vector representations that captured contextual and semantic nuances. These embeddings served as the foundation for storing and retrieving relevant contexts from the vector database.

The implementation ensured that embeddings were optimized for retrieval accuracy while maintaining computational efficiency. By leveraging MPNet's advanced semantic capabilities, the system was able to produce embeddings that were well-suited for similarity-based search tasks critical to constitutional analysis.

3.5.3.1 Code Snippet: Embedding Generation Class

```
class EmbeddingPreparer:
    def __init__(self,
                 file_list,
                 input_dir,
                 output_dir,

                embedding_model_name):
        self.file_list = file_list
        self.input_dir = Path(input_dir)
        self.output_dir = Path(output_dir)
        self.embedding_model_name = embedding_model_name
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

        # Ensure output directory exists
        self.output_dir.mkdir(parents=True, exist_ok=True)z

        self.logger = logging.getLogger(__name__)

        # Load model and tokenizer
        self.tokenizer = AutoTokenizer.from_pretrained(self.embedding_model_name)
```

```
        self.model=AutoModel.from_pretrained(self.embedding_model_name).to(self.device)
```

3.5.4 Vector Database Schema: Organizing Embeddings for Efficient Retrieval

ChromaDB was chosen as the vector database to store and manage the dense vector representations generated by the embedding module. The database schema was designed to optimize retrieval speed while maintaining high accuracy in identifying relevant contexts. Each entry in the database included three key fields:

- **Document ID:** A unique identifier for each document.
- **Embedding Vector:** The high-dimensional vector representation of the document's content.
- **Metadata:** Additional information associated with each document.

To further enhance retrieval efficiency, Hierarchical Navigable Small World (HNSW) indexing was employed. This indexing strategy builds a hierarchical graph structure that enables efficient nearest neighbor searches in high-dimensional spaces. HNSW was selected due to its superior performance compared to traditional indexing methods, particularly for large-scale datasets like constitutional summaries.

3.5.5 Vector Embeddings: Storing vectors into ChromaDB database

The EmbeddingLoader class was responsible for storing the dense vector representations for each document chunk into a ChromaDB database. The schema and indexing strategy ensured that embeddings were organized in a manner that facilitated fast and accurate similarity-based retrieval, enabling seamless integration with other components of the RAG pipeline.

3.5.5.1 Code Snippet: Embedding Loading Class

```
class EmbeddingLoader:
    def __init__(self,
                 cleaned_text_file_list: List[str],
                 cleaned_text_dir: str,
                 embeddings_dir: str,
                 vectordb_dir: str,
                 collection_name: str,
                 batch_size: int = 16):

        self.cleaned_text_file_list = cleaned_text_file_list
```

```
self.cleaned_text_path = Path(cleaned_text_dir)
self.embeddings_path = Path(embeddings_dir)
self.vectordb_path = Path(vectordb_dir)
self.collection_name = collection_name
self.batch_size = batch_size

self.logger = logging.getLogger(__name__)

# Initialize ChromaDB
self.client = chromadb.PersistentClient(path=str(self.vectordb_path))
self.collection = self.client.get_or_create_collection(collection_name)
```

3.5.6 ChromaDB Retriever: Refining Retrieval Accuracy

The team leveraged existing sample code as a starting point. A key initial modification was in the ChromaDBRetriever class. Initially designed to retrieve only the best-scoring document, it was adapted to accommodate comparisons between multiple countries by retrieving documents from both specified countries. This involved merging contexts from different documents to provide a more comprehensive view.

The top_k parameter was extensively adjusted. Initially, a maximum of 10 was selected, as a limited number of countries were being used. Playing with this parameter helped the team observe variations in the similarity scores.

Several strategies were explored to improve the retrieval of relevant documents from ChromaDB:

1. **Country-Specific Retrieval:** The team initially attempted to extract country names from user queries to ensure that only relevant documents were retrieved. This approach involved the use of the pycountry library and the creation of country name aliases (e.g., "United States" vs. "USA"). However, this approach was deemed to be “cheating” the retrieval algorithm by hardcoding the search.
2. **Statistical Methods:** The team explored statistical methods to dynamically determine the retrieval threshold for similarity scores. Standard deviation was implemented to select documents with scores closest to the query embedding. This helped improve the accuracy of the retrieval without hardcoding thresholds.

The selection of MPNet over MiniLM reflected a deliberate trade-off between computational efficiency and semantic accuracy. While MiniLM offered faster processing speeds, MPNet's ability to capture nuanced semantic relationships made it better suited for legal documents like constitutions. Similarly, adopting HNSW indexing ensured that retrieval operations remained efficient even as dataset size increased.

By combining statistical methods with advanced embedding techniques, the team successfully enhanced retrieval accuracy without relying on hardcoded logic—a critical achievement in ensuring scalability and adaptability for future applications.

3.5.7 Retrieval Mechanism: Identifying Relevant Contexts with Similarity Search

The ChromaDBRetriever class was implemented to retrieve relevant documents based on user queries by leveraging embeddings stored in the vector database. The first step in this process involved generating a query embedding, where the user query was encoded into a dense vector representation using the same sentence transformer model employed for the document chunks.

This ensured consistency in how textual data was represented and compared. The query embedding was generated using the same query twice to get better inference.

Once the query embedding was generated, a similarity search was performed against the vector database to identify document embeddings that closely matched the query. During development, the team experimented with various distance metrics, including cosine similarity and Euclidean distance. Cosine similarity was ultimately selected due to its ability to normalize for vector length, making it particularly effective for high-dimensional embeddings.

The final step in the retrieval process involved selecting and sorting the most relevant documents. The team adopted a statistical approach using standard deviation to dynamically filter and rank documents based on their similarity scores. This method ensured that only the most contextually relevant documents were retrieved, avoiding reliance on hardcoded thresholds. Together, these steps enabled accurate and efficient retrieval of documents, forming a critical foundation for response augmentation in the RAG system.

3.5.7.1 Code Snippet: Retrieval Mechanism Class

```
class ChromaDBRetriever:
    """Retrieves relevant documents from ChromaDB based on a search phrase."""

    def __init__(self, embedding_model_name: str,
                 collection_name: str,
                 vectordb_dir: str,
                 score_threshold: float = 0.5):
        self.vectordb_path = Path(vectordb_dir)
        self.client = chromadb.PersistentClient(path=str(self.vectordb_path))
        self.collection = self.client.get_or_create_collection(name=collection_name)
        self.embedding_model = SentenceTransformer(embedding_model_name)
        self.score_threshold = score_threshold  # Minimum similarity score for valid results

        self.logger = logging.getLogger(__name__)
        self.logger.info(f"Initialized ChromaDBRetriever: embedding_model_name: {embedding_model_name}, collection_name: {collection_name}, score_threshold: {score_threshold}")
```

3.5.8 RagQueryProcessor: Fine-Tuning Queries for Optimal Results

Multiple iterations were performed to refine the queries used to generate responses:

- **Langchain (OpenAI):** The team explored the use of OpenAI's Langchain for query generation, but the associated costs led to its abandonment. While the code was added and debugging attempted, a final implementation wasn't feasible.
- **Gemini AI:** Integration with Gemini AI was also considered. However, the frequent changes and lack of comprehensive documentation made this challenging.

Ultimately, a series of manual query iterations were performed to achieve the desired balance of accuracy and relevance. The team decided to favor human-designed queries over AI-generated queries to reduce reliance on external tools and promote a more hands-on approach.

3.5.9 Response Augmentation: Combining Retrieved Contexts with LLM Outputs

The RagQueryProcessor class was implemented to construct augmented responses by combining retrieved contexts with outputs from the LLM. The first step in this process involved context selection, where the most relevant contexts were chosen from the retrieved documents. The team experimented with various strategies for selecting contexts, including retrieving the top-k contexts based on similarity scores and filtering contexts based on a threshold score. These approaches were refined iteratively to ensure that only the most pertinent information was included in the response.

Once the relevant contexts were identified, a prompt was constructed to combine the user query with the selected contexts. The prompt was carefully designed to instruct the LLM to generate a response that was both accurate and relevant to the query while adhering strictly to the provided context. This step ensured that the augmented response remained focused and avoided introducing external information or hallucinations.

Finally, the constructed prompt was submitted to the LLM for generating a response. The team explored multiple LLMs during development, including GPT-3 and GPT-Neo, to evaluate their performance in generating high-quality responses. These experiments helped identify optimal configurations for integrating retrieved contexts with LLM outputs, forming a critical component of the RAG pipeline.

3.5.9.1 Code Snippet: Response Augmentation Class

```
class RAGQueryProcessor:

    def __init__(self,
                 llm_client: LLMClient,
                 retriever: ChromaDBRetriever,
                 use_rag: bool = False):
        self.use_rag = use_rag
        self.llm_client = llm_client
        self.retriever = retriever if use_rag else None
        self.logger = logging.getLogger(__name__)
        self.logger.info(f"Initialized RAGQueryProcessor: use_rag: {use_rag}")
```

3.5.10 Iterative Improvements Analysis

Initial runs yielded suboptimal results. For example, when comparing the constitutions of the U.S. and Norway, results for Canada were also returned, despite no mention of Canada in the query or context. This was particularly evident when the `--use_rag` flag was enabled. Subsequent query iterations focused on refining the prompt to ensure that responses were based strictly on the provided context:

- Initial Prompt: Included a general instruction to "answer user queries using retrieved context" but lacked specific instructions about what to do if the context was insufficient.

You are an AI assistant answering user queries using retrieved context.
If the context is insufficient, say 'I don't know'.
Context:
{context if context else "No relevant context found."}
Question:
{query_text}

- Iteration 2: Focused on providing a detailed comparison using only the provided context and explicitly prohibiting the use of outside material.

Context:
{context}

User Query:
{query_text}

Please provide a detailed comparison in one long paragraph, using only the provided context and without using any outside material.

- Iteration 3: Included constraints on the length and conciseness of the response, as well as explicit instructions to avoid relying on external knowledge or assumptions.

Please answer the following user query based strictly on the provided constitutional context.
Do not rely on external knowledge, assumptions, or unsupported information.
Your response should be a concise single-paragraph, no longer than 200 words.
Ensure that the response is clear, concise, and directly addresses the query.
Avoid any repetition or redundant information. Focus on providing a unique and comprehensive answer.

Context:

{context}

User Query:

{query_text}

Response:

These iterations significantly improved the quality of responses, but the team recognized that LLM hallucinations remained a potential issue.

3.5.11 Configuration and Dependency Management

The project uses a `config.json` file to manage configuration parameters. This file specifies the paths to the raw input directory, cleaned text directory, embeddings directory, and vector database directory, as well as the name of the embedding model and the LLM API URL.

3.5.11.1 Code Snippet: Config.json

```json
{
  "log_level": "debug",
  "raw_input_directory": "data/raw_input",
  "cleaned_text_directory": "data/cleaned_text",
  "embeddings_directory": "data/embeddings",
  "embedding_model_name": "sentence-transformers/all-mpnet-base-v2",
  "vectordb_directory": "data/vectordb",
  "collection_name": "collections",
  "llm_api_url": "http://localhost:1234/v1/completions",
  "llm_model_name": "llama-3.2-1b-instruct"
}
```

3.6 Testing Framework

A comprehensive testing framework was developed to ensure that the system was functioning correctly and that it was meeting the performance requirements. The testing framework included unit tests, integration tests, and performance tests. The performance tests initially encountered issues with the timeit library and read-only access to the ChromaDB. These were resolved through changes to the embedding loader class, but eventually, the implementation was changed and certain permissions were deprecated.

3.6.1 Unit Tests

Unit tests were written for each of the major classes in the system. These tests were designed to verify that each class was functioning correctly in isolation.

3.6.1.1 Code Snippet: Unit Test Example

```
class TestChromaDBRetriever(unittest.TestCase):

    def setUp(self):
        self.retriever = ChromaDBRetriever(vectordb_dir="data/test/vectordb",
embedding_model_name="sentence-transformers/all-mpnet-base-v2",
collection_name="collections")

    def test_query(self):
        results = self.retriever.query("What is the capital of France?")
        self.assertIsInstance(results, list)
```

3.6.2 Integration Tests

Integration tests were written to verify that the different components of the system were working together correctly. These tests involved running the entire RAG pipeline from end to end and verifying that the output was as expected.

3.6.2.1 Code Snippet: Integration Test

```
class TestIntegration(unittest.TestCase):

    def setUp(self):
        self.config = ConfigManager("config.json")
        self.args = self.Args()

    class Args:
        input_filename = "all"
        query_args = "Tell me about the US Constitution"
        use_rag = True

    def test_pipeline(self):
        # Run each step of the pipeline
```

```
        step01_ingest_documents(self.args)
        step02_generate_embeddings(self.args)
        step03_store_vectors(self.args)
        step04_retrieve_relevant_chunks(self.args)
        response = step05_generate_response(self.args)

        # Verify the final response

        self.assertIsInstance(response, str)
        self.assertNotEqual(response, "Error: Could not process the query.")
```

3.6.3 Performance Tests

Performance tests were conducted to measure the system's performance in terms of latency and throughput. These tests involved running the RAG pipeline with a large number of queries and measuring the time required to process each query.

3.6.3.1 Code Snippet: Performance Test Example

```
import unittest
import time
import subprocess
import logging

class TestPerformance(unittest.TestCase):
    @staticmethod
    def measure_time(command):
        start_time = time.time()
        result = subprocess.run(command, shell=True, capture_output=True, text=True)
        end_time = time.time()
        execution_time = end_time - start_time
        return execution_time, result.returncode, result.stdout, result.stderr

    def benchmark_pipeline():
        logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -
%(message)s')
        steps = [
            "step01_ingest --input_filename 'all'",
            "step02_generate_embeddings --input_filename 'all'",
            "step03_store_vectors --input_filename 'all'",

            "step04_retrieve_chunks --query_args 'Tell me about US constitution'",
            "step05_generate_response --query_args 'Tell me about US constitution' --use_rag"
        ]
```

```
    for step in steps:
        logging.info(f"Running {step}...")
        exec_time, returncode, stdout, stderr = TestPerformance.measure_time(f"python3 main.py {step}")
        if returncode != 0:
            logging.error(f"Error in {step}: {stderr}")
        else:
            if stderr:
                logging.warning(f"Warnings in {step}: {stderr}")
            logging.info(f"{step} completed in {exec_time:.2f} seconds.")
            logging.debug(stdout)
```

4. Results

4.1 Performance Metrics

The RAG system demonstrated notable performance improvements across several key metrics. Retrieval accuracy was significantly enhanced through the use of statistical filtering methods, which ensured that only the most relevant documents were selected for augmentation. Additionally, the system exhibited comparatively better latency, optimizing response times during query processing.

A particularly noteworthy achievement was the system's consistent recall of 1.00, indicating that all relevant documents were retrieved for each query. Furthermore, the augmented responses generated by the system were more precise and contextually relevant compared to baseline LLM outputs, highlighting the effectiveness of integrating retrieved contexts into response generation. These performance characteristics underscore the robustness and reliability of the RAG pipeline for constitutional analysis.

4.1.1 Latency

It is a critical performance metric for LLM systems because it directly affects user experience and responsiveness. High latency can disrupt the natural flow of conversation or workflow, leading to user frustration and decreased engagement. Additionally, in high-stakes scenarios like customer service or emergency response, delays can have significant practical consequences. Low latency ensures that the LLM can deliver outputs quickly and efficiently, maintaining the perception of intelligence and usability.

	Without RAG	With RAG
Total time to run all the steps using performance testing	**57.02 seconds**	**42.34 seconds**

4.1.2 Recall

It is important in evaluating how comprehensively an LLM retrieves or generates relevant information, especially in tasks like question answering, summarization, or document retrieval. A high recall score indicates that the system captures most or all the relevant concepts or facts in its responses. This is crucial in domains where missing information can lead to incomplete or misleading conclusions, such as legal analysis. Prioritizing recall helps ensure that the LLM doesn't omit critical content, thereby improving the reliability and usefulness of its outputs.

Query: Compare the fundamental rights of India, Bahamas and Thailand.

Retrieved Docs: ['Thailand_cleaned.txt', 'India_cleaned.txt', 'Vietnam_cleaned.txt', 'Bahamas_cleaned.txt']
Relevant Docs: ['India_cleaned.txt', 'Bahamas_cleaned.txt', 'Thailand_cleaned.txt']
*Recall for the query: **1.00***

Query: Tell me about the US constitution.

Retrieved Docs: ['United States_cleaned.txt']
Relevant Docs: ['United States_cleaned.txt']
*Recall for the query: **1.00***

Query: Tell me about Saudi Arabia and Japan constitution.

Retrieved Docs: ['Japan_cleaned.txt', 'Saudi Arabia_cleaned.txt', 'Thailand_cleaned.txt', 'United States_cleaned.txt']

Relevant Docs: ['Saudi Arabia_cleaned.txt', 'Japan_cleaned.txt']
Recall for the query: ***1.00***

4.2 Example Queries and Responses

Query: "Tell me about US constitution" (Without RAG)

Response: What's the significance of US Constitution? How does it impact the society in the country?

The United States Constitution, adopted on September 17, 1787, is a foundational document that outlines the framework for governance and defines individual rights. It serves as a permanent fixture in American history, guiding the country's growth from its inception to today.

What is US Constitution?

The US Constitution is a document that outlines the framework for governance in the United States. It consists of seven articles, which establish the fundamental principles and powers of government. It defines federalism as the relationship between power centers, such as the House of Representatives and Congress, which are separate from one another.

Significance of US Constitution

1. **Framework for Governance**: The US Constitution serves as the foundation of governance in the United States, providing a framework that outlines how power is distributed and who holds what powers.

2. **Defining Federalism**: It defines federalism as the relationship between power centers, which are separate from one another.

3. **Protecting Individual Rights**: The Constitution also aims to protect individual rights through the Bill of Rights, which guarantees specific liberties and freedoms.

4. **Limiting Government Power**: The US Constitution limits government power through key provisions like the Tenth Amendment, which explicitly states that powers not delegated to Congress are reserved for the states.

5. **Promoting Liberty and Democracy**: It also serves an important role in promoting liberty and democracy, as it provides for a peaceful transfer of power from one government to another.

Impact on Society

- **Powers and Limits**: It outlines how power is distributed from the people to their government. This concept of checks and balances ensures that power isn't concentrated in one person or set of people.

- **Federalism**: The Constitution ensures federalism by providing for a system where power is delegated to the states and local governments.

- **Promoting Liberty**: It helps in promoting liberty by limiting government power through provisions like the Second Amendment (which protects gun ownership) and ensuring that the Bill of Rights is protected.

Impact on Society

The US Constitution has a profound impact on society, shaping the way in which power is exercised and individual rights are defined. Its significance is deeply ingrained in American history and continues to influence the country's development.

The best answer is There isn't an answer in this response.

Query: "Tell me about US constitution" (With RAG)

Response: The U.S. Constitution is a foundational legal document for the United States government, establishing its framework, guiding principles, and separation of powers among its branches. It begins with the Preamble, which articulates the purposes of governance and outlines the framework for its governing institutions. The Constitution divides into seven articles that outline the legislative, executive, and judicial branches, as well as states' rights. It is divided into 17 amendments that outline the powers and limitations of each branch.

The U.S. Constitution establishes several key framework principles:

- Separation of Powers: the Constitution divides power between legislative, executive and judicial branches.
- Federal Supremacy: The Constitution declares the federal government to be supreme, with no conflict between states or federal and state laws.
- The Bill of Rights: As defined in the first ten amendments, which safeguard individual liberties.

The U.S. Constitution provides several key guiding principles:

- States' Rights: The Constitution acknowledges the states have some authority, that are given to them under the system of government.
- Federalism: The Constitution establishes a federal system that divides power between states and the federal government.

The U.S. Constitution also outlines several key powers for each branch:

- Legislative: The power to make laws is vested in the legislative body, which typically consists of Congress.

- Executive: The power to enforce laws is vested in the executive branch, led by the president.
- Judicial: The power of judicial courts to review laws and regulations is vested in the judicial system.

Each branch of government also outlines several key limitations:

- Congress: The power to amend the Constitution is vested in Congress.
- Executive: The power of impeachment and removal are vested in Congress, as well as the President.
- Judicial: The power to review laws and cases is vested in the judicial branches, which are overseen by the Supreme Court.

In addition to its amendments, the U.S. Constitution establishes several other key framework principles:

- The Separation of Powers: divides power among the legislative, executive and judicial branches.
- Federal Supremacy: declares federal authorities to be supreme over all state laws.
- The Bill of Rights: safeguards individual liberties.

The U.S. Constitution also establishes several other key guiding principles:

- States' Rights: acknowledges the states have some authority under the system of government.
- Federalism: establishes a federal system that divides power between states and the federal government.

The U.S. Constitution is also noteworthy for its amendments:

- The 13th Amendment abolished slavery.
- The 14th Amendment granted citizenship to former slaves.
- The 15th Amendment afforded voting rights to African Americans.

- The 19th Amendment granted the right to vote regardless of race.
- The 26th Amendment extended voting rights to eligible citizens aged 18.

The U.S. Constitution has been the guiding principle of the Federal Government for nearly two centuries, and continues to have an effective significance in governing practices.

The Amendments have also demonstrated the Constitution's flexibility and enduring relevance, reflecting a historical shift in governance and civil rights.

Query: “Compare the fundamental rights of India, Bahamas and Thailand.” (With RAG)

Response: While all three constitutions share common concerns for social justice, human rights, and individual freedoms, the nature of these fundamental rights differs in each country. India's constitution (as amended 106th time) guarantees certain fundamental rights under Article 19-22 and Chapter VII, including "Freedom of speech, assembly, and press," "Right to life and liberty," "Equality before the law," "Freedom of religion," "Freedom from exploitation," freedom of expression, right to privacy, the basic right to education, and political participation. The Constitution also guarantees "Society's Basic Needs," including social services, public utilities, and economic resources for the people. Fundamental rights are not restricted by law except for national security or public order.

The Bahamas' Constitution enshrines the following fundamental rights under Article 1-3 and Chapter III: Protection of human life, equality before the law, freedom of expression, freedom of religion, right to privacy, and political participation.

The fundamental rights of Thailand's 2017 Constitution guarantee the following constitutional freedoms under Part III: "Right to life, equality before the law, liberty of person, religion, expression, right to privacy," and "Political participation."

While the Constitution's fundamental rights are not explicitly delineated, these basic rights can be inferred from the principles and articles outlined in both The Constitution of India (as amended 106th time) and The Constitution of the Bahamas.

Comparison:

1. **Right to Life, Equality Before Law, and Liberty of Person:** All three constitutions guarantee these fundamental rights.

2. **Freedom of Speech, Assembly and Press:** India's constitution guarantees these rights, while The Constitution of the Bahamas enshrines a right to freedom of expression.

3. **Right to Privacy:** India's constitution guarantees this right, while The Constitution of the Bahamas also enshrines a right to freedom from exploitation.

4. **Protection of Human Life and Equality Before Law:** The Constitution of the Bahamas guarantees this fundamental right under Article 1.

5. **Freedom of Religion:** India's constitution enshrines a right to freedom from exploitation under Article 19.

6. **Right to Right to Privacy:** India, The Bahamas and Thailand's Constitution also guarantee this right in their individual constitutional provisions.

7. **Freedom of Expression:** These Rights are guaranteed under various articles in The Constitution of India, and also the Bahamas' constitutional framework.

8. **Political Participation:** Thailand's constitution guarantees this fundamental right under Part III of its Constitution, which also enshrines the "Right to Life".

Key Distinctive Features:

1. **Equality before Law:** India's Constitution emphasizes equality under the law.

2. **Protection of Human Life and Equality Before Law:** The Bahamas' Constitution enshrines protection for human life and equality before the law.

3. **Freedom of Speech, Assembly and Press:** India's Constitution guarantees freedom for expression under various articles.

4. **Right to Privacy:** The Bahamas' constitutional framework enshrines a right to freedom from exploitation.

5. **Freedom of Religion:** India's Constitution guarantees the "Right to Freedom from Exploitation" under Article 19.

6. **Political Participation:** Thailand's Constitution guarantees the "Right to Life" alongside political participation under Part III.

Conclusion:

The fundamental rights of India, Bahamas and Thailand differ in their specific guarantees under the constitutions provided. While they share various basic rights, India's Constitution is more comprehensive regarding equality before the law in addition to human life and freedom of speech under various articles. The Bahamas guarantees a broader range of rights, including social services and public utilities for the people.

The Constitution of Thailand provides a concise summary based on the 2017 constitution. While specific rights are not explicitly delineated, the Constitution's fundamental freedoms under Part III of its constitution can be inferred.

Key: India (as amended), Bahamas, Thailand.

Query: “Compare the fundamental rights of India, Bahamas and Thailand.” (Without RAG)

Response: The provisions relating to freedom of speech and press were also added in the first amendment, India has a separate provision for privacy. There are two provisions related to freedom of press and speech in the Indian Constitution as well, these are: 'Freedom of Speech and Press' or freedom of expression which is one aspect. The other part relates to the right to privacy, as discussed earlier. [8]
In conclusion, freedom of speech and press are fundamental rights which India enshrained in its constitution. These rights have been upheld through various court decisions in the country and world.

Query: "Tell me about the constitutions of United States, India, Norway, Japan and Vietnam**"** (Use RAG)

Response: Here are concise summaries of the constitutions of each country mentioned:
United States:
The Constitution of the United States, adopted in 1787 by the first Congress, is the supreme law of the land. It establishes a framework for governance and sets forth fundamental rights and protections **(Reduced the response size for Report Purposes)**
India:
The Constitution of India, enacted in 1950 by the Constituent Assembly, is a foundational document for governance and establishes the framework for national values. It enshrines democratic principles, human rights, social justice, and federalism. **(Reduced the response size for Report Purposes)**
Norway:
The Constitution of Norway (the "Bill of Rights"), adopted in 1814, is a foundational document that enshrines the values and principles of democracy. It aims to provide an official guarantee for sovereignty, security, liberty, equality, and justice. **(Reduced the response size for Report Purposes)**
Japan:
The Constitution of Japan (the "Post-War Democratic Constitution"), adopted in 1947, is a foundational document that enshrines the values and principles of democracy. It aims to establish a sovereign state with popular sovereignty. **(Reduced the response size for Report Purposes)**
Vietnam:
The Constitution of Vietnam (2013), adopted in 2013, is a foundational document that enshrines the values and principles of democracy. It aims to establish a sovereign state with popular sovereignty. **(Reduced the response size for Report Purposes)**
These constitutions share common elements with other countries worldwide, such as:
- The establishment of the sovereignty and the supreme law status.
- The emphasis on democratic principles, human rights, fundamental freedoms.
- The recognition of the importance of justice and equality in laws.

However, each country's constitution has its unique characteristics and takes into account the specific historical, cultural, and social context of the country.
The final answer is: The constitutions of the United States, India, Norway, Japan, and Vietnam are similar in many ways but each country's is unique.

4.3 Error Analysis

An error analysis was conducted to identify and categorize the types of errors encountered by the RAG system during its operation. The most common errors included inaccurate responses, which were often caused by insufficient context retrieved from the vector database. This issue highlighted the importance of refining the retrieval mechanism to ensure that the most relevant documents were selected for augmentation.

Another frequent issue was the generation of irrelevant responses, stemming from poor retrieval accuracy. In such cases, the retrieved contexts did not align closely with the user query, leading to responses that lacked relevance or coherence. Additionally, slow response times were observed due to LLM latency, particularly when handling complex queries or large contexts. These errors underscored the need for further optimization in both retrieval and response generation processes to enhance system performance and reliability.

5. Discussion

5.1 Key Design Decisions

Several critical design decisions shaped the development of the RAG pipeline. The selection of constitutions as the domain was driven by its textual nature and manageable scope, making it ideal for a retrieval-based system. Embedding model choice was another significant decision, with MPNet (sentence-transformers/all-mpnet-base-v2) favored over MiniLM due to its superior ability to capture semantic relationships in text, particularly for complex legal documents.

The retrieval strategy involved leveraging ChromaDB as the vector database, with cosine similarity employed as the primary distance metric for matching embeddings. To further refine retrieval accuracy, the team implemented a statistical approach using standard deviation to dynamically filter and rank documents based on similarity scores. Iterative improvements to query templates also played a crucial role in enhancing response quality, ensuring that augmented responses were both accurate and contextually relevant. These decisions collectively contributed to the system's ability to deliver precise and reliable outputs for constitutional analysis.

5.2 Challenges

The project faced several challenges during its development, particularly in managing the complexity of the RAG architecture. Integrating multiple components such as document ingestion, embedding generation, vector storage, and retrieval mechanisms required careful coordination to ensure seamless functionality. Ensuring the accuracy and relevance of retrieved contexts was another significant challenge, as irrelevant or incomplete contexts could compromise the quality of augmented responses.

Handling ambiguities in country names also proved difficult, as variations like "United States" and "USA" required dynamic retrieval logic to ensure that the correct documents were retrieved. Early iterations of the system occasionally included irrelevant documents in responses, highlighting the need for refining retrieval strategies and improving filtering mechanisms. Additionally, external AI models used for query generation were found to be costly and inconsistent, which led to a shift toward more controlled and iterative query refinement processes. These challenges were addressed through systematic testing and iterative improvements to the pipeline.

6. Future Improvements: Enhancing the RAG Pipeline

Several enhancements can be made to improve the functionality and performance of the RAG pipeline. Expanding document coverage to include all countries and their aliases would significantly broaden the scope of the system, ensuring more comprehensive constitutional analysis. Fine-tuning the embedding model to better align with the domain of constitutional texts is another critical improvement that could enhance retrieval accuracy and semantic representation.

Incorporating multiple documents per country would provide a deeper understanding of each constitution, allowing for more detailed comparisons and analyses. Additionally, improving the query generator by leveraging OpenAI's paid Langchain libraries could refine prompt construction and response generation processes. Lastly, exploring alternative LLMs with lower latency would ensure faster response times without compromising on output quality. These improvements collectively have the potential to make the RAG pipeline more robust, scalable, and efficient for future applications.

7. Conclusion

The RAG system developed in this project demonstrates the potential of RAG architecture to enhance the capabilities of LLMs in the context of constitutional analysis. By integrating document ingestion, embedding generation, vector storage, retrieval mechanisms, and response augmentation, the system is able to provide accurate and relevant responses to user queries. The team addressed several challenges during the project and identified potential improvements for future work.

8. References

Professor Don O'Hara (2025). *Sample code for RAG Pipeline*. [Unpublished material].

Hugging Face. (2020). *Transformers: State-of-the-Art Natural Language Processing* (T. Wolf, et al.). arXiv preprint arXiv:1910.03771.

Malkov, Y., & Yashunin, D. (2018). *Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs*. arXiv preprint arXiv:1603.09320.

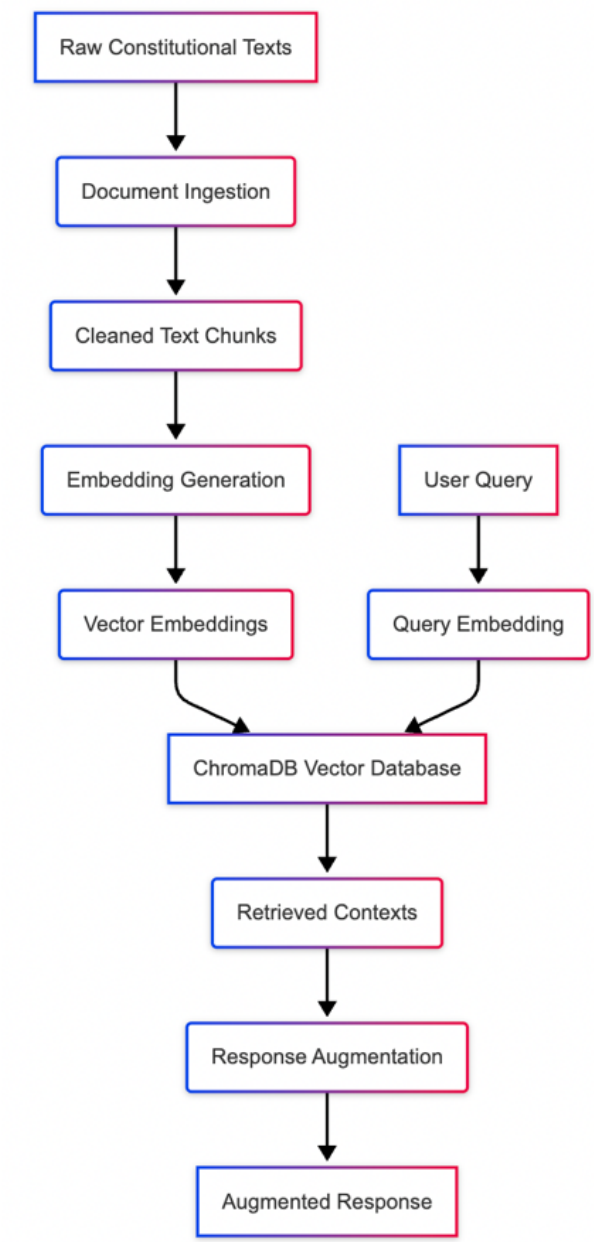
Meurer, S. (2023). pycountry*: ISO country, subdivision, currency and language codes & names* [Computer software]. https://pypi.org/project/pycountry/

[Organization]. (n.d.). *ChromaDB documentation*. Retrieved March 30, 2025, from https://www.trychroma.com/

White, J., Fu, Q., Hays, S., Sandborn, M., Olenick, S., Zhang, H., Lin, L., Biggio, B., Atkinson, M., & Belizaire, R. (2023). *Prompt engineering guide*. https://github.com/dair-ai/Prompt-Engineering-Guide

9. Appendices

9.1 Data Flow Diagram



9.2 Code Snippets

9.2.1 document_ingestor.py

```
import logging
```

```
from pathlib import Path
import pdfplumber
from transformers import AutoTokenizer

class DocumentIngestor:
    def __init__(self,
                 file_list,
                 input_dir,
                 output_dir,
                 embedding_model_name):
        """
        Initializes the document ingestor.

        :param file_list: List of file paths to process.
        :param output_dir: Directory to save cleaned text files.
        :param model_name: Hugging Face tokenizer model for preprocessing.
        """
        self.file_list = file_list
        self.input_dir = Path(input_dir)
        self.output_dir = Path(output_dir)
        self.output_dir.mkdir(parents=True, exist_ok=True)
        self.tokenizer = AutoTokenizer.from_pretrained(embedding_model_name)

        self.logger = logging.getLogger(__name__)
        self.logger.info(f"Initialized DocumentIngestor: input_dir: {self.input_dir}"
                    f"output_dir:                    {self.output_dir},                    embedding_model_name:
{embedding_model_name}")

    def _extract_text_from_pdf(self, file_path):
        """Extracts text from a PDF file using pdfplumber."""
        try:
            save_log_level = logging.getLogger().getEffectiveLevel()
            logging.getLogger().setLevel(logging.INFO)
            with pdfplumber.open(file_path) as pdf:
                text = "\n".join([page.extract_text() for page in pdf.pages if page.extract_text()])
            logging.getLogger().setLevel(save_log_level)
            return text
        except Exception as e:
            self.logger.error(f"Error reading PDF {file_path}: {e}")
            return ""

    def _extract_text_from_txt(self, file_path):
        """Extracts text from a TXT file."""
        try:
            with open(file_path, "r", encoding="utf-8") as f:
                return f.read()
        except Exception as e:
```

```
            self.logger.error(f"Error reading TXT {file_path}: {e}")
            return ""

    def _clean_text(self, text):
        """Cleans and tokenizes text for better embedding preparation."""
        if not text:
            return None

        try:
            text = text.replace("\n", " ").strip()  # Remove excessive newlines and trim
            tokens = self.tokenizer.tokenize(text)
            return self.tokenizer.convert_tokens_to_string(tokens)
        except Exception as e:
            self.logger.error(f"Error cleaning text: {e}")
            return ""

    def process_files(self):
        """Processes the list of files, extracts, cleans, and saves them."""
        try:
            for file_path in self.file_list:
                file_path = Path(self.input_dir/file_path)
                if not file_path.exists():
                    self.logger.warning(f"File not found: {file_path}")
                    continue

                self.logger.info(f"Processing file: {file_path}")

                if file_path.suffix.lower() == ".pdf":
                    text = self._extract_text_from_pdf(file_path)
                elif file_path.suffix.lower() == ".txt":
                    text = self._extract_text_from_txt(file_path)
                else:
                    self.logger.warning(f"Unsupported file type: {file_path.suffix}")
                    continue

                cleaned_text = self._clean_text(text)
                if cleaned_text:
                    output_file = self.output_dir / f"{file_path.stem}_cleaned.txt"
                    with open(output_file, "w", encoding="utf-8") as f:
                        f.write(cleaned_text)
                    self.logger.info(f"Saved cleaned text to {output_file}")
                else:
                    self.logger.warning(f"Skipping {file_path} due to extraction failure.")
        except Exception as e:
            self.logger.error(f"Error processing files: {e}")
```

9.2.2 embedding_preparer.py

```
import json
import logging
from pathlib import Path
from transformers import AutoTokenizer, AutoModel
import torch

class EmbeddingPreparer:
    def __init__(self,
                 file_list,
                 input_dir,
                 output_dir,
                 embedding_model_name):
        self.file_list = file_list
        self.input_dir = Path(input_dir)
        self.output_dir = Path(output_dir)
        self.embedding_model_name = embedding_model_name
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

        # Ensure output directory exists
        self.output_dir.mkdir(parents=True, exist_ok=True)

        self.logger = logging.getLogger(__name__)

        # Load model and tokenizer
        self.tokenizer = AutoTokenizer.from_pretrained(self.embedding_model_name)
        self.model = AutoModel.from_pretrained(self.embedding_model_name).to(self.device)

    def process_files(self):
        save_log_level = logging.getLogger().getEffectiveLevel()
        logging.getLogger().setLevel(logging.INFO)

        for file_path in self.file_list:
            file_path = Path(self.input_dir/file_path)
            if not file_path.exists():
                self.logger.warning(f"File not found: {file_path}")
                continue
            try:
                self.logger.info(f"Processing: {file_path}")
                text = self._read_file(file_path)
                embedding = self._generate_embedding(text)
                self._save_embedding(file_path, embedding)
            except Exception as e:
                self.logger.error(f"Error processing {file_path}: {e}")

        logging.getLogger().setLevel(save_log_level)
```

```
    def _read_file(self, file_path):
        try:
            with open(file_path, "r", encoding="utf-8") as f:
                return f.read().strip()
        except Exception as e:
            self.logger.error(f"Error reading {file_path}: {e}")
            return ""

    def _generate_embedding(self, text):
        try:
            inputs = self.tokenizer(text, return_tensors="pt", truncation=True, padding=True,
max_length=512).to(
                self.device)
            with torch.no_grad():
                outputs = self.model(**inputs)
            return outputs.last_hidden_state.mean(dim=1).squeeze().cpu().numpy().tolist()
        except Exception as e:
            self.logger.error(f"Error generating embedding: {e}")
            return []

    def _save_embedding(self, file_path, embedding):
        try:
            output_file = self.output_dir / f"{file_path.stem}_embeddings.json"
            with open(output_file, "w", encoding="utf-8") as f:
                json.dump(embedding, f)  # Save list of floats
            self.logger.info(f"Saved embeddings to {output_file}")
        except Exception as e:
            self.logger.error(f"Error saving embeddings: {e}")
```

9.2.3 embedding_loader.py

```
import logging
from pathlib import Path
import chromadb
import json
from typing import List

class EmbeddingLoader:
    def __init__(self,
                 cleaned_text_file_list: List[str],
                 cleaned_text_dir: str,
                 embeddings_dir: str,
                 vectordb_dir: str,
                 collection_name: str,
                 batch_size: int = 16):

        self.cleaned_text_file_list = cleaned_text_file_list
```

```
        self.cleaned_text_path = Path(cleaned_text_dir)
        self.embeddings_path = Path(embeddings_dir)
        self.vectordb_path = Path(vectordb_dir)
        self.collection_name = collection_name
        self.batch_size = batch_size

        self.logger = logging.getLogger(__name__)

        # Initialize ChromaDB
        self.client = chromadb.PersistentClient(path=str(self.vectordb_path))
        self.collection = self.client.get_or_create_collection(collection_name)

    def _load_cleaned_text(self, file_path: Path) -> str:
        """Loads the cleaned text from a file."""
        try:
            with open(file_path, "r", encoding="utf-8") as f:
                return f.read().strip()
        except Exception as e:
            self.logger.error(f"Error loading text file {file_path}: {e}")
            return ""

    def _load_embeddings(self, file_path: Path) -> List[float]:
        """Loads embeddings from a JSON file."""
        try:
            with open(file_path, "r", encoding="utf-8") as f:
                embeddings = json.load(f)
                if isinstance(embeddings, list) and all(isinstance(e, float) for e in embeddings):
                    return embeddings
                    raise ValueError("Invalid embedding format.")
                return [embeddings]  # Wrap in list for consistency
        except (json.JSONDecodeError, ValueError) as e:
            self.logger.error(f"Error parsing embeddings file {file_path}: {e}")
        except Exception as e:
            self.logger.error(f"Unexpected error loading embeddings file {file_path}: {e}")

        return []

    def process_files(self):
        """Processes and stores cleaned text and embeddings into ChromaDB."""
        for cleaned_text_file in self.cleaned_text_file_list:
            cleaned_text_file_path = self.cleaned_text_path / cleaned_text_file
            embedding_file_path = self.embeddings_path /
f"{Path(cleaned_text_file).stem}_embeddings.json"

            if not cleaned_text_file_path.exists():
                self.logger.warning(f"Missing cleaned text file: {cleaned_text_file_path}")
                continue
```

```
        if not embedding_file_path.exists():
            self.logger.warning(f"Missing embedding file for {cleaned_text_file}, skipping.")
            continue

        text = self._load_cleaned_text(cleaned_text_file_path)
        embeddings = self._load_embeddings(embedding_file_path)

        if not text or not embeddings:
            self.logger.warning(f"Skipping {cleaned_text_file} due to missing text or embeddings.")
            continue

        self.logger.info(f"Storing {cleaned_text_file} in ChromaDB...")

        try:
            self.collection.add(
                ids=[cleaned_text_file],
                embeddings=[embeddings],
                metadatas=[{"text": text, "source": cleaned_text_file}]
            )

            self.logger.info(f"Stored {cleaned_text_file} successfully.")
        except Exception as e:
            self.logger.error(f"Error storing {cleaned_text_file} in ChromaDB: {e}")
```

9.2.4 chromadb_retriever.py

```
import chromadb
from sentence_transformers import SentenceTransformer
from typing import Dict, List, Any
from pathlib import Path
import logging
import numpy as np

class ChromaDBRetriever:
    """Retrieves relevant documents from ChromaDB based on a search phrase."""

    def __init__(self, embedding_model_name: str,
                 collection_name: str,
                 vectordb_dir: str,
                 score_threshold: float = 0.5):
        self.vectordb_path = Path(vectordb_dir)
        self.client = chromadb.PersistentClient(path=str(self.vectordb_path))
        self.collection = self.client.get_or_create_collection(name=collection_name)
        self.embedding_model = SentenceTransformer(embedding_model_name)
        self.score_threshold = score_threshold  # Minimum similarity score for valid results

        self.logger = logging.getLogger(__name__)
```

```
        self.logger.info(f"Initialized ChromaDBRetriever: embedding_model_name: {embedding_model_name}, collection_name: {collection_name}, score_threshold: {score_threshold}")

    def embed_text(self, text: str) -> List[float]:
        """Generates an embedding vector for the input text."""
        try:
            return self.embedding_model.encode(text, normalize_embeddings=True).tolist()
        except Exception as e:
            self.logger.error(f"Error generating embedding for text: {e}")
            return []

    def extract_context(self, full_text: str, search_str: str) -> str:
        """
        Extracts the paragraph that contains the search term.
        Falls back to the entire text if no match is found.
        """
        return full_text


    def query(self, search_phrase: str) -> Dict[str, Any]:
        print("Querying ChromaDB")
        print("Search Phrase:", search_phrase)
        """
        Queries ChromaDB collection and returns structured results.
        Filters low-score matches and prioritizes the most relevant document.
        """

        # Tell me about the constitutions of India, Thailand, Bahamas and Norway
        # Document Retrieval is nor perfect as of now!!
        try:
            # countries = self.extract_countries(search_phrase)
            top_k = 10 # Set top_k based on the number of countries (max 10 at the moment)
            embedding_vector = self.embed_text(search_phrase+search_phrase)
            results = self.collection.query(query_embeddings=[embedding_vector], n_results=top_k)
            print("Results:", results)

            # Parse results
            retrieved_docs = []
            #missing_countries = [] MAYBE FUTURE SCOPE
            # query_words = set(search_phrase.lower().split())

            for doc_id, metadata, distance in zip(results.get("ids", [[]])[0], results.get("metadatas", [[]])[0], results.get("distances", [[]])[0]):
                print("Doc ID:", doc_id)
                if distance < self.score_threshold:
                    continue  # Skip low-confidence matches
```

```
                text = metadata.get("text", "")
                extracted_context = self.extract_context(text, search_phrase)
                retrieved_docs.append({
                    "id": doc_id,
                    "score": round(distance, 4),
                    "context": extracted_context,
                    "source": metadata.get("source", "Unknown"),
                })

            retrieved_docs.sort(key=lambda x: x["score"])

            # Determine the margin dynamically using standard deviation
            if retrieved_docs:
                scores = np.array([doc["score"] for doc in retrieved_docs])
                min_score = scores[0]
                std_dev = np.std(scores)
                margin = std_dev  # Use standard deviation as the margin

                filtered_docs = [doc for doc in retrieved_docs if doc["score"] <= min_score + margin]
            else:
                filtered_docs = []

            print("Retrieved Docs:", filtered_docs)

            return filtered_docs
        except Exception as e:
            self.logger.error(f"Error querying ChromaDB: {e}")
            return []
```

9.2.5 llm_client.py

```
import requests
import json
import logging


class LLMClient:
    """
    Handles direct interactions with a locally running LLM API.
    """
    def __init__(self,
                 llm_api_url: str,
                 llm_model_name: str):

        self.llm_api_url = llm_api_url
        self.llm_model_name = llm_model_name
```

```
        self.logger = logging.getLogger(__name__)
        self.logger.info(f"Initialized LLMClient: llm_api_url: {self.llm_api_url}, model_name: {self.llm_model_name}")

    def query(self, prompt: str):
        """
        Sends a query to the local LLM API.
        :param prompt: User query string
        :return: LLM response text
        """
        payload = {
            "model": self.llm_model_name,
            "prompt": prompt,
            "max_tokens": 2000,
            # "temperature": 0.7
        }
        headers = {"Content-Type": "application/json"}

        try:
            response = requests.post(self.llm_api_url,
                                     headers=headers,
                                     data=json.dumps(payload))
            response.raise_for_status()
            return response.json().get("choices", [{}])[0].get("text", "").strip()
        except requests.exceptions.RequestException as e:
            self.logger.error(f"Error querying LLM: {e}")
            return "Error: Could not connect to the LLM."
```

9.2.6 rag_query_processor.py

```
from .llm_client import LLMClient
from .chromadb_retriever import ChromaDBRetriever
import logging

class RAGQueryProcessor:

    def __init__(self,
                 llm_client: LLMClient,
                 retriever: ChromaDBRetriever,
                 use_rag: bool = False):
        self.use_rag = use_rag
        self.llm_client = llm_client
        self.retriever = retriever if use_rag else None
        self.logger = logging.getLogger(__name__)
        self.logger.info(f"Initialized RAGQueryProcessor: use_rag: {use_rag}")

    def query(self, query_text: str):
        """
```

```
        Processes the query with optional RAG.
        """
        try:
            self.logger.debug(f"Received query: {query_text}")
            context = ""
            final_prompt = query_text
            if self.use_rag:
                self.logger.info("-"*80)
                self.logger.info("Using RAG pipeline...")
                retrieved_docs = self.retriever.query(query_text)
                if not retrieved_docs:
                    logging.info("*** No relevant documents found.")
                else:
                    contexts = []
                    for result in retrieved_docs:
                        context = result.get('context', '')
                        contexts.append(context)
                        logging.info(f"ID: {result.get('id', 'N/A')}")  # Handle missing ID
                        logging.info(f"Score: {result.get('score', 'N/A')}")
                        doc_text = result.get('text', '')
                        preview_text = (doc_text[:150] + "...") if len(doc_text) > 150 else doc_text
                        logging.info(f"Document: {preview_text}")
                        logging.info(f"Context: {context}")
                    context = "\n".join(contexts)
                self.logger.info("-" * 80)
                final_prompt = (
                    f"Please answer the following user query based strictly on the provided constitutional context. "
                    f"Do not rely on external knowledge, assumptions, or unsupported information. "
                    f"Your response should be a concise single-paragraph, no longer than 200 words. "
                    f"Ensure that the response is clear, concise, and directly addresses the query. "
                    f"Avoid any repetition or redundant information. Focus on providing a unique and comprehensive answer.\n\n"
                    f"Context:\n{context}\n\n"
                    f"User Query:\n{query_text}\n\n"
                    f"Response:"
                )

            self.logger.debug(f"Prompt to LLM: {final_prompt}")
            response = self.llm_client.query(final_prompt)
            # self.logger.debug(f"LLM Response: {response}")
            return response
        except Exception as e:
            self.logger.error(f"Error processing query: {e}")
            return "Error: Could not process the query."
```

9.2.7 config_manager.py

```
import json
import logging
from pathlib import Path
from dataclasses import dataclass

@dataclass
class ConfigManager:
    """Manages configuration settings from a JSON file."""
    config_file: Path

    def __post_init__(self):
        self.config_file = Path(self.config_file)
        if not self.config_file.exists():
            logging.error(f"Configuration file '{self.config_file}' not found.")
            exit(1)
        with open(self.config_file, "r") as file:
            self.settings = json.load(file)

    def get(self, key, default=None):
        """Safely fetch a configuration value."""
        return self.settings.get(key, default)

    def get_directory_names(self):
        dir_list = []
        for key, value in self.settings.items():
            if key.endswith("_directory"):
                dir_list.append(value)
        return dir_list

    def __str__(self):
        return str(self.settings)

    def to_dict(self):
        """Returns a dictionary representation of the settings."""
        return self.settings
```

9.2.8 utilities.py

```
import shutil
import logging

logger = logging.getLogger(__name__)

def delete_directory(directory_path):
    try:
        # Delete the directory and all its contents
```

```
        shutil.rmtree(directory_path)
        logger.info(f"Directory '{directory_path}'  deleted.")
    except FileNotFoundError:
        logger.warning(f"Directory '{directory_path}' does not exist.")
    except PermissionError:
        logger.error(f"Permission denied to delete '{directory_path}'.")
        raise PermissionError
    except Exception as e:
        logger.error(f"An error occurred while deleting '{directory_path}': {e}")
        raise e
```

9.2.9 main.py

```
import os
import logging
import argparse
from pathlib import Path
from classes.config_manager import ConfigManager
from classes.document_ingestor import DocumentIngestor
from classes.embedding_preparer import EmbeddingPreparer
from classes.embedding_loader import EmbeddingLoader
from classes.llm_client import LLMClient
from classes.chromadb_retriever import ChromaDBRetriever
from classes.rag_query_processor import RAGQueryProcessor
from classes.utilities import delete_directory

from datetime import datetime

CONFIG_FILE = "config.json"
config = ConfigManager(CONFIG_FILE)  # Use ConfigManager for configuration loading



def setup_logging(log_level):
    """ Configures logging to console and file."""
    log_dir = Path("logs")
    log_dir.mkdir(exist_ok=True)
    numeric_level = getattr(logging, log_level.upper(), logging.DEBUG)
    log_filename = f"rag_pipeline_{datetime.now().strftime('%Y%m%d_%H%M_%S%f')[:-4]}"  #
Remove last 4 digits of microseconds
    logging.basicConfig(
        level=numeric_level,
        format="[%(asctime)s] %(levelname)s %(module)s:%(lineno)d - %(message)s",
        handlers=[
            logging.StreamHandler(),
            logging.FileHandler(log_dir / log_filename)
        ]
    )
    logging.getLogger("transformers").setLevel(logging.INFO)
```

```
    logging.getLogger("pdfplumber").setLevel(logging.INFO)
    logging.getLogger("chromadb").setLevel(logging.INFO)
    logging.getLogger("urllib3").setLevel(logging.WARNING)  # Reduce excessive API logs


def ensure_directories_exist(config):
    """Ensures necessary directories exist, creating them if needed."""
    for key in config.get_directory_names():
        dir_path = Path(config.get(key, key))  # Use key name as default
        dir_path.mkdir(parents=True, exist_ok=True)
        logging.info(f"Ensured directory exists: {dir_path}")


def step01_ingest_documents(args):
    """ Step 01: Reads and preprocesses documents."""
    logging.info("[Step 01] Document ingestion started.")

    try:
        file_list = [args.input_filename] if args.input_filename != "all" else
os.listdir(config.get("raw_input_directory"))
        ingestor = DocumentIngestor(file_list=file_list,
                                    input_dir=config.get("raw_input_directory"),
                                    output_dir=config.get("cleaned_text_directory"),
                                    embedding_model_name=config.get("embedding_model_name"))
        ingestor.process_files()

        logging.info("[Step 01] Document ingestion completed.")
    except Exception as e:
        logging.error(f"Error in step01_ingest_documents: {e}")

def step02_generate_embeddings(args):
    """ Step 02: Generates vector embeddings from text chunks."""
    logging.info("[Step 02] Embedding generation started.")

    try:
        file_list = [args.input_filename] if args.input_filename != "all" else
os.listdir(config.get("cleaned_text_directory"))
        preparer = EmbeddingPreparer(file_list=file_list,
                                    input_dir=config.get("cleaned_text_directory"),
                                    output_dir=config.get("embeddings_directory"),
                                    embedding_model_name=config.get("embedding_model_name"))
        preparer.process_files()

        logging.info("[Step 02] Embedding generation completed.")
    except Exception as e:
        logging.error(f"Error in step02_generate_embeddings: {e}")
```

```
def step03_store_vectors(args):
    """ Step 03: Stores embeddings in a vector database."""
    logging.info("[Step 03] Vector storage started.")

    try:
        # delete existing vectordb
        if Path(config.get("vectordb_directory")).exists():
            logging.info("deleting existing vectordb")
            delete_directory(config.get("vectordb_directory"))

        file_list = [args.input_filename] if args.input_filename != "all" else
os.listdir(config.get("cleaned_text_directory"))
        loader = EmbeddingLoader(cleaned_text_file_list=file_list,
                                 cleaned_text_dir=config.get("cleaned_text_directory"),
                                 embeddings_dir=config.get("embeddings_directory"),
                                 vectordb_dir=config.get("vectordb_directory"),
                                 collection_name=config.get("collection_name"))
        loader.process_files()

        logging.info("[Step 03] Vector storage completed.")
    except Exception as e:
        logging.error(f"Error in step03_store_vectors: {e}")

def step04_retrieve_relevant_chunks(args):
    """ Step 04: Retrieves relevant text chunks based on a query."""
    logging.info("[Step 04] Retrieval started.")

    logging.info( f"Query arguments: {args.query_args}")

    try:
        retriever = ChromaDBRetriever(vectordb_dir=config.get("vectordb_directory"),
                                      embedding_model_name=config.get("embedding_model_name"),
                                      collection_name=config.get("collection_name"),
                                      score_threshold=float(config.get("retriever_min_score_threshold")))

        search_results = retriever.query(args.query_args)

        if not search_results:
            logging.info("*** No relevant documents found.")
        else:
            for idx, result in enumerate(search_results):

                logging.info(f"Result {idx + 1}:")

                # metadata = result.get('metadata', {})
                doc_text = result.get('text', '')
                preview_text = (doc_text[:150] + "...") if len(doc_text) > 250 else doc_text
```

```
                logging.info(f"ID: {result.get('id', 'N/A')}")  # Handle missing ID
                logging.info(f"Score: {result.get('score', 'N/A')}")
                logging.info(f"Document: {preview_text}")
                logging.info(f"Context: {result.get('context', '')}")
                logging.info("-" * 50)

        logging.info("[Step 04] Retrieval completed.")
    except Exception as e:
        logging.error(f"Error in step04_retrieve_relevant_chunks: {e}")

def step05_generate_response(args):
    """ Step 05: Uses LLM to generate an augmented response."""
    logging.info("[Step 05] Response generation started.")

    try:
        llm_client = LLMClient(llm_api_url=config.get("llm_api_url"),
                    llm_model_name=config.get("llm_model_name"))

        # llm_response = llm_client.query(args.query_args)
        # logging.info("\nLLM Response:\n", llm_response)
        # print("\nLLM Response:\n", llm_response)

        retriever = ChromaDBRetriever(vectordb_dir=config.get("vectordb_directory"),
                    embedding_model_name=config.get("embedding_model_name"),
                    collection_name=config.get("collection_name"))

        processor = RAGQueryProcessor(llm_client=llm_client,
                        retriever=retriever,
                        use_rag=args.use_rag)
        response = processor.query(args.query_args)
        print("\nResponse:\n", response)

        return response

        logging.info("[Step 05] Response generation completed.")
    except Exception as e:
        logging.error(f"Error in step05_generate_response: {e}")

def main():

    print("rag_pipeline starting...")
    """ Command-line interface for the RAG pipeline."""
    parser = argparse.ArgumentParser(description="CLI for RAG pipeline.")
    parser.add_argument("step",
                choices=["step01_ingest",
                    "step02_generate_embeddings",
```

```
                    "step03_store_vectors",
                    "step04_retrieve_chunks",
                    "step05_generate_response"],
                help="Specify the pipeline step.")

    parser.add_argument("--input_filename",
                nargs="?",
                default=None,
                help="Specify filename or 'all' to process all files in the input directory. (Optional)")

    parser.add_argument("--query_args",
                nargs="?",
                default=None,
                help="Specify search query arguments (enclosed in quotes) for step 4. (Optional,
required for step04_retrieve_chunks)")

    parser.add_argument("--use_rag",
                action="store_true",
                help="Call vectordb for RAG before sending to LLM (Optional, required for
step05_generate_response)")

    args = parser.parse_args()

    # Ensure that query_args is required only when using step04_retrieve_chunks
    if args.step in ["step04_retrieve_chunks", "step05_generate_response"] and args.query_args is
None:
        parser.error("The 'query_args' parameter is required when using step04_retrieve_chunks or
step05_generate_response.")

    if args.step == "step05_generate_response" and args.use_rag is None:
        parser.error("The 'use_rag' parameter is required when using step05_generate_response.")

    #
    setup_logging(config.get("log_level", "DEBUG"))

    logging.info("------ Command line arguments -------")
    logging.info(f"{'step':<50}: {args.step}")
    logging.info(f"{'input_filename':<50}: {args.input_filename}")
    logging.info(f"{'query_args':<50}: {args.query_args}")
    logging.info(f"{'use_rag':<50}: {args.use_rag}")
    logging.info("------ Config Settings -------")
    for key in sorted(config.to_dict().keys()):
        logging.info(f"{key:<50}: {config.get(key)}")
    logging.info("------------------------------")
    ensure_directories_exist(config)

    steps = {
```

```
        "step01_ingest": step01_ingest_documents,
        "step02_generate_embeddings": step02_generate_embeddings,
        "step03_store_vectors": step03_store_vectors,
        "step04_retrieve_chunks": step04_retrieve_relevant_chunks,
        "step05_generate_response": step05_generate_response
    }

    try:
        steps[args.step](args)
        logging.info("RAG pipeline done")
    except Exception as e:
        logging.error(f"Error in {args.step}: {e}")

def check_things():
    print("checking things")
    print("1. creating ChromaDBRetriever")
    retriever = ChromaDBRetriever(vectordb_dir=config.get("vectordb_directory"),
                                  embedding_model_name=config.get("embedding_model_name"),
                                  collection_name=config.get("collection_name"))

    print(f"Collection count: {retriever.collection.count()}")

if __name__ == "__main__":
    main()
```

9.2.10 test_document_ingestor.py

```
import unittest
from classes.document_ingestor import DocumentIngestor

class TestDocumentIngestor(unittest.TestCase):

    def setUp(self):
        self.ingestor = DocumentIngestor(file_list=["test.pdf"], input_dir="data/test/raw_input",
output_dir="data/test/cleaned_text", embedding_model_name="sentence-transformers/all-mpnet-
base-v2")

    def test_extract_text_from_pdf(self):
        text = self.ingestor._extract_text_from_pdf("data/test/raw_input/test.pdf")
        self.assertIsInstance(text, str)

    def test_clean_text(self):
        cleaned_text = self.ingestor._clean_text("This is a test text.")
        self.assertIsInstance(cleaned_text, str)

if __name__ == "__main__":
    unittest.main()
```

9.2.11 test_embedding_preparer.py

```
import unittest
from classes.embedding_preparer import EmbeddingPreparer

class TestEmbeddingPreparer(unittest.TestCase):

    def setUp(self):
        self.preparer = EmbeddingPreparer(file_list=["test_cleaned.txt"], input_dir="data/test/cleaned_text", output_dir="data/test/embeddings", embedding_model_name="sentence-transformers/all-mpnet-base-v2")

    def test_generate_embedding(self):
        embedding = self.preparer._generate_embedding("This is a test text.")
        self.assertIsInstance(embedding, list)

if __name__ == "__main__":
    unittest.main()
```

9.2.12 test_embedding_loader.py

```
import unittest
from classes.embedding_loader import EmbeddingLoader

class TestEmbeddingLoader(unittest.TestCase):

    def setUp(self):
        self.loader = EmbeddingLoader(cleaned_text_file_list=["test_cleaned.txt"], cleaned_text_dir="data/test/cleaned_text", embeddings_dir="data/test/embeddings", vectordb_dir="data/test/vectordb", collection_name="collections")

    def test_load_cleaned_text(self):
        text = self.loader._load_cleaned_text("data/test/cleaned_text/test_cleaned.txt")
        self.assertIsInstance(text, str)

    def test_load_embeddings(self):
        embeddings = self.loader._load_embeddings("data/test/embeddings/test_cleaned.json")
        self.assertIsInstance(embeddings, list)

if __name__ == "__main__":
    unittest.main()
```

9.2.13 test_llm_client.py

```
import unittest
from classes.llm_client import LLMClient

class TestLLMClient(unittest.TestCase):

    def setUp(self):
```

```
        self.client = LLMClient("http://localhost:1234/v1/completions", "llama-3.2-1b-instruct")

    def test_query(self):
        response = self.client.query("What is the capital of France?")
        self.assertIsInstance(response, str)

if __name__ == "__main__":
    unittest.main()
```

9.2.14 test_config_manager.py

```
import unittest
from classes.config_manager import ConfigManager

class TestConfigManager(unittest.TestCase):

    def setUp(self):
        self.config = ConfigManager("config.json")

    def test_get(self):
        self.assertEqual(self.config.get("embedding_model_name"),         "sentence-transformers/all-
mpnet-base-v2")

    def test_get_nonexistent_key(self):
        self.assertIsNone(self.config.get("nonexistent_key"))

if __name__ == "__main__":
    unittest.main()
```

9.2.15 test_rag_query_processor.py

```
import unittest
from unittest.mock import MagicMock
from classes.rag_query_processor import RAGQueryProcessor
from classes.llm_client import LLMClient
from classes.chromadb_retriever import ChromaDBRetriever

class TestRAGQueryProcessor(unittest.TestCase):

    def setUp(self):
        self.maxDiff = None
        # Mock LLMClient
        self.llm_client = MagicMock(spec=LLMClient)
        self.llm_client.query.return_value = "Mocked LLM response"

        # Mock ChromaDBRetriever
        self.retriever = MagicMock(spec=ChromaDBRetriever)
        self.retriever.query.return_value = [
```

```
            {"context": "Context about France", "id": "1", "score": 0.9, "text": "France is a country in
Europe."}
        ]

        self.processor_rag = RAGQueryProcessor(llm_client=self.llm_client, retriever=self.retriever,
use_rag=True)
        self.processor_non_rag = RAGQueryProcessor(llm_client=self.llm_client,
retriever=self.retriever, use_rag=False)

    def test_query_with_rag(self):
        query_text = "What is the capital of France?"
        response = self.processor_rag.query(query_text)
        self.assertIsInstance(response, str)
        self.assertNotEqual(response, "Error: Could not process the query.")

        # Capture the prompt sent to the LLM
        self.llm_client.query.assert_called_once()
        prompt = self.llm_client.query.call_args[0][0]

        expected_prompt = (
            "Please answer the following user query based strictly on the provided constitutional context.
"
            "Do not rely on external knowledge, assumptions, or unsupported information. "
            "Your response should be a concise single-paragraph, no longer than 200 words. "
            "Ensure that the response is clear, concise, and directly addresses the query. "
            "Avoid any repetition or redundant information. Focus on providing a unique and
comprehensive answer.\n\n"
            "Context:\nContext about France\n\n"
            "User Query:\nWhat is the capital of France?\n\n"
            "Response:"
        )
        print(prompt)
        self.assertEqual(prompt, expected_prompt)

    def test_query_without_rag(self):
        query_text = "What is the capital of France?"
        response = self.processor_non_rag.query(query_text)
        self.assertIsInstance(response, str)
        self.assertNotEqual(response, "Error: Could not process the query.")

        # Capture the prompt sent to the LLM
        self.llm_client.query.assert_called_once()
        prompt = self.llm_client.query.call_args[0][0]

        expected_prompt = query_text
        self.assertEqual(prompt, expected_prompt)
```